Functional Adaptive Programming with DemeterF

Bryan Chadwick Karl Lieberherr

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

Abstract

In this paper we present a new functional traversal abstraction for processing OO data structures that decomposes traversal computation into three function objects and a traversal control function. Function objects compute and combine values over a general traversal while the control function allows programmers to limit the extent of a traversal. Our new abstraction is supported by a Java library, called DemeterF, that allows programmers to use OOP techniques to develop traversal related programs. The library provides a rich set of default traversal behavior and a multiple dispatch mechanism to match methods during data structure traversal. We demonstrate the usefulness of our library by developing a type checker and evaluator for a small functional OO language.

1. Introduction

Data structure traversal is used in all forms of data processing, from programming language implementations to XML processing. In Object Oriented (OO) Languages the separation of interface and implementation in makes specification of different traversals across classes difficult. Patterns and domain specific languages (traversals/strategies) (Ovlinger and Wand 1999; Lieberherr 1996) have provided solutions to this problem by allowing functions (or objects) with local state (*i.e.*, visitors) to be executed over a specific instance of a data-structure.

While previous OO solutions allow one to express traversals, both internal and external to the data structure, they rely solely on state mutation for computation. This makes some forms of computation (data transformations/rewrites) cumbersome to implement in the given abstraction. In this paper we present an innovative functional abstraction for processing objects that leverages the power and flexibility of OOP and ideas from structure-shy and functional programming to provide state-free computation over a data structure. We introduce a Java library, called DemeterF, that supports our abstraction, giving the programmer control over object traversals and the values they produce. Our new abstraction provides the following benefits:

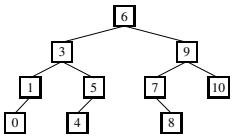
- Separation of traversal and control in a functional setting
- Rich traversal specialization and default behavior
- Dynamic traversal library with static type checking
- No data structure changes or language extension required

Structure-shy programming (Lieberherr 1996) allows a program to mention only the data-types of interest in a computation, while uninteresting types receive some default behavior. In most cases this means separating traversal code, *where-to-go*, while using a visitor instance to compute results, *what-to-do*. In an imperative setting *what-to-do* usually modifies local visitor state, making changes during traversal. At uninteresting nodes we simply do nothing.

In the functional setting, we wish to traverse an instance of a structure and produce values at interesting nodes; computing some value from the results. To support this style of programming we have separated traversal computation, *what-to-do*, into three *function objects*[1], while a control function tells the traversal *where-to-go*. Our DemeterF library provides general traversal and control functions, with suitable defaults for each of the function objects. This provides traversal flexibility while allowing the programmer to focus on the portions of the data structure related to a computation. In some cases the programmer need only deal with the type to be transformed, instead of the types that may be near it in the structure.

Typical solutions in the functional community (Lämmel and Peyton Jones 2003; Lämmel and Visser 2002) focus on two specific types of computation: type-preserving transformations and type-unifying folds. Our decomposition of

[1] We define a *function object* as an instance of a class that contains methods and class or instance specific constants, representing a set of functions/methods.




```
abstract class Comp<T>{
  abstract boolean comp(T a, T b);
}
class LT extends Comp<Integer>{
  boolean comp(Integer a, Integer b){ return a < b; }
}
class BST<T>{
  BST<T> insert(T d, Comp<T> c)
   { return new Node<T>(d, this, this); }
}
class Node<T> extends BST<T>{
  T data;
  BST<T> left, right;

  Node(T d, BST<T> l, BST<T> r)
   { data = d; left = l; right = r; }
  BST<T> insert(T d, Comp<T> c){
     if(c.comp(d, data))
       return new Node<T>(data, left.insert(d,c), right);
     return new Node<T>(data, left, right.insert(d,c));
  }
}
```

Figure 1. BST Implementation

what-to-do makes these forms of computation special cases. Because our traversal is separate, we can independently control *where-to-go*, and focus on *what-to-do* with traversal results. With a static form of traversal specification our dynamic traversal can be type-checked as if it was static, using a class-dictionary (or *type schema*) to determine what values sub-traversals will produce. Being written as a reflective Java library, DemeterF does not require any changes to the language, existing classes, or the compile process; it simply runs along side other programs.

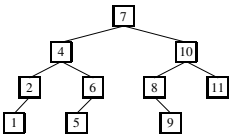
2. Motivating Example

As a concrete example, consider a typical generic BST implementation in Java (Figure 1). A `BST<Integer>` instance constructed with `LT` is drawn above the code for reference.

Using this definition there are several functions we may wish to compute. Traditional visitors (Gamma et al. 1995; Palsberg and Jay 1998) are good for computing aggregate values, *e.g.*, the sum of all numbers in a given tree, or its string representation, but suppose we would like to create a new tree where some data elements are modified? For instance, given a `BST<Integer>` we might like to construct a new one with each data element incremented, leaving the old tree alone.

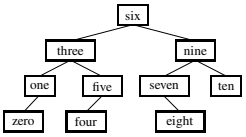
Figure 2 shows a class that implements the increment operation using our DemeterF library. The structure-shy element of this program comes from the default traversal behavior that rebuilds the underlying data structure[2]. In fact, this class can be used for any structure that contains **`int`**s

[2] Similar to SYB transformations (Lämmel and Peyton Jones 2003)




```
class Incr extends IDf{
  int apply(int i){ return i+1; }
}
```

Figure 2. Functional BST Increment




```
class Strs extends IDf{
  static String nums[] = {"zero","one", /*...*/,
                          "nine","ten"};
  String apply(int i){ return nums[i]; }
}
```

Figure 3. Functional BST Conversion

that we would like to increment, *e.g.*, lists, queues or stacks. The class can then be used in a traversal with something like the statement below, where `aBST` is the tree we would like to transform.

```
BST newBST = new Traversal(new Incr()).traverse(aBST);
```

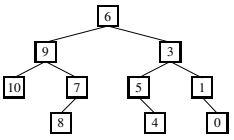
Since our function objects are just Java classes, we can leverage the power of Java generics to produce more than just `BST<Integer>`s. Figure 3 shows a class that converts a `BST<Integer>` into a `BST<String>` that contains the strings of the English words corresponding to the numbers in the tree.

As an example of a non-standard transformation consider the task of reversing a given BST. At every `Node` we wish to swap its `left` and `right` BSTs. Figure 4 shows a class that implements this transformation. At each `Node` we *combine* recursive results into a new reversed node. These examples use the default traversal control that proceeds *everywhere*. In later sections we introduce the traversal control abstraction that allows us to both optimize traversals (ignoring uninteresting portions of a data structure) and implement more complex recursive algorithms using our generic traversal.

The rest of this paper is organized as follows: Section 3 describes DemeterF traversals, function objects, and their semantics. Section 4 describes our library classes and implementation. As an larger example, we use our library to implement an interpreter for a functional OO language, presented in Section 5. We review related work in Section 6 and conclude with Section 7.

3. Functional AP

DemeterF traversals merge ideas prevalent in functional programming with those found in OOP and Adaptive Program-




```
class Rev extends IDb{
  BST combine(BST leaf){ return leaf; }
  BST combine(Node t, Object d, BST lf, BST rt)
   { return new Node(d, rt, lf); }
}
```

Figure 4. Functional BST Reverse

ming (AP) (Lieberherr 1996). Recursive traversals in functional languages are written in an elegant way, but usually repeat common structure. Typical support for transformations (Lämmel and Peyton Jones 2003; Lämmel and Visser 2002) relieve the programmer of boilerplate code, but remove a lot of the flexibility programmers rely on to implement algorithms. To support the flexibility of hand-written traversal code and avoid boilerplate code in typical situations we abstract traversals to allow functional computation with the kind of traversal control typically found in AP.

3.1 DemeterF Traversals

A complete DemeterF traversal is defined by three function objects (*what-to-do*) and a traversal control function (*where-to-go*). The function objects (or *sets* of functions), which we call *transformers*, *builders* and *augmentors*, manipulate values over a predefined recursive traversal aided by a multiple dispatch mechanism. The control function decides which fields of a data type should be traversed[3].

The traversal function, $T_{f,\beta,\alpha,c}$, and related abstractions are described in Figure 5. We use sets of functions to describe our algorithms, though our implementation uses function objects. The traversal is divided into two cases: user defined types, represented abstractly as a sequence of *fields*; and *BuiltIn* types(*e.g.*, `int`, `boolean`, *etc.*). The traversal accepts an extra argument, d_a in the figure, which is *updated* before traversing any fields of a data type.

To make the traversal sufficiently general, it is parametrized by function objects and a control function that represent aspects of hand-coded traversals:

f : Transformations; run at each node of the data structure

β : Reconstruction or folds using values from sub-traversals

α : Modification or replacement of a traversal argument.

c : Decide which sub-traversals of fields should be run.

When traversing a user defined data type (Figure 5) we first choose a function in α to *update* the traversal argument before processing any fields; this allows information to be passed *down* during traversal. We then traverse each *field* of the data type if the control function returns *true*, otherwise we simply transform it, with f, passing the new traversal argument in either case. Once all fields have been completed, we select a function from β to *combine* the result values; then give f a final chance to transform the combined result before returning it to the caller.

[3] Field numbers and names are interchangeable; though our specification uses numbers, our implementation uses names.

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

Figure 6. Default Function Definitions

As with most functional traversals, we have formulated the traversal function to minimize data dependencies between individual values making it simple to parallelize. Each calculation of d'_i can be done in separate threads, implicitly synchronizing on the dispatch to β. Separating the traversal computation into sets of functions also increases opportunities for reuse; allowing our implementation to provide suitable defaults for common scenarios.

Figure 6 describes the default functions we have found useful in practice. The id_f and id_α functions are straightforward, but the builders id_β and β_c are special. The behavior of id_β being *error* helps with runtime debugging of programs, while the constructing builder, β_c, calls the constructor of type C (the type of D) passing the traversal results as new fields. Assuming consistent constructor definitions, we can use β_c to do functional updates as shown in the earlier BST examples (Figures 2 and 3). The `Rev` class (Figure 4) that reverses a BST is an example of a class that extends id_β; covering both data type cases eliminating any chance for errors. To create a traversal using a `Rev` object we implicitly use id_f and id_α, though traversal arguments are not needed within our *combine* methods[4].

3.2 Dispatch: Function Selection

Our dispatch function, δ, selects the most specific function from a set (or object) based on the types of *all* actual arguments during traversal. Figure 7 describes our algorithm for function dispatch where $\prec$ is the traditional transitive, antisymmetric subtype relation and $\preceq$ is its reflexive extension. To select a function we first filter the set, leaving only those applicable to prefixes of the given argument types. We can then sort the functions in G' based on the defined comparison function *better*; applying the *least* (most specific) function, g, to the first m arguments provided.

The *filter* step and implementations of *better* and *moreSpecific* are chosen to allow later function arguments to be

[4] Traversal arguments are optional in DemeterF.

$$
\begin{array}{lll}
T_{f,\beta,\alpha,c}(D,\ d_a)\ \Rightarrow\ \textbf{if}\ D \equiv (d_0, \ldots, d_n)\ \textbf{then} & & :: \text{ user defined types} \\
\quad \textbf{let}\ d'_a = \delta(\alpha,\ (D,\ d_a)) & & -\text{ traversal argument } \textit{update} \\
\quad\quad d'_i = \textbf{if}\ c\,(D,\ i) & & -\text{ traverse the } \textit{field}? \\
\quad\quad\quad \textbf{then}\ T_{f,\beta,\alpha,c}(d_i,\ d'_a) & & -\text{ yes!} \\
\quad\quad\quad \textbf{else}\ \delta(f,\ (d_i,\ d'_a)) & & -\text{ no, just } \textit{apply} \text{ an } f \\
\quad\quad \hat{D} = \delta(\beta,\ (D,\ d'_0, \ldots, d'_n,\ d_a)) & & -\ \textit{combine} \text{ Results} \\
\quad \textbf{in}\ \delta(f,\ (\hat{D},\ d_a)) & & -\ \textit{apply} \text{ an } f \\
\textbf{else}\ \delta(f,\ (D,\ d_a)) & & :: \textit{BuiltIn} \text{ (primitive) types}
\end{array}
$$

f, β, and α are sets of functions; c is a two argument predicate.

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

Figure 5. DemeterF Traversal Algorithm

$$
\begin{array}{l}
\delta(G,\ (a_1 : C_1 \ldots a_n : C_n))\ \Rightarrow \\
\quad \textbf{let}\ G' = \{\ g(S_1 \ldots S_m) \in G \mid m \leq n\ \wedge\ \forall i \leq m.\ C_i \preceq S_i\ \} \\
\quad\quad g = head(\,sort(G',\ better)) \\
\quad\quad m = arity(g) \\
\quad \textbf{in}\ g(a_1 \ldots a_m) \\
better(\,g(S_1 \ldots S_n),\ h(U_1 \ldots U_m))\ \Rightarrow \\
\quad (n > m)\ \textbf{or}\ (n = m\ \textbf{and}\ moreSpecific\,((S_1 \ldots S_n),\ (U_1 \ldots U_n),\ n)) \\
moreSpecific\,((S_1 \ldots S_n),\ (U_1 \ldots U_n),\ n)\ \Rightarrow \\
\quad (n = 0)\ \textbf{or}\ (S_1 \prec U_1)\ \textbf{or}\ (S_1 = U_1\ \textbf{and}\ moreSpecific(\,(S_2 \ldots S_n),\ (U_2 \ldots U_n),\ n-1))
\end{array}
$$

Figure 7. DemeterF Function Dispatch Algorithm

optional. Sorting functions with more arguments to the front of the list is a consequence of optional arguments, which allows more general functions with a greater number of arguments to be selected ahead of those with fewer but more specific arguments. This also avoids the algorithmic (and theoretic) complexity of comparing function types with different numbers of arguments.

The function *moreSpecific* compares equal length sequences of argument types, stopping at the first inequality. This ensures that arguments at the front of the signature are given priority in function selection. It also compliments the original data element being the first argument— the most important argument for structuring traversal code is also the most important in function dispatch. These functions (δ, *better*, and *moreSpecific*) are implemented in DemeterF using sets built by reflecting on the function objects given when a traversal is created. We simply compare the types of traversal results with the sets of functions that parametrize each traversal as in the algorithm.

3.3 Type Checking Traversals

Another benefit of this functional traversal organization is the ability to type check traversals and results. We define a traversal type-error as the case when the filter step of the dispatch algorithm returns the empty-set. Because of the way our default functions have been defined, this can only occur when dispatching *builders*. Not surprisingly, this kind of error can be caught with static information about the data structures to be traversed, the function objects provided, and the traversal control function.

For simplicity of presentation, Figure 8 shows our three typing rules for DemeterF traversals ignoring traversal arguments and control. We reuse a modified form of the DemeterJ Class Dictionary (CD) syntax to differentiate between type definitions. *Sum* (or *union*) types are shown with '**:**' (meaning "*is supertype of*") using '|' to separate variants. Sum types represent abstract Java classes, specifying the reverse of extension. *Product* (or *record*) types are described with '**=**' using '$\langle \cdot \rangle$' for field definitions followed by their

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

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

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

Figure 8. DemeterF Traversal Typing Rules

type. They represent normal Java class definitions with any number fields.

When typing traversals, the judgment $\triangleright\ \tau_{f,\beta}(u) : s$ means traversing a value of type u returns a value of type s. The type dispatch function, Δ, follows the selection algorithm described earlier, but produces the return type of the chosen function. Though slightly informal[5], this description has been used to produce a static type checker for DemeterF, written in DemeterF. Static traversal control adds a few special cases to the presentation but does not affect our ability to check traversals for violations. The need for type safety has driven our choice of static traversal control in DemeterF.

4. DemeterF Library

The DemeterF library (Chadwick 2008) contains generic traversal and function classes written in pure Java that use reflection for data structure traversal and argument matching dispatch. It provides a simple Java translation of $T_{f,\beta,\alpha,c}$ (Figure 5), the dispatch function, δ, and various combinations of the default function sets defined in Figure 6.

We use *function objects* to represent sets of functions, which allows users to override and overload methods to extend or separate functionality and support new data structures. To differentiate the three types of functions within the same object we use a different method name for each. The various sets of functions f, β, and α are implemented by writing `apply()`, `combine()`, and `update()` methods respectively. This allows users to assemble function objects that implement a number of methods of any kind.

Figure 9 describes provided class names and the implementation of traversal related functions and objects. Most of the default implementations are as simple as the one below.

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

Programmers can then use Java inheritance to implement desired functionality over the traversal.

A `Traversal` instance is constructed with instances of `Function`, `Builder`, `Augmentor`, and `EdgeControl`. The static factory method `EdgeControl.everywhere()` is used to create the default traversal control function, an implementation of *everywhere*. Users can also use `EdgeControl.create(...)` to specify `Edges` (class/field-name pairs) that should not be traversed. Figure 10 describes a few of the provided `Traversal` constructors and default function choices for each case. The various combinations of function objects implement multiple interfaces, allowing programmers use a single class to implement a traversal solution.

The `Incr` and `Rev` classes from Figures 2 and 4 are examples that extend `IDf` and `IDb`. `Incr` implements an `apply` method that transforms integers, relying on the default builder, `Bc`, to rebuild `Nodes` during traversal. `Rev` can be

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

`Traversal`	Generic reflective traversal function
`EdgeControl`	Allows control over field traversal
`Function`	interface of `IDf`
`Builder`	interface of `IDb`
`Augmentor`	interface of `IDa`
`IDf`	Java implementation of id_f
`IDb`	Java implementation of id_β
`Bc`	Java implementation of β_c
`IDa`	Java implementation of id_α
`ID`	Java implementation of $(id_f \cup id_\beta \cup id_\alpha)$
`IDfa, IDfb, IDba`	Various default combinations

Figure 9. DemeterF Provided Classes & Function Objects

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

Figure 10. DemeterF Default Traversal Constructions

```
class Height extends IDba{
  int update(Node n, int h){ return h+1; }
  int combine(BST l, int h){ return h; }
  int combine(Node t, Object d, int l, int r)
   { return Math.max(l,r); }
}
```

Figure 11. BST Top-down Height using *update*()

used in a traversal with `IDf` (the default) or any other transformer. If paired with `Incr` we get a traversal that increments and reverses a given BST.

The function object that we have not seen is an augmentor. Figure 11 shows a function class that calculates the height of a BST *top-down*. The traversal argument is placed at the end of our method argument lists. At a `Node` we increase the height argument by 1; when reaching a non-`Node` BST, we return the accumulated height. After sub-traversal completes at a `Node`, within the `combine` method we return the greater height of the two sub-trees. The traversal argument is ignored since it does not affect the height calculation. The method dispatch allows signatures to leave out later arguments for situations when they are not needed.

Traversal control in DemeterF is encapsulated in a the `EdgeControl` class. Instances of the class decide which fields to traverse and which classes are considered *BuiltIns* on a per-traversal basis. Declaring classes as builtins allows the programmer to define new *leafs* of the data structure, cutting off traversal at such instances. Adding `Edges` to be bypassed gives programmers concise control over the depth of specific portions of the traversal. Control is important both for optimizing traversals, when results do not affect a computation, and implementing algorithms over possibly recursive objects.

5. Extended Example: FOOP

As a real example of data structure traversals and computation using DemeterF we discuss the implementation of an interpreter for a Functional, OOP Language we'll call *FOOP*. The FOOP syntax is a subset of Java that only allows assignment to fields within a class constructor. Our syntax for the major structures of FOOP are shown in Figure 12. We leave out the syntax for *Exp* as it will be discussed later.

What (reasonable) programming language is complete without a definition of *factorial*? A factorial program in FOOP is shown in Figure 13. From this example, a few differences from Java are obvious. The first is that FOOP uses if/then *expressions*; this is done to avoid the need for statements and assignments. The second is that constructors are required. To simplify its presentation (and type-checking/interpretation) we have eliminated class extension. The other major change (not evident here) is the removal of explicit field access; fields are only available implicitly within methods of the class to which they belong. Methods are implicitly public, but the implicit parameter **`this`** is not available within constructors, so methods cannot be called

```
Program    := ClassList Exp
ClassDef   := "class" Ident "{" FieldList MethodDefs "}"
ClassList  := ClassDef*
Field      := Type Ident ";"
FieldList  := Field*
MethodDefs := Constructor MethodList
Constructor := Ident "(" FormalList ")" "{" Assign* "}"
Assign     := Ident "=" Exp ";"
Method     := Type Ident "(" FormalList ")" "{" Def* Return "}"
Def        := Type Ident "=" Exp ";"
Return     := "return" Exp ";"
MethodList := Method*
FormalList := [ Type Ident [ "," FormalList ] ]
Type       := "int" | "boolean" | Ident
```

Figure 12. FOOP Syntax

```
class Fact{
  Fact(){}
  int fact(int i){
     return if (i < 2) then 1
            else i * this.fact(i + -1);
  }
}

new Fact().fact(7) // = 5040
```

Figure 13. Factorial Program in FOOP

until an object is fully constructed. To simplify the parsing and structures we introduce only four binary operators (addition, multiplication, less-than, and conjunction) and a unary operator for negation (hence "`+ -1`" in `fact`).

5.1 Parsing Translation

For the implementation of FOOP we chose to use the programming tool DemeterJ (The Demeter Group 2007) to generate Java classes and a corresponding parser from a mix of concrete and abstract syntax description known as a Class Dictionary (CD) file. Creating parsable data structures poses the limitation that they must be LL(k) and cannot be generalized; parsing `List<X>` requires a concrete data definition[6]. Infix expression parsing provides its own difficulties as the hierarchy of operations must be built to ensure correct precedence ordering.

To alleviate the eventual type checking and evaluation traversals from the hassles of overly verbose data structures we use a translation step to reduce more complicated `Exps` to simpler, more programmer (and traversal) friendly structures. Figure 14 shows a small portion of the FOOP CD file that encodes the precedence between addition and multiplication; `Term` is a subtype of `Exp`. We use an interface (`TermI`) to simplify the translation of `Terms` and `TermLists` to `AddExps`. The fields are not named because we will not use them after the parse tree is translated. Similar structures are repeated for parsing `Conjunct` ('&&'), `Compare` ('<'), and `Factor` ('*') expressions, introduc-

[6] This is a limitation of the DemeterJ class and parser generation.

```
//* In place of parsing generics...
interface ConsList = .
interface EmptyList = .

//* Parse the complicated...
Term =  Factor TermList extends Exp implements TermI.
TermList: TermCons | TermEmpty common implements TermI.
TermCons = "+" Factor TermList implements ConsList.
TermEmpty = implements EmptyList.
interface TermI = .

//* Want the much simpler...
AddExp = <left> Exp *s <right> Exp.
```

Figure 14. Term Parse CD Snippet

```
Exp combine(Exp h, Exp left, EmptyList el)
 { return left; }
Exp combine(ConsList h, Exp left, EmptyList el)
 { return left; }
Exp combine(ConjunctI h, Exp left, Exp right)
 { return new AndExp(left, right); }
Exp combine(CompareI h, Exp left, Exp right)
 { return new LessExp(left, right); }
Exp combine(TermI h, Exp left, Exp right)
 { return new AddExp(left, right); }
Exp combine(FactorI h, Exp left, Exp right)
 { return new MultExp(left, right); }
```

Figure 15. Parser `Translate` Class Snippet

```
MethodBody: Return | RevDef.
Return =  "return" *s <ret> Exp ";".
RevDef = <exp> Exp <rest> RevDefRest.
RevDefRest = <type> Type <id> Ident <rest> MethodBody.

TypePair = <bind> Type <rest> Type.
```

Figure 16. Abstract Syntax for Reversed `Defs`

ing analogous interfaces, `ConjunctI`, `CompareI`, and `FactorI`, used in Figure 15.

Once parsed, the data structures are traversed and restructured to produce simpler binary expressions. A portion of this code (from the `Translate` class) is shown in Figure 15. We translate the rightmost expression into itself (first two cases) and convert other kinds of complex expressions into the corresponding simpler expressions. The two list interfaces, `ConsList` and `EmptyList`, allow our methods to be more general. To reconstruct non-`Exps` (*e.g.*, Method bodies) we create a traversal that goes to all `Exps` and transforms them using a `Translate` instance. In addition to rewriting expressions, we also transform variable definitions, `Defs`, into `RevDefs` to reverse the binding and expression into a nested structure; the related classes are shown in Figure 16. After this translation evaluation is simplified and type checking can proceed without any traversal intervention.

5.2 Type Checking FOOP Programs

As defined, FOOP is a language without *subtypes*, which makes method calls and field accesses significantly easier to type check. Figure 17 shows the mixed abstract/concrete syntax of primary expressions in FOOP and the `Types` and `Values` that are used during type checking and evaluation. We introduce `Types` and `Values` for integers, booleans, and user defined objects. `VarT` is a name/type pair that is used as an element of the generic class `Env<T>` representing various (*i.e.*, `Type` and `Value`) environments.

Type checking FOOP expressions can be done without traversal control because the language contains static type declarations. Figure 18 shows a class that implements a type checker for simple expressions. This class extends `ID`; the class that implements all three traversal function class interfaces. It is very similar to hand written functional type checkers with one exception: our traversal abstraction eliminates the need to write any traversal code. The two methods for `IfExp` use the argument matching to differentiate between valid and invalid cases. The first (more specific) method simply checks that the *then* and *else* expressions have the same type. The second catches all cases where the type of the condition expression is not *boolean*.

Figure 19 adds methods to support the type checking of methods to the simple expression type checker. The *update* methods add variables to the type environment. `ClassMeth` is a structure that contains the name of the class and a list of `MethodDescs` for a given class. When traversal reaches a `ClassMeth` we add the fields of the class with their types and the special variable **`this`** to the environment with the type of the given class. When we reach a `MethodDesc`, we add the arguments to the type environment, and finally, when reaching a `RevDefRest` we can add the binding to the environment.

The combine method for *SymExp* looks up the variable type in the environment; the type of a `Return` is just the type of the inner expression. Reverse definitions require a `TypePair` to return both the type of the binding and the type of the nested expression. Once the expression within a `RevDef` has been typed we check it against the defined type, returning the result type if they match. We leave out the checking of constructors as it follows the same style; checking that the types of all assignments are correct. After type checking a FOOP program, we then convert variable names (`SymExps`) into stack addresses using a version of *de Bruijn* indices, making the evaluation function classes very easy to follow.

5.3 Evaluating FOOP Programs

Evaluation is the first traversal that requires a change to the default traversal control. We begin with an `Eval` class (Figure 20) that contains a `Traversal` as an optimization for implementing recursion. We use assignment to *tie the knot*, creating a traversal that will skip *then* and *else* of each `IfExp`, and the *rest* field of `RevDefs`. The way the language is defined requires each definition within a method to be evaluated in order, adding it to the environment before evaluating other definitions.

```
Primary: Negate | ParenExp | IntLit | BoolLit
       | VarExp | NewExp | IfExp.
Negate  = "-" Exp.
ParenExp= "(" Exp ")".
IntLit  = <value> int.
BoolLit = <value> boolean.
VarExp: SymExp.
SymExp = <id> Ident.
NewExp = "new" <type> Type "(" <args> ExpList ")".
IfExp  = "if" "(" Exp ")" "then" <thn> Exp
                      "else" <els> Exp.
```

```
Type: IntT | BoolT | UserT.
IntT  = "int".
BoolT = "boolean".
UserT = <name> Ident.

VarT  = <name> String <type> Type.

Value: IntV | BoolV | ObjV | NullV.
NullV = .
IntV  = <val> int.
BoolV = <val> boolean.
ObjV  = <type> Type <fields> ValueList.
```

Figure 17. *Left*: Primary Expression Syntax *Right*: Type and Value Structures

```
class ExpCheck extends ID{
  static Type intt = new IntT();
  static Type boolt = new BoolT();

  Type combine(IntLit il){ return intt; }
  Type combine(BoolLit il){ return boolt; }
  Type combine(AddExp e,  IntT l, IntT r){ return intt; }
  Type combine(MultExp e, IntT l, IntT r){ return intt; }
  Type combine(LessExp e, IntT l, IntT r){ return boolt; }
  Type combine(NegExp e,  IntT l){ return intt; }
  Type combine(AndExp e,  BoolT l, BoolT r){ return boolt; }
  Type combine(OpExp e){ throw new TypeErr("Bad OpExp"); }

  Type combine(IfExp e,   BoolT c, Type thn, Type els){
     if(thn.equals(els))return thn;
     throw new TypeErr("IfExp: Then & Else Mismatch");
  }
  Type combine(IfExp e, Type c)
   { throw new TypeErr("IfExp: Non-boolean Condition"); }
}
```

Figure 18. Simple expression type checker

```
class MethodCheck extends ExpCheck{
  MethodCheck(ClassList c){ /* ... */ }

  Env<VarT> update(ClassMeth c, Env<VarT> env){
     return env.push(classes.find(c.name).flds.env())
               .push(new VarT("this", new UserT(c.name)));
  }
  Env<VarT> update(MethodDesc m, Env<VarT> env){ return env.push(m.args.env()); }
  Env<VarT> update(RevDefRest d, Env<VarT> env){ return push(env, ""+d.id,d.type); }

  Type combine(SymExp s, Ident id, Env<VarT> env)
   { return env.find(new VarT(id)).type; }

  Type combine(Return r, Type t){ return t; }
  Type combine(RevDef h, Type exp, TypePair def){
     if(exp.equals(def.bind))return def.rest;
     throw new TypeErr("Def: Type Mismatch");
  }
  TypePair combine(RevDefRest h, Type b, Ident n, Type r){ return new TypePair(b,r); }

  Type combine(MethDesc h, String n, MethType mt, TypeList arg, Type body){
     if(!mt.ret.equals(body))
        throw new TypeErr("Method: Return Type Mismatch");
     return body;
  }
}
```

Figure 19. Method type checker

```
class Eval extends ID{
    static Env<Value> empty = Env.<Value>empty();
    Traversal trav;

    static Value doEval(Exp e, ClassList c){
        Eval eval = new MethodEval(c);
        EdgeControl ctrl =    //** Skip traversal of these 'Edges'
             EdgeControl.create(new Edge(IfExp.class, "thn"),
                                new Edge(IfExp.class, "els"),
                                new Edge(RevDef.class, "rest"));
        eval.trav = new Traversal(eval, ctrl);
        return eval.eval(e, empty);
    }
    Value eval(Object e, Env<Value> env){ return trav.traverse(e, env); }
}
```

Figure 20. Base Evaluator Class

We choose a simple stack for an environment, pushing values in order of: object fields, **this**, then method arguments. After rewriting variable expressions into stack addresses, the evaluation follows the format seen in the type checker; our FOOP evaluation classes are shown in Figure 21. For incremental development and testing it makes sense to divide the functionality into separate classes. We then add features to support more expressions. `LitEval` handles simple *Value*s; `ExpEval` adds support for binary and *if* expressions. `MethodEval` contains the evaluation of a `VarExp` variant (`AddrExp`) that represents the stack address of a variable (field, method argument, or local definition).

`CallExp` allows the evaluation of recursive calls, setting up the environment before evaluating the body of a method. For a `RevDef` we recall the traversal on the `rest` field with the new `Value` on the stack; remember that we did not traverse the `rest` field due to the `EdgeControl`. As with the type checking example, constructors are similar to method evaluation, dealing with assignments to a list of fields, returning an `ObjV`. Because we can use Java inheritence with function objects, we can divide the evaluator into modular units, which is useful for testing and organizing code.

6. Related Work

DemeterF's functional traversals and dynamic dispatch are closely related to several disjoint technologies in both the functional and OOP communities. The *Scrap Your Boilerplate* (Lämmel and Peyton Jones 2003, 2004, 2005) series and related papers on strategic programming (Lämmel et al. 2004; Lämmel and Visser 2002) discuss similar typed transformations through traversals. They divide traversal computation into two main cases: *type-preserving* (TP) and *type-unifying* (TU). The TP case is similar to our `Incr` example (Figure 2), where the given BST is transformed into another BST. Our `Height` example (Figure 11) is a form of TU traversal, as all methods return `int`. Our traversal decomposition makes TP and TU computations special cases while allowing programmers to express traversals that are not entirely TP or TU, *e.g.*, `MethodCheck` in Figure 19. Our main contribution in this space is the addition of traversal control from functional computation while maintaining traversal separation and the addition of a traversal arguments.

Our dispatch function is similar to ideas found in predicate and multiple dispatch. JPred (Millstein 2004) and MultiJava (Clifton et al. 2000) introduce a special syntax for multiple dispatch methods, available in general class definitions. Our version of multiple dispatch is strictly available during traversal on function objects. Though our library implementation could be used outside of the traversal, it is not meant for general class dispatch. It is not clear if our traversal dispatch could be implemented or statically generated for those languages, but traversals could certainly be written that take advantages of predicate of multiple dispatch similar to our decomposition.

The functional computations that result from our traversal organization are similar to attribute grammars (Knuth 1968). Our *augmentors* allow computation of inherited attributes, while other function objects can be used to synthesize attributes. Since our library is written in Java, traversal computation is similar to *Reference Attribute Grammars* (Hedin 2000) as we allow any Java value to be passed between traversal functions. Using our traversal function programmers must compute their own attributes, though our library could be used as a lower level implementation language for attribute evaluation.

The traversal and control found in DemeterF are clearly related to ideas from Adaptive Programming (AP) (Lieberherr 1996). AP specifies traversal computation using a domain specific *strategy* language (*where-to-go*) for use with visitors (*what-to-do*). Strategies are very expressive, combining static class descriptions and dynamic instance conditions (generally type existence) to control the extent of visitor method execution. A static description of the class hierarchy is used to guide dynamic traversal execution (Orleans and Lieberherr 2001) or static traversal method generation (Lieberherr et al. 2004). Besides removing the need for mutation in traversals, DemeterF allows only a static subset of traversal control found in other Demeter related tools. As

```
class LitEval extends Eval{
  Value combine(IntLit lit, int i){ return new IntV(i); }
  Value combine(BoolLit lit, boolean b){ return new BoolV(b); }
}

class ExpEval extends LitEval{
  Value combine(AddExp e,  IntV l, IntV r){ return l.add(r); }
  Value combine(MultExp e, IntV l, IntV r){ return l.mult(r); }
  Value combine(NegExp e,  IntV l){ return new IntV(-l.val); }
  Value combine(LessExp e, IntV l, IntV r){ return l.less(r); }
  Value combine(AndExp e,  BoolV l, BoolV r){ return l.and(r); }

  Value combine(IfExp f, BoolV c, Exp t, Exp e, Env<Value> env)
   { return eval((c.val?t:e), env); }
}

class MethodEval extends VarEval{
  ClassMethList mthds;
  MethodEval(ClassList c){ /* ... */ }

  Value combine(CallExp c, ObjV v, SymMethExp m, ValueList args, Env<Value> env){
    MethDesc meth = mthds.findMethod(v.type, m.id);
    Env<Value> nenv = empty.push(v.fields.env())
                           .push(v).push(args.env());
    return eval(meth.body, nenv);
  }
  Value combine(AddrExp a, int addr, Env<Value> env)
   { return env.get(addr); }

  Value combine(RevDef rd, Value v, RevDefRest rst, Env<Value> env)
   { return eval(rst.rest, env.push(v)); }
  Value combine(Return r, Value v){ return v; }
}
```

Figure 21. Evaluator Traversal Classes

traversals produce values, we limit control to eliminate possible typing issues at runtime.

Functional Visitors in DJ (Wu et al. 2003) are the most similar AP tool, though this is mainly because they are also functional. Functional visitors have methods that return values and a single `combine` method that is used to combine all sub-traversal values at each portion of the traversal, similar to SYB type-unifying transformations. Visitor methods all return the type `Object` while the `combine` method takes an `Object` array as an argument. Because our function objects can be written with more specific types, traversal can be statically verified. Our traversal decomposition is more flexible allowing separate combine methods for different parts of the traversal.

7. Conclusion and Future Work

We have introduced an innovative functional abstraction that merges ideas from structure shy and functional programming to support OO traversals. Traversal computation is decomposed into three function objects and a control function, which allow programmers to leverage the power and flexibility of OOP to write mutation free algorithms. Our new abstraction is supported by a Java library, called DemeterF, that uses reflection to implement data structure traversal and multiple argument dispatch for method execution. Our library provides a rich set of classes and default function objects that support structure shy programming without language or data structure changes.

In the future we will work towards proving type safety for our traversals and exploring more complex traversal control specifications. In addition we would like to consider alternative implementation techniques to enhance performance such as static code generation or traversal optimizations.

Acknowledgment: We would like to thank Ralf Lämmel for helpful comments on earlier versions of our work.

References

Bryan Chadwick. DemeterF library and examples. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

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

Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968. URL `http://dx.doi.org/10.1007\%2FBF01692511`.

R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, 2004. URL `citeseer.ist.psu.edu/lammel02essence.html`.

R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

Ralf Lämmel and Simon 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 Workshop on Types in Language Design and Implementation (TLDI 2003).

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

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

Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. URL `http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html`. 616 pages, ISBN 0-534-94602-X.

Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

Todd Millstein. Practical predicate dispatch. *SIGPLAN Not.*, 39 (10):345–364, 2004. ISSN 0362-1340.

Doug Orleans and Karl 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.

Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–81, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-238-7.

Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, Washington, DC, USA, 1998. ISBN 0-8186-8585-9.

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

Pengcheng Wu, Shriram Krishnamurthi, and Karl Lieberherr. Traversing recursive object structures: The functional visitor in demeter. In *AOSD 2003, Software engineering Properties for Languages and Aspect Technologies (SPLAT) Workshop*, 2003.