Functional Adaptive Programming

A dissertation presented

by

Bryan Chadwick

2010-08-12 16:07

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

August, 2010

Abstract

The development of complex software requires the implementation of operations over recursively defined data structures. Complex data structures lead to an increase of code dealing with structure access and navigation. This 'boilerplate' code in turn makes programs tedious to develop, difficult to maintain, prone to errors, and separates important functionality, all of which result in the loss of clarity. Generic (or polytypic) programming and higher order functions can resolve some of these issues, but are usually too general to be practically useful for large collections of data types.

This dissertation proposes a new approach to developing structure-based functions and describes an implementation of these ideas in Java, called DemeterF. Our approach uses function-objects over an adaptive traversal to implement deep, fold-like functions over data structures. Function-classes/objects provide a useful and flexible form of generic programming that adapts to different data structures using a type-based multiple dispatch. We model DemeterF with function sets and structural recursion, and give it a type system that shows our function-objects, multiple dispatch, and traversals can be checked for safety. In order to show that our approach is efficient we present the results of several performance tests comparing DemeterF to hand-written methods and visitor implementations in Java.

Contents

List of Figures

CHAPTER 1

Introduction

The development of complex software requires programmers to implement operations over a variety of recursively defined data structures. While the design and modeling of interesting data structures can be difficult, complex data leads to more complex operations. A typical side-effect of complex data structures is an increase in so-called *boilerplate* code that deals with data structure access and navigation. Boilerplate code makes programs tedious to develop, prone to errors, and difficult to maintain, and entangles important functionality, resulting in a loss of software clarity.

In object-oriented methodology this separation of interesting code is obvious (and extreme), since functions (*i.e.*, methods) are grouped by class. Abstract classes can be viewed as types, with concrete subclass akin to *value constructors*.[1] Each class contains all of its related methods, though common cases can be moved up to a common superclass.

In functional languages the separation of functionality is a bit more subtle. Since each function handles a single type, it usually contains all cases value constructors of a datatype. When implementing a single operation over small, self recursive datatypes, a single function will suffice. Complex data structures are, however, made up of multiple typically mutual recursive types. With each type requiring a separate function our initial cohesion is lost.

Taming increases in the structural complexity of software requires a different approach, and typical solutions differ by implementation language and paradigm. In class-based object-oriented languages, such as Java [29] or C# [32], the visitor design pattern [26] is traditionally used as a guide to separate operations from data structures. In functional languages like Haskell [38], ML [57], or Scheme [39], reusable computation is abstracted into *higher-order* functions of folds [35]. Neither one is a perfect solution.

The visitor pattern is useful for abstracting certain parts of an implementation, *e.g.*, case dispatch or data structure traversal, but there is a tension between safety and flexibility. Many instantiations of the pattern, including the original description, opt for imperative, *i.e.*, `void`, methods rather than

[1] More accurately, an abstract class is like a *type constructor* in, say, Haskell.

forcing clients to deal with return types, contravariance, and composition. Imperative methods force clients to implement functionality via mutation to shared state, which makes functions difficult, if not impossible, to understand. The pattern can be functionalized [42, 60], but not without inhibiting safety, flexibility, or data structure extension. Overall, the visitor pattern is simply a *pattern* for separating operations from structures, and is not meant to help programmers deal directly with complex data hierarchies.

In contrast, functional languages typically separate the definitions of functions and data structures. Since functions are values, programmers typically write structurally recursive functions, called *folds*, that act as recursion operators, replacing value constructors, like `cons`, with client functions. Generalized folds [54, 64] provide a blueprint for writing (or generating) fold abstractions for user-defined datatypes, but with more complex datatypes containing multiple or mutually recursive types, the number and order of functions that must be passed can quickly become overwhelming. Polytypic programming [37, 53, 27, 9] provides a similar service, allowing programmers to define functions that are applicable to *all* datatypes. There are situations where this is useful, but functions are usually defined over a universal datatype consisting of binary sum and product types. The universal nature of the definitions makes writing functions that operate on high level datatype notions, *e.g.*, term *reduction*, impossible.

1.1 My Thesis

This dissertation presents a complete reformulation of Adaptive Programming (AP) [51] to provide a flexible, safe, and efficient approach to writing side-effect free operations over complex data structures in an object-oriented setting. We refer to the approach as *functional adaptive programming*, or AP-F. The goals of AP-F are to: (1) eliminate the boilerplate code associated with the implementations of data structures and operations, (2) separate and modularize interesting functionality while remaining safe, and (3) provide maximum flexibility in both function and traversal implementations, while maintaining a high level of performance.

We achieve these goals by grouping methods of an operation into a *function-class*, instances of which (*i.e.*, *function-objects*) are applied over a data structure by a generic traversal. After an object's fields have been processed, the traversal uses a type-based multiple dispatch to select an appropriate method that matches the type of the current object and the types of the recursive results. The separation of traversal, function-objects, and dispatch supports four important features:

- *Function-class extension*: new methods can be added using normal object-oriented inheritance,
- *Case abstraction*: a single method can be called for multiple type cases,

- *Case specialization*: a more specific method/signature can override a more general one, and
- *Traversal control*: recursion can be easily controlled/limited for efficiency and/or algorithm correctness.

This brings me to my thesis:

> *Function-objects applied over data structure traversal are a useful, safe, and efficient way to write functions.*

In support of this thesis I have developed *DemeterF* [16, 19], a library and collection of tools that enable flexible, safe, and efficient traversal-based (generic) programming with function-objects/classes. My implementation combines features of object-oriented traversals, adaptive programming [51], higher-order functions, and multiple dispatch to support the development of traversal-based functions that are flexibly typed, provably safe, and efficient to execute.

1.2 Dissertation Outline

The rest of this dissertation is structured as follows:

- The rest of this chapter gives a background on functions, data structures, and traversals.
- In chapter 2 I introduce DemeterF, its class generator, traversal library. I use examples to demonstrate my approach to traversal-based programming with function-objects and how they can be used for both specific and generic programming.
- I then present a model of DemeterF's essential features in chapter 3, providing syntax, a small-step operational semantics, and a type system. I prove that our model of dispatch is type safe, meaning that well-typed traversals do not *go wrong*.
- Chapter 4 discusses abstractions of several algorithms involved in the implementation, safety, and performance of DemeterF-based programs.
- The final portion of my thesis is addressed in chapter 5, where I discuss empirical results of testing the performance of a number of DemeterF-based programs, including DemeterF itself.[2]
- Chapter 6 discusses previous works related to AP-F, DemeterF, and my implementation.
- Chapter 7 completes the dissertation with conclusions and a discussion of future work.

[2]DemeterF is, of course, implemented in DemeterF.

1.3 Background

I complete the introduction by giving a background of data structures, operations, and traversals, sprinkled with a brief review of related work.

1.3.1 Data Structures and Operations

There are two main schools of thought as to the definition and extension of data structures and operations. The first is well known by object-oriented programmers, where operations are grouped together with the structure to which they apply. I refer to this organization as *datatype-centric*, since it allows programmers to easily extend their datatypes, but adding new operations requires programmers to add a new method to each of the classes of related datatypes. On the other side is the more function-based approach that one uses, for instance, in Scheme. I refer to this as *function-centric*, since it allows programmers to easily add new functions to her system, but adding a new datatype case requires an addition to the functions of all related datatypes.

Of course, this data versus function dilemma is not new. It is traditionally called the *expression problem* (a term coined by Wadler [70]) and has been studied extensively [42, 73, 14, 67, 1, 59].[3] To make the idea more concrete, we will illustrate using representations of an abstract syntax tree (AST) of boolean expressions. Our initial structures will include boolean literals (`True` and `False`), unary negation, and binary `And` and `Or`.

1.3.1.1 Datatype-Centric: Java

We begin by representing the structures in a typical class-based object-oriented language, namely Java. Listing 1.1 shows straightforward class definitions that describe our types to represent boolean expressions, `BExp`s. We use

```
// Boolean Expression
abstract class BExp{}

// Literals
abstract class Lit extends BExp{}
class True extends Lit{}
class False extends Lit{}

// Negation
class Neg extends BExp{
   BExp inner;
   Neg(BExp inner){ /*...*/ }
}
```

```
// Binary And
class And extends BExp{
   BExp left, right;
   And(BExp left, BExp right)
   { /*...*/ }
}

// Binary Or
class Or extends BExp{
   BExp left, right;
   Or(BExp left, BExp right)
   { /*...*/ }
 }
```

Listing 1.1: Boolean expression structures in Java.

[3] Although it has been extensively studied, it has not been fully resolved.

abstract classes to represent disjoint unions, *e.g.*, BExp and Lit, and concrete classes to represent variants, *e.g.*, True and False. An instance of BExp that represents the expression $(true \wedge \neg false)$ can be constructed as follows:

```
new And(new True(),
        new Neg(new False()))
```

If we follow good object-oriented style, then implementing an operation over BExps requires that we add a new method to each class. Listing 1.2 shows an implementation of evaluation, eval(), which reduces a BExp to a Lit representing **true** or **false**. The comments above the methods describe into

```
// Added to BExp
abstract Lit eval();


// Added to Lit
abstract Lit eval();
abstract boolean isTrue();


// Added to True
Lit eval(){ return this; }
boolean isTrue(){ return true; }

// Added to False
Lit eval(){ return this; }
boolean isTrue(){ return false; }
```

```
// Added to Neg
Lit eval(){
  if(inner.eval().isTrue())
    return new False();
  return new True();
}
// Added to And
Lit eval(){
  if(left.eval().isTrue() &&
     right.eval().isTrue())
    return left;
  return new False();
}
// Added to Or
Lit eval(){
  if(!left.eval().isTrue() &&
     !right.eval().isTrue())
    return left;
  return new True();
}
```

Listing 1.2: Boolean expression evaluation in Java.

which class they should be placed. We use Lit rather than **boolean** in order to provide a fair comparison between later implementations in different languages.

Within the Lit and **BExp** classes we introduce an **abstract** method, eval() that will be implemented in concrete subclasses. For True and False we simply return **this** instance of Lit. To determine the representative **boolean** value of a Lit we introduce another **abstract** helper method, isTrue(), within Lit, and provide implementations in True and False. The eval methods for other concrete classes are straightforward, testing the result of recursive eval calls to subcomponents and returning the correct result.

Operation Extension Adding a new operation requires us to add a new method to each class. For example, adding an operation to convert a BExp to a String requires the addition of a toString() method to each class. Most method implementations will follow a similar pattern to eval, but will nonetheless be spread throughout our class definitions.

Datatype Extension In contrast, adding a new datatype is comparatively easy. For example, adding an `Xor` expression variant only requires that we add a new class, shown in Listing 1.3. The class **`extends`** `BExp`, and provides

```
class Xor extends BExp{
   BExp left, right;
   Or(BExp left, BExp right){ /*...*/ }

   Lit eval(){ /*...*/ }
   String toString(){ /*...*/ }
}
```

Listing 1.3: Datatype extension in Java

implementations for each of its operations, similar to `And` and `Or`[4]

1.3.1.2 Function-Centric: Scheme

Similar boolean expression structures can also be defined in a functional language. Listing 1.4 shows a structure-based boolean expression representation written in Scheme. Our use of **`define-struct`** is analogous to our

```
;; A BExp is one of Lit, Neg,...
;; A Lit is one of True or False
;; (make-True)
(define-struct True ())
;; (make-False)
(define-struct False ())
```

```
;; (make-Neg BExp)
(define-struct Neg (inner))
;; (make-And BExp BExp)
(define-struct And (left right))
;; (make-Or BExp BExp)
(define-struct Or (left right))
```

Listing 1.4: Boolean expression structures in Scheme.

concrete classes in Java. Comments take the place of abstract classes, documenting our intended unions, *e.g.*, `BExp` and `Lit`, and the expected values to be stored in our concrete structures. A value representing the expression $(true \wedge \neg false)$ can be constructed as follows:

```
(make-And (make-True)
          (make-Neg (make-False)))
```

In Scheme we can implement most operations on `BExp`s as a single function. We use a **`cond`** expression to differentiate between the structure variants, and decompose the concrete structure to compute a result. For example, Listing 1.5 shows an implementation of evaluation, `eval`, which reduces a `BExp` to a `Lit` representing `#t` or `#f`. Each defined structure has a **`cond`**-clause that handles its particular case, but the implementation is otherwise identical in spirit to the Java methods.

[4]We could actually have extended `And` or `Or` in order to partially reuse an implementation of `eval`, but this is clearer, and maintains our implicit ideal of subsumption.

```
;; eval: BExp -> Lit
(define (eval e)
  (cond [(True? e) e]
        [(False? e) e]
        [(Neg? e) (if (True? (eval (Neg-inner e)))
                      (make-False)
                      (make-True))]
        [(And? e) (if (and (True? (eval (And-left e)))
                           (True? (eval (And-right e))))
                      (And-left e)
                      (make-False))]
        [(Or? e) (if (and (False? (eval (Or-left e)))
                          (False? (eval (Or-right e))))
                     (Or-left e)
                     (make-True))]))
```

Listing 1.5: Boolean expression evaluation in Scheme

Datatype Extension Adding a new datatype requires us to add a new case to each previously defined function. Following our Java example, adding a new `Xor` structure requires us to update the implementation of `eval` by adding a new case to the **`cond`**. If there are more operations on `BExp`s then our necessary updates are scattered throughout the program.

Operation Extension On the other hand, adding a new operation only requires that we add a new function. Following our Java example, adding an operation to convert a `BExp` to a `string` is shown in Listing 1.6. The imple-

```
;; tostring: BExp -> string
(define (tostring e)
  (cond [(True? e) "true"]
        [(False? e) "false"]
        [(Neg? e) #/.../#]
        [(And? e) #/.../#]
        [(Or? e)  #/.../#]
        [(Or? e)  #/.../#]))
```

Listing 1.6: Operational extension in Scheme

mentation follows a similar pattern to `eval`, selecting branches of the **`cond`** based on the structure type, but all our functionality is centrally located.

1.3.2 Implementing Operations

While the organization of datatypes and operations differs in our two examples, they do have something in common. When implementing operations (functions and methods) over a data structure each requires a large amount of code that deals with structural recursion. Each point where our data structure is recursive, our function/method also has a recursive call.[5] In

[5]Sometimes recursive calls may be to *other* functions/methods that coordinate to implement the same operation.

many situations the code dealing strictly with structural recursion is almost identical across all operations. In this section we review solutions that attempt to remove structural boiler-plate code. We discuss the limitations of these solutions and mention related work and extensions to the approaches.

1.3.2.1 Visitors and Traversals

It is likely that every object-oriented programmer has seen or heard of the Visitor Pattern in one form or another. The original description by Gamma *et al*. [26] uses a combination of `accept` and `visit` methods to define operations over a collection of classes (*i.e.*, a class hierarchy). Operation specific behavior is modularized into a `Visitor` instance that implements `visit` methods. The `accept` methods are placed within the class hierarchy to implement a traversal over the structures while making calls to a visitor object's `visit` methods.

A Java imperative visitor interface, `ImpVis`, is given in Listing 1.7(a). Shown in Listing 1.7(b) is a functional visitor interface, `FunVis`, similar to

```
interface ImpVis{
   void visit(True e);
   void visit(False e);
   void visit(Neg e);
   void visit(And e);
   void visit(Or e);
}
```

(a) Imperative

```
interface FunVis<R>{
   R visit(True e);
   R visit(False e);
   R visit(Neg e, R inner);
   R visit(And e, R lft, R rht);
   R visit(Or e, R lft, R rht);
}
```

(b) Functional

Listing 1.7: Visitor Interfaces in Java

the presentation of Oliveira *et al*. [60]. An imperative visitor implements **`void`** methods that communicate using the visitor's local state. In contrast, the functional visitor is parametrized by a type `R`, which represents the return type of the operation to be implemented. In addition to this type parameter, the functional `visit` methods also have more method arguments than the corresponding imperative methods. The functional `visit` methods for composite classes like `Neg` and `And` have extra arguments that represent recursive results of `visit`ing corresponding subcomponents.

The final piece of the visitor puzzle is the implementation of `accept` methods that must be added to our `BExp` classes. The corresponding `accept` methods are shown in Listing 1.8: (a) shows imperative `accept` methods and (b) shows the functional counterpart.[6] `False` and `Or` cases are left out since they are similar to `True` and `And`, respectively. Together, the `accept` methods implement the *traversal* of a `BExp` instance. The visitor's `visit` methods are called at interesting points, effectively implementing *double-dispatch*. Calls to the specific `visit` methods are statically resolved within

[6]Because the parameter types are different, these implementations can coexist in the same class hierarchy, though usually one visitor type is chosen over the other.

```
// Added to BExp
abstract void accept(ImpVis v);

// Added to True
void accept(ImpVis v)
{ v.visit(this); }

// Added to Neg
void accept(ImpVis v){
  inner.accept(v);
  v.visit(this);
}
// Added to And
void accept(ImpVis v){
  left.accept(v);
  right.accept(v);
  v.visit(this);
}
```

(a) Imperative Traversal

```
// Added to BExp
abstract <R> R accept(FuncVis<R> v);

// Added to True
<R> R accept(FuncVis<R> v)
{ return v.visit(this); }

// Added to Neg
<R> R accept(FuncVis<R> v){
  return v.visit(this,
                 inner.accept(v));
}
// Added to And
<R> R accept(FuncVis<R> v){
  return v.visit(this,
                 left.accept(v),
                 right.accept(v));
}
```

(b) Functional Traversal

Listing 1.8: Accept methods for `BExp`s

each class by the type of their first parameter, *e.g.*, `v.visit(this, ...)`, since the type of **`this`** is known.[7]

With both forms of the visitor pattern in place, we can use visitors to implement boolean expression evaluation. Using a visitor, we can now implemented the operation *outside* of our `BExp` class definitions. Listing 1.9(a) and (b) give visitor implementations of `eval` in imperative and functional styles, respectively. Again, we elide the methods for `False` and `Or` since they are almost identical to those for `True` and `And` respectively. Because the imperative visitor, `EvalImp`, must operate via mutation, it keeps a `Stack` of evaluated `Lit` results. After the traversal of sub-expressions is complete, each `accept` method calls `visit`, passing the original `BExp`. Within the corresponding `visit` method the necessary values are `pop`ped from the stack, tested, and the resulting `Lit` is `push`ed back on the stack.

The functional version, `EvalFun`, is implemented similarly. After subexpressions have been traversed, each `accept` method will call `visit`, passing the original `BExp` and the recursive `accept` results. Our functional version is much less complicated, since it uses the implicit call stack to manage recursion. In fact, it is almost identical to our earlier object-oriented version (Listing 1.2), though it is now separated from the class definitions.

Each of the visitors also includes an `eval` entry point that invokes the initial `accept` method with **`this`** instance of the visitor. The imperative visitor must do a little more work. Since its return is stored in the stack, upon completion of the traversal the `eval` method must return the top of the stack (*i.e.*, `pop()`). Each of the operations can be invoked by creating a visitor instance and calling `eval` on a `BExp`:

[7] technically the type of **`this`** cannot be known until runtime, but the Java compiler can at least determine a reasonable static bound.

```
class EvalImp implements ImpVis{
  Stack<Lit> stack = /*...*/;
  void visit(True e)
  { stack.push(e); }
  void visit(Neg e){
    if(stack.pop().isTrue())
      stack.push(new False());
    else
      stack.push(new True());
  }
  void visit(And e){
    if(stack.pop().isTrue() &&
       stack.pop().isTrue())
      stack.push(new True());
    else
      stack.push(new False());
  }
  Lit eval(BExp e){
    e.accept(this);
    return stack.pop();
  }
}
```

(a) Imperative visitor

```
class EvalFun implements FunVis<Lit>{

  Lit visit(True e)
  { return e; }
  Lit visit(Neg e, Lit inn){
    if(inn.isTrue())
      return new False();
    return new True();
  }

  Lit visit(And e, Lit lft, Lit rht){
    if(lft.isTrue() && rht.isTrue())
      return new lft();
    return new False();
  }


  Lit eval(BExp e)
  { return e.accept(this); }
}
```

(b) Functional visitor

Listing 1.9: Visitor-based `eval` implementations

```
new EvalImp().eval(a_bexp);
```

And similarly for `EvalFun`.

1.3.2.2 Visitor Limitations and Extensions

We have organized our methods this way (*i.e.*, `accept` and `visit`) so that the work done writing the `accept` methods (*i.e.*, the traversal) can be reused when we write new operations that follow this pattern. In theory visitors can be used to write all different kinds of operations, but in practice visitor implementations have a number of limitations. Consequently, there have been many extensions to the pattern to deal with implementation challenges.

Necessary Methods The visitor implementation at a minimum requires the addition of `accept` methods to the relevant portions of a class hierarchy. When the programmer does not have control over the classes (*e.g.*, they are provided in binary form) this is much more difficult. When the programmer has control, writing these methods can be tedious and error prone.

Palsberg and Jay [62] introduced a special visitor, called `Walkabout`, that uses Java reflection to mimic double dispatch. Using reflection supports visitor extensions (*e.g.*, new `visit` methods) and allows the traversal to be implemented over unmodified class hierarchies. Grothoff [30] took a similar approach by compiling double-dispatch at runtime with `Runabout`, and the performance was greatly improved, In Aspect-Oriented Programming [40] this notion of making changes without prior knowledge is typically referred

to as *obliviousness*.

In this dissertation we focus on *external* traversals with several different possible implementations. We compare our performance in chapter 5.

Fixed Traversal Implementing a fixed traversal strategy within the class hierarchy makes writing different operations difficult, inefficient, or even impossible. For this reason most programmers simply use visitor pattern as a way to modularize case dispatch, eliminating recursive `accept` calls within the class hierarchy. If, for example, we remove `inner.accept(v)` from `Neg` in the imperative case, or use `inner` in place of `inner.accept(v)` in the functional case, then the visitor can control its own traversal.

When the visitor is in charge of traversal, Buchlovsky and Thielecke [15] refer to them as *external* visitors, as the traversal is external to the datatypes. Oliveira *et al*. [60] demonstrated a functional visitor library[8] that works with both *external* and *internal* `accept` implementations. Lämmel *et al*. [50, 43, 69] introduced composable strategies to define and constrain the order and depth of a traversal. Lieberherr *et al*. [51, 52] use a domain specific language to describe traversal strategies with imperative visitors.

In the next chapter (Section 2.3) we discuss *traversal control* as an alternative to fixed recursion schemes. Our approach is a simplified version of Lieberherr *et al*.'s traversal strategies.

Side-Effects Using **`void`** methods forces programmers to use local mutation to communicate between different parts of a structure. In practice **`void`** methods allow visitors to handle multiple/mutually recursive hierarchies consistently and slightly easier to extend, since a default behavior of "*do nothing*" is always an option. However, using mutation makes imperative visitors difficult to understand, visit-order dependent, difficult to compose, and expensive to parallelize.

Recent visitor work [60, 59] has focused on providing a functional alternative, but places added constraints on visitor extensions and return types. In particular, mutually recursive datatypes require multiple, separate visitor definitions, and shared visitor references, which must be initialized via mutation.

In this dissertation we will use a functional approach that eliminates the need for side-effects. As a result, our traversals will be implicitly (and trivially) parallelizable.

1.3.2.3 Higher-Order Functions

Every seasoned functional programmer has witnessed the value of higher-order functions. While iteration functions like `map` are used extensively for predefined structures like lists, higher-order functions are also very use-

[8] The approach is actually more of an instance of the pattern that makes use of Scala's [58] abstract types.

ful for use with user-defined structures. One of the most common uses of higher-order functions has been to abstract traversals by creating structural recursion operators, typically called *folds* [64, 54].

Returning to our Scheme implementation of BExps, we can use our data definitions as a guide to create a function that folds BExps into a different structure, shown in Listing 1.10.

```
;; fold-bexp: BExp A A (A -> A) (A A -> A) (A A -> A) -> A
(define (fold-bexp e tru fals nott andd orr)
  (cond [(True? e) tru]
        [(False? e) fals]
        [(Neg? e) (nott (fold-bexp (Neg-inner e)
                                   tru fals nott andd orr))]
        [(And? e) (andd (fold-bexp (And-left e)
                                   tru fals nott andd orr)
                        (fold-bexp (And-right e)
                                   tru fals nott andd orr))]
        ...))
```

Listing 1.10: Fold function for BExp structures

The comment preceding fold-bexp describes its signature. The function accepts five arguments, one for each structure definition (*i.e.*, *concrete* variants of BExp). We use A as a place-holder for the return type of our function, since it should be the same throughout. The individual functions passed to fold-bexp match the arity of the corresponding value constructors (*e.g.*, make-Neg), but instead of zero-argument functions we use values for True and False.

For each datatype case we replace the original constructor by calling the corresponding function argument with the results of recursively folding the immediate fields of the structure. We elide the case for Or, since it is almost identical to And. Using a **cond** expression to do case dispatch is analogous to the double-dispatch of our visitor's accept/visit methods.

Once we define the necessary helper functions, we can use fold-bexp to give a more succinct definition of eval, without mentioning any structural recursion. Our functions are shown in Listing 1.11. For each compound constructor (*e.g.*, Not or And) we create a function that will be called on the recursive results to produce a single Lit.

Our implementation of eval has successfully been reduced to a concise, two line function. Similar to our visitor solution, the traversal of our data structure has now been neatly separated from our operation specific implementation, and we can reuse our fold to write other functions that match this pattern of structural recursion.

1.3.2.4 Limitations and Extensions

As is common with the visitor pattern, the fold functions take many different forms, but in practice implementations suffer from a number of limitations.

```
;; eval: BExp -> Lit
(define (eval e)
  (fold-bexp e (make-True) (make-False) not-lit
             and-lit or-lit))

;; not-lit: Lit -> Lit
(define (not-lit a)
  (if (True? a) (make-False) (make-True)))
;; and-lit: Lit * Lit -> Lit
(define (and-lit a b)
  (if (and (True? a) (True? b)) a (make-False)))
;; or-lit: Lit * Lit  -> Lit
(define (or-lit a b) ...)
```

Listing 1.11: Fold-based `eval` implementation

Structures and Fixed Recursion How the fold is implemented is important. Deep folds as we have implemented here in `fold-bexp` are more expressive than shallow folds [28], but run into the same problems as the visitor implementations since the recursion scheme is fixed.

Generalized folds [54, 64] and other forms of generic programming [12, 37, 9, 33, 53] offer alternatives that eliminate the tedium of writing folds for data structures, but again offer fixed traversals. Abstracting the recursion scheme of folds has lead to several variants of strategic and combinator approaches [44, 46, 43]

One Constructor, One Function Each constructor is handled by exactly one argument to `fold-bexp`. For large structures the number and order of these parameters contributes to the program's boilerplate. Sheard [64] places functions into nested tuples, and Lämmel [45] collects related functions into a record. These are, however, only partial solutions, since the structure holding the functions must then be constructed and deconstructed.]

Having one function per constructor also limits programmers' opportunities to abstract multiple, similar cases into a single function, or to override a more general function with a more specific case.

In the next chapter we discuss our use of function-classes over traversals, which support function extension, case abstraction, and specialization.

Return Types The signature of our fold function, `fold-bexp`, suggests that the types of the arguments (*e.g.*, the return type of the functions), be consistent, *e.g.*, `A`. The functions passed as arguments must accept and return values of this type. In a statically typed language like ML [57] or Haskell [38] this forces programmers to write several separate functions. In many cases a single return type for all parts of a structure is not flexible enough, *e.g.*, `map` where the function passed transforms the elements of the list, but the list structure remains intact.

Implementations usually treat functions like `map` (so-called *homomorphisms* [54]) as a special case, or provide implementations for a small num-

ber of situations [47, 44].

In this dissertation, chapter 3 in particular, we show that it is possible to relax this constraint without compromising safety.

CHAPTER 2

DemeterF

The DemeterF system consists of a number of tools and libraries. In this chapter we discuss our class generator, traversal library, and generic programming tools. We begin with an overview of the structural and behavioral aspects of DemeterF-based programs, followed by an example driven introduction to writing traversal-based functions. We conclude with a discussion of the generic and generative programming features of DemeterF.

2.1 Structures and Classes

Writing traversal-based functions begins with a description the data structures involved and the relations between different datatypes, *i.e.*, fields or *has-a* relationships, and subtyping or *is-a* relationships. DemeterF allows programmers to separate descriptions of the structural elements of a hierarchy from their methods and behavior by merging definitions into generated Java classes.

As input, the DemeterF class generator accepts a *class dictionary* (CD) file [51] and a *behavior* (BEH) file. A CD describes the structures of a class hierarchy and the BEH file provides extra, class-specific definitions and methods to be placed in the body of generated source files. The format of DemeterF CDs includes a number of improvements over Lieberherr's original design. Notably, we support generic definitions with bounded parametric polymorphism, and the inclusion of other CDs.

2.1.1 Class Dictionaries

Figure 2.1 describes a simplified version of DemeterF CD syntax using BNF. For the purposes of this dissertation we have left out syntactic features related to datatype generic programming, though relevant points will be discussed in Section 2.7. In our BNF notation, concrete syntax is enclosed in double-quotes, *e.g.*, "`include`". Optional syntax is placed in square brackets, [ ], and zero or more repetitions is denoted by a postfix Kleene star. The non-terminal IDENT represents valid Java identifiers and CHAR stands for a

```
  CDFile ::= Include* TypeDef*
 Include ::= "include" String ";"
  String ::= "" Char* ""

 TypeDef ::= ["extern"] ( Class | Intfc )
  Syntax ::= String | Annot
  Annot ::= "*s" | "*l"

   Class ::= Decl "=" [ Use ("|" Use)* ] (Field | Syntax)* [ Impl ] "."
   Intfc ::= "interface" Decl "=" [ Use ("|" Use)* ] "."

   Field ::= "<" Ident ">" Use
    Decl ::= Ident ["(" Ident ("," Ident)* ")"]
     Use ::= Ident ["(" Use ("," Use)* ")"]
```

Figure 2.1: Simplified Class Dictionary (CD) syntax

(possibly escaped) character literal. A CDFile begins with a possibly empty sequence of Include statements. The semantics of CD file inclusion is simply concatenation of the TypeDef sequences from each file. The body of a CDFile is sequence of class and/or interface definitions. A Class is defined by declaring the class' name and type parameters. The right-hand side of a Class definition contains a possibly empty list of subclasses followed by a possibly empty list of Field and/or Syntax definitions, terminated by a period, ".". An Intfc is prefixed with "`interface`" and is defined as a possibly empty list of implementing classes. Each TypeDef can optionally be declared with the prefix "`extern`", meaning that the class has already been defined externally, *e.g.*, in another library, and should not be generated.

A class with a non-empty list of subclasses is termed *abstract*, and, as in Java, cannot be directly instantiated. A class definition that only contains fields and/or syntax (*i.e.*, no subtypes) is termed *concrete* and can be used to create structures at runtime. Fields can be interspersed with Syntax (quoted strings or printing annotations) that guide generated parsers and printers. Strings define the concrete syntax of both the input and output language for a group of class definitions. The literals **`*s`** and **`*l`** add spaces and newlines, respectively, to generated `print` methods that are added to the generated Java files if requested.

Listing 2.1 shows a CD representing integer binary search tree (BST) structures. The hierarchy consists of an abstract class, `IntBST`, with two concrete subclasses, `IntNode` and `IntLeaf`. `IntNode` represents BST interior tree nodes with an integer field `data` and `left` and `right` subtrees. `IntLeaf` represents a terminal/empty tree.

From this structural description DemeterF creates three main Java files, one for each class definition. If requested, the system will also generate

```
IntBST  = IntNode | IntLeaf.
IntNode = "(" <data> int *s <left> IntBST
                         *s <right> IntBST ")".
IntLeaf = "*".
```

Listing 2.1: CD file describing functional integer BSTs

a printer and parser, using the the syntax inferred from the CD.[1] All generated fields are by default declared **`final`** and class constructors follow a functional initialization pattern, similar to that enforced by Featherweight Java [36]. Each class' constructor accepts a value for each field as arguments and initializes all fields immediately.

Listing 2.2 shows the definition of three instances of `IntBST`. The first is

```
IntBST t1 = new IntLeaf(),
       t2 = new IntNode(2, new IntNode(1, t1, t1),
                           new IntNode(3, t1, t1)),
       t3 = IntBST.parse("(2 (1 * *) (3 * *))");
```

Listing 2.2: `IntBST` uses: constructors and parsing

an empty `IntLeaf` tree, which is used to construct a tree with a root node of 2, and left and right children of 1 and 3 respectively. The second and third `IntBST`s, `t2` and `t3`, represent the same BST: `t2` is created using constructor calls, and `t3` using a generated (**`static`**) *`parse`* method in `IntBST`. In addition to `parse` and `print` methods, DemeterF is also able to generate field getters (*e.g.*, **`int`** *`getData`*`()`, functional field updaters (*e.g.*, `IntNode` *`updateData`*`(`**`int`**` d)`), and a number of other useful methods if requested.

2.1.2 Behavior Files

Behavior (BEH) definitions allow methods and other Java syntax (*e.g.*, comments or **`static`** fields) to be placed in generated classes while remaining separate from their structural definitions (*i.e.*, the CD). Figure 2.2 shows our simple behavior syntax in BNF. Similar to CD files, we allow other BEH files

BEHFILE ::= INCLUDE* BEH*
BEH ::= IDENT "{{" TEXT "}}"

Figure 2.2: BEH File Syntax

to be included. The terminal TEXT represents any string that does not include double braces, "}}". The TEXT within the double braces is bound to

[1] Other so-called *datatype-generic* programming functions can also be defined. The specifics will be discussed in Section 2.7.

the preceding IDENT. Once different TEXTs with the same bindings has been merged, the code is injected into the body of the CLASS or INTFC definition when files are generated. An example behavior file for our IntBST classes is given in Listing 2.3. Each of the BEH definitions defines a method. To-

```
IntBST{{
   abstract IntBST insert(int d);
}}
IntNode{{
   IntBST insert(int d){
      if(d <= data)
         return new IntNode(data, left.insert(d), right);
      return new IntNode(data, left, right.insert(d));
   }
}}
IntLeaf{{
   IntBST insert(int d){ return new IntNode(d, this, this); }
}}
```

Listing 2.3: BEH definitions for integer BST insertion

gether the methods implement the insertion of an **int** into an IntBST. The **abstract** class IntBST declares an **abstract** method, insert, that its subclasses must implement. For IntNode we use data to decide between recursive insertion into the left or right subtree, and return a new IntNode with corresponding subtree updated. For IntLeaf we construct a new IntNode with d as its data, and **this** IntLeaf as its left and right subtrees.[2]

When Java files are generated for our BST classes, DemeterF combines the CD and BEH definitions into Java files. Listing 2.4 shows the resulting class definition for IntNode. The generated class **extends** IntBST, which is the result of it appearing on the right-hand side of the IntBST definition. IntNode also has **protected final** fields that match its CD definition. The generated constructor accepts and initializes the instance's three fields, and a parse method is added to parse an instance from a String using the language defined by the CD. Our insert method from the BEH definitions is placed verbatim after the generated constructor and methods.

2.1.3 Parametric Polymorphism

Type parametrization is now common place, even in object-oriented communities. While ML and Haskell allow *implicit* parametric polymorphism, Java and C# support explicit parametric polymorphism in both classes and methods using so-called *generic* declarations.[3] DemeterF allows pro-

[2]Note that this version of insert is functional, *i.e.*, mutation free. Side-effecting versions are possible with DemeterF, but the author hopes to discourage the practice of mutable data structures, hence the **final** fields by default.

[3]Java and C# do support a limited form of inference that can recover type variable annotations.

```
// IntNode.java
public class IntNode extends IntBST{
   protected final int data;
   protected final IntBST left;
   protected final IntBST left;

   /** Construct a(n) IntNode Instance */
   public IntNode(int data, IntBST left, IntBST right){
      this.data = data;  this.left = left; this.right = right;
   }

   /** Parse an instance of IntNode from the given String */
   public static IntNode parse(String inpt){ /*...*/ }

   /*... Other generated methods ...*/

   IntBST insert(int d){
      if(d <= data)
         return new IntNode(data, left.insert(d), right);
      return new IntNode(data, left, right.insert(d));
   }
}
```

Listing 2.4: Combined CD and BEH in `IntNode.java`

grammers to define parametrized classes and interfaces that correspond to definitions of generic classes in Java and C#.

CD definitions can be parametrized by placing type parameters in parentheses, separated by commas (DEF and USE in Figure 2.1). Explicit bounds on type parameters are supported by generating class definitions that make use of Java's **`extends`** syntax and C#'s **`where`** clauses. Our CD notation uses colons to separate a type variable from its bound.

For example, Listing 2.5 shows a CD that defines classes for a generic BST implementation. We provide an **`extern`** definition for Java's parametrized

```
extern interface Comparable(X) = .

BST(D:Comparable(D)) = Node(D) | Leaf(D).
Node(D:Comparable(D)) = "(" <data> D *s <left> BST(D)
                                   *s <right> BST(D) ")".
Leaf(D:Comparable(D)) = "*".
```

Listing 2.5: CD definitions for parametrized `BST`s.

`Comparable` interface, which is used as a bound for our type parameter, `D`, representing the `data` to be stored in the tree. The type of `data` stored, `D`, must implement `Comparable(D)`, so that nodes in the tree can be ordered to maintain a typical BST invariant.

Listing 2.6 shows behavior definitions that implement an `insert` method for our generic `BST`s. The `insert` methods are similar to what we implemented for `IntBST`s, though each now works on `data` of the type parameter `D` and returns a result of type `BST<D>`. Our generic BST class can be instanti-

```
BST{{
   abstract BST<D> insert(D d);
}}
Node{{
   BST<D> insert(D d){
      if(d.compareTo(data) <= 0)
         return new Node<D>(data, left.insert(d), right);
      return new Node<D>(data, left, right.insert(d));
   }
}}
Leaf{{
   BST<D> insert(D d){
      return new Node<D>(d, this, this);
   }
}}
```

Listing 2.6: BEH definitions for generic BST insertion

ated in another (or the same) CD to support parsing and printing, as shown in Listing 2.7(a), with Java code using the generated classes shown in Listing 2.7(b). We create two wrapper classes: one that instantiates BST to store

```
DoubleBST = <bst> BST(Double).
CharBST = <bst> BST(Character).
```

(a) CD use of generic BST

```
// BST of Doubles...
BST<Double> dbst = DoubleBST.parse("(1.2 (1.1 * *) (1.3 * *))")
                            .bst.insert(1.4);
// BST of Characters...
BST<Character> cbst = CharBST.parse("('B' ('A' * *) ('C' * *))")
                             .bst.insert('D');
```

(b) Use within Java code

Listing 2.7: Generic BST instantiation and uses

Double values, and the other which stores Characters.[4] Our parametrization is valid, since Double and Character both **implement** Comparable for their respective types, matching our type parameter bounds. Once the wrapper/instantiation has been *parse*d we can access its field and insert a new element. In both cases the new element is placed to the right of the root's right subtree. Note that the wrapper classes are only necessary to support parsing and printing, since they alert DemeterF that a particular instantiation of the classes is required. Generated parametrized classes are otherwise the same as their Java equivalents.

[4]Java allows only *reference* types to be used as type parameters, so we use the *boxed* equivalents. In C# this is not necessary, since the language allows *value* types as parameters.

2.1.4 Running Example

For the rest of this chapter, we return to the example data structures defined in chapter 1 (Listing 1.1). Listing 2.8 shows a CD that defines the same classes using a DemeterF CD.

```
BExp = Lit | Neg | And | Or.

Lit = True | False.
True = "True".
False = "False".

Neg = "!" <inner> BExp.
And = "(&&" *s <left> BExp *s <right> BExp ")".
Or  = "(||" *s <left> BExp *s <right> BExp ")".
```

Listing 2.8: CD definitions for boolean expressions

The first line corresponds to our **abstract** class `BExp`, with four subclasses: `Lit`, `Neg`, `And`, and `Or`. `Lit` is also **abstract**, with concrete subclasses of `True` and `False`. `True` and `False` are defined only as syntax.

Other definitions are concrete classes with *recursive* fields. `Neg` has a single `inner` expression, while `And` and `Or` are binary expressions, each with two recursive fields. For completeness we give `And` and `Or` prefix operators as concrete syntax, since infix operators would make the resulting grammar non-LL(k).[5]

For the rest of this chapter we use these definitions to demonstrate the various features of writing traversal-based functions using DemeterF. When necessary we will extend our structures to illuminate different aspects of our system.

2.2 Functions and Traversals

CDs are useful for describing the structure and syntax of data, but what we eventually want to do is write functions over instances of our structures that return meaningful results. In order to write traversal-based functions, DemeterF provides classes that represent *function-classes* and traversals, that together are used to implement functions over a data structure.

2.2.1 Function-Classes

A DemeterF function-class represents a *set* of functions using Java methods with the special name *`combine`*. DemeterF provides a base function-class,

[5] DemeterF currently uses JavaCC [3] to generate parsers. JavaCC can do more complex look-ahead, but DemeterF generates LL(k) parser descriptions, which fail to work correctly in the presence of left recursion. However, using the generated classes without the generated parser works as expected.

FC, that represents the empty set of functions. To create a new function-class, programmers can extend FC by adding specific *combine* methods for a given data structure. The *combine* methods of a function-class instance (or *function-object*) are interpreted as *fold* functions over an adaptive, generic traversal.

As a first example, Listing 2.9 defines a simple function-class with the intent of converting a BExp into a String with the help of a traversal. Our

```
// Convert a BExp to a String...
class ToString extends FC{
   String combine(True t){ return "True"; }
   String combine(False f){ return "False"; }
   String combine(Neg n, String i){ return "!"+i; }

   String combine(And a, String l, String r)
   { return "(&& "+l+" "+r+")"; }
   String combine(Or o, String l, String r)
   { return "(|| "+l+" "+r+")"; }
}
```

Listing 2.9: ToString function-class

class, appropriately named ToString, **extends** the base function-class FC. It adds a *combine* method for each concrete case of our BExp data structures. In this case the *combine* methods can be identified by the type of their first argument. In each of the methods we return a String that corresponds to the concrete syntax from our original CD (Listing 2.8).

2.2.2 DemeterF Traversals

In order to turn an instance of a function-class (*i.e.*, a function-object) into a function, we apply its *combine* methods over the traversal of a data structure. To do this, DemeterF provides a class, Traversal, that takes an instance of a function-class. A typical Traversal usage is shown in Listing 2.10. The method toString is added to the ToString class. The method

```
// Added to ToString
String toString(BExp e){
   return new Traversal(this).traverse(e);
}
```

Listing 2.10: Traversal invocation for ToString

accepts a BExp instance and traverses it in order to convert it into a String. A new Traversal is constructed by passing **this** function-object, and the BExp is traversed. The use of **this** references the current instance of ToString (*i.e.*, the function-object), whose *combine* methods are called by the Traversal to fold together the BExp instance.

When called, the *traverse* method proceeds with a depth-first walk of the given object, in this case a BExp. After recursively traversing the fields of the current node, the Traversal selects a *combine* method from the given function-object that best matches: (1) the type of the current node, and (2) the result types of recursively traversing each of the node's fields. This is termed *multiple dispatch*, since all argument types determine the selected method. Once selected, the *combine* is then applied to the original node (as its first argument) and the traversal results of its fields.

Getting back to ToString, instances of True or False have no fields, so selecting a *combine* method is simple. The traversal selects the first or second method in ToString based on the type of the object itself. When applied to a Neg instance, *traverse* first recursively processes the object's inner field. If the result of the traversal is a String, then the third method is selected and applied. Similarly for And and Or, with both fields (left and right) being traversed before a matching method is selected. Any case for which the function-object does not have a matching *combine* method, *e.g.*, (Neg, int), results in a runtime/dispatch exception.

As with our visitor solutions (Section 1.3.2.1), our function-class and traversal can be used by creating a new function-object and calling our toString with a BExp:

```
new ToString().toString(a_bexp)
```

If the function was needed more than once, we could name a reference to the ToString instance for use with multiple calls. This style of definition could be considered *object-oriented*, since the toString method is only associated with an instance of ToString. A more global/functional implementation is also possible by making toString a **static** method that constructs a Traversal with a **new** ToString instance.

2.2.3 Case Abstract and Specialization

As a second example of a traversal-based function we implement *strict* BExp evaluation, similar to our fold-based Scheme example (Listing 1.11 in Section 1.3.2.3). Listing 2.11 shows a complete function-class that implements a slightly inefficient version of BExp evaluation. Our function-class, StrictEval, has a number of interesting *combine* methods, each of which matches a specific case of evaluation.

The first method matches both True and False instances with their supertype, Lit, returning the literal unchanged. For Neg we match possible cases with separate *combine* methods, returning the negation of the recursive inner traversal result. The first two *combine* methods for And and Or match the important situations where the recursive results are both True or both False, in which case the left result, l, can be returned. The final two cases match default cases for And and Or, where we can return False and True respectively. Again we implement a wrapper method, eval, that con-

```
// Evaluate a BExp
class StrictEval extends FC{
   Lit combine(Lit l){ return l; }
   Lit combine(Neg n, True t){ return new False(); }
   Lit combine(Neg n, False f){ return new True(); }

   Lit combine(And a, True l, True r){ return l; }
   Lit combine(Or  a, False l, False r){ return l; }
   Lit combine(And a, Lit l, Lit r){ return new False(); }
   Lit combine(Or  a, Lit l, Lit r){ return new True(); }

   Lit eval(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Listing 2.11: DemeterF-based strict boolean expression evaluation

structs a `Traversal` with **`this`** function-object and calls *`traverse`* on the given `BExp`.

The `StrinctEval` example demonstrates two novel features of using multiple dispatch over data structure traversal. The first is abstraction: our *`combine`* selection allows us to *abstract* multiple common cases into a single method. This occurs with the *`combine`* for `Lit` where we avoid mentioning separate cases for `True` and `False`, and the second/default methods for `And` and `Or`, which each handle 3 cases. The dual of abstraction is specialization: we can write a method signature that overrides a more general case with a specific result. This occurs in the first two *`combine`* methods for `And` and `Or`, where the specialized signature, *e.g.*, `(And True True)`, overrides the abstracted case. In either case the traversal's multiple dispatch selects the *`combine`* method with the most appropriate signature. These two features help to support extensible function-classes, making the function-objects over traversal more useful.

2.3 Traversal Control

The separation of functions into function-classes and `Traversal` allows us to easily augment the traversal with additional features. Continuing with our `BExp` evaluation example, we originally used visitors (Listing 1.9), higher-order functions (Listing 1.11), and DemeterF traversal (Listing 2.11) to implement evaluation. Although these forms of traversal eliminate the boilerplate of traversal, they were not capable of *short-cutting* the traversal. In order to implement the well-known notion of non-strict boolean evaluation we will use DemeterF traversal *control* with our function-class.

DemeterF supports a version of traversal control that is a simplification of that found in Adaptive Programming *strategies* [51, 52]. The `Traversal` class provides a second constructor that takes a two arguments, the first is a function-object and the second is of type `Control`. The DemeterF class

`Control` has creator methods that allow a programmer to describe specific fields to be *bypassed* (or skipped over) during traversal, effectively guiding the `Traversal` through a data structure. To make the evaluation of `And` and `Or` non-strict, we specify that their `right` field should be *`bypass`*ed.

Listing 2.12 shows a function-class that correctly implements short-cutting `BExp` evaluation. Our function class, `Eval`, is quite similar to our previous

```
// Evaluate a BExp
class Eval extends FC{
   Lit combine(Lit l){ return l; }
   Lit combine(Neg n, True t){ return new False(); }
   Lit combine(Neg n, False f){ return new True(); }

   // The "right" field will not be traversed
   Lit combine(And a, False l, BExp r){ return l; }
   Lit combine(Or  a, True  l, BExp r){ return l; }
   Lit combine(And a, True  l, BExp r){ return eval(r); }
   Lit combine(Or  a, False l, BExp r){ return eval(r); }

   Lit eval(BExp e){
      return new Traversal(this,
                    Control.bypass("And.right Or.right"))
             .traverse(e);
   }
}
```

Listing 2.12: DemeterF-based non-strict boolean expression evaluation

example, `StrictEval` (Listing 2.11). For `Lit` and `Neg` instances, the method selection is the same as `StrictEval`.[6]

Before describing the rest of the function set, it is important to take a closer look at the `eval` method. We construct our `Traversal` by passing **`this`** function-object and a `Control` object created using *`bypass`*. The string given to *`bypass`* represents the fields to be skipped, in this case `And.right` and `Or.right`.[7] During the execution of `traverse`, when the current node is an instance of `And` or `Or` our `Control` tells the traversal to *skip* its `right` field. After the traversal of the `left` field is complete, a method is selected based on the type of the current node (*i.e.*, `And` or `Or`), the result type of the recursive traversal of the `left` field, and the type of the unchanged `right` field.

Our plan to bypass the `right` field is reflected in the type of the third argument of our last four *`combine`* methods. We use the type `BExp` (instead of `True` or `False`), which matches the field's original type. In the first two cases we can immediately return the result of the `left` traversal, `True` or `False` respectively. In the final two cases we make a recursive call to evalu-

[6] In fact, we could have just extended `StrictEval`, but we save that discussion for later (Section 2.5).

[7] If Java had macros we could better integrate *Control/`bypass`* into the language. Our implementation using Scheme provides a much more user friendly integration [18] without exposing implementation details.

ate the `right` side of the expression. Since the right side of the expression is only traversed when necessary, we achieve our short-cutting/non-strict evaluation strategy.

2.3.1 Efficient Recursion

When a field is *`bypass`*ed during traversal it is common to hand-code a recursive call after checking a condition. In those cases it is inefficient to reconstruct an identical traversal, *e.g.*, in our `eval` method, for each recursive call. We can instead create and store the `Traversal` instance in a local variable when the function object is initialized. This initialization effectively "*ties*" the recursive "*knot*", so the traversal can be references for hand-made recursive calls. Listing 2.13 shows this caching strategy implemented for our `Eval` function-class.

```
// Replacement "eval" for Listing 2.12
Lit eval(BExp e){ return trav.traverse(e); }

// Cached Traversal/Control and constructor
Traversal trav;
Eval(){
   trav = new Traversal(this,
                Control.bypass("And.right Or.right"));
}
```

Listing 2.13: Cached `Traversal` for efficient recursion

Depending on the size of the `BExp` instance that is traversed and the number of recursive calls required, this caching can save a significant amount of space, time, and more importantly, object allocations.[8]

2.4 Traversal Contexts

There are times when writing purely compositional functions will not suffice. In cases where information about the ancestors of a sub-structure is important to a method's result, programmers typically add an argument to the method definition. This argument is then passed to recursive invocations and updated when appropriate. DemeterF supports this style of traversal-based function using a notion of *traversal contexts*.

2.4.1 Update Methods

In addition to *`combine`* methods, a function-class can define *`update`* methods. While *`combine`* methods are akin to fold functions (*i.e.*, bottom-up),

[8]Parallel execution seems to be more dependent on allocations due to Java's shared heap and garbage collection.

update methods are responsible for updating the traversal context a interesting points,[9] *top-down*, similar to inherited attributes in Attribute Grammars [41]. The context is available to each *combine* method as its last argument. Methods can, however, ignore the context (or other later arguments) simply by declaring a shorter signature.

Methods that *update* the traversal context can accept up to three arguments that represent (1) the current node of the structure, (2) the next field to be traversed, and (3) the current node's context. The field to be traversed is encoded as an instance of a *field-class*. For each field of a CD definition, *e.g.*, `left` from the class `And`, DemeterF generates a **`static`** inner class whose instances represent the pending traversal of the field. Each field-type is defined as a subtype of the DemeterF class `Fields.any` (another inner class). Our representation has the added benefit of making field-classes in *update* methods look like field accesses, *e.g.*, `And.left` is the field-class of the `left` field of `And` instances.

To demonstrate traversal contexts with another `BExp` example, we extend our `BExp` structures with variable expressions and implement a traversal-based function that transforms a `BExp` into *negation normal* form. The modified CD definitions are shown in Listing 2.14 along with classes to represent the `Sign` of nested negations. For brevity we elide our unchanged structures.

```
// Add Var to the BExp definition
BExp = Lit | Neg | And | Or | Var.
Var = <id> ident.

// Sign of nested negations
Sign = Even | Odd.
Even = .
Odd = .
```

Listing 2.14: Adding `Var` and `Sign` contexts

The class `Var` is added as a subtype of `BExp`. The new concrete class contains an `ident`, a DemeterF library class that represents identifiers.

Our strategy for implementing negation normalization is to keep track of the number of nested outer `Neg` expressions during the traversal as our context. We represent the nesting depth by the abstract class `Sign`, which is either positive, `Even`, or negative, `Odd`. Before traversal proceeds into the `inner` field of a `Neg` instance we use an *update* method to flip the `Sign` of the context for the `inner` subtraversal. When variables or literals are reached we return an adjusted instance based on the `Sign` of the context. For `And` and `Or` we follow the usual rules for `And` and `Or` under negation when the context is `Odd`.

Listing 2.15 shows the complete implementation of negation normalization as a function-class. The class is best explained case by case. The *update*

[9] By "update" we mean *functional update*, where mutation is avoided by constructing a new, updated instance.

```
class NegNormal extends FC{
   // Flip the Sign when entering a Neg
   Sign update(Neg n, Neg.inner f, Even s){ return new Odd(); }
   Sign update(Neg n, Neg.inner f, Odd s){ return new Even(); }

   // Literals and Vars
   BExp combine(Lit l, Even s){ return l; }
   BExp combine(True f, Odd s){ return new False(); }
   BExp combine(False f, Odd s){ return new True(); }
   BExp combine(Var v, ident id, Even s){ return v; }
   BExp combine(Var v, ident id, Odd s){ return new Neg(v); }

   BExp combine(Neg n, BExp e){ return e; }

   // Follow De Morgan Laws...
   BExp combine(And a, BExp l, BExp r, Even s){ return new And(l,r); }
   BExp combine(And a, BExp l, BExp r, Odd s){ return new Or(l,r); }
   BExp combine(Or o, BExp l, BExp r, Even s){ return new Or(l,r); }
   BExp combine(Or o, BExp l, BExp r, Odd s){ return new And(l,r); }

   // Main Entry...
   BExp normalize(BExp e)
   { return new Traversal(this).traverse(e, new Even()); }
}
```

Listing 2.15: BExp negation normalization

methods will be called when the current node is an instance of Neg, before traversing into its inner field. This is encoded by the first argument, Neg, and the second argument type of Neg.inner. For each of the context types we return the opposite Sign: Even for Odd and vice versa. For other cases the traversal automatically propagates the context unchanged.

As for the *combine* methods, the first matches after traversing a Lit instance within an Even context and returns the original literal. The next two cases match True and False instances within an Odd context, returning their negation. After normalization, only variables are negated, so the *combine* for Neg accepts just two arguments, ignoring its context, and returns the recursively normalized inner BExp.

The cases for Var return the original variable within an Even context, and its negation within an Odd context.[10] The final four *combine* methods rebuild or convert And and Or instances under Even or Odd contexts respectively. The cases follow De Morgan conversion rules for conjunction/disjunction, *e.g.*, $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$, with the traversal having already propagated negations and recursively normalized the left and right fields.

The normalize method completes our implementation by creating a new Traversal and calling *traverse*. We pass two arguments to traverse: the given BExp and a root context. Since we begin with no outer Neg, our initial context is Even.

[10]Note that we reference the original Var rather than building a **new** one, though in most cases the two will be indistinguishable.

2.5 Extensible Functions

The separation of function-classes and traversal allows us to independently extend/override *combine* and *update* methods. DemeterF supports such extension using Java inheritance. As with traditional inheritance, duplicate signatures will be overridden, and other methods will be overloaded, with preferences determined from the methods' argument types by multiple dispatch.

A typical use of traversals where function-class extension is convenient is when performing *functional updates* over a particular structure, similar to `map` over lists.[11] Listing 2.16 shows a class named `Copy`, which is used as a foundation for such a transformation over `BExp`s. Each *combine* method

```
class Copy extends FC{
   Lit combine(Lit l){ return l; }
   Neg combine(Neg e, BExp in){ return new Neg(in); }
   And combine(And e, BExp l, BExp r){ return new And(l,r); }
   Or  combine(Or  e, BExp l, BExp r){ return new Or(l,r); }
   Var combine(Var e, ident id){ return new Var(id); }
}
```

Listing 2.16: `Copy`: functional updates for `BExp`s

rebuilds our `BExp` structures during traversal by calling the individual constructors on recursive results.

As an example, we can extend `Copy` with specialized *combine* methods that will simplify constant (non-variable) expressions to `True` or `False` literals. Listing 2.17 shows our extended function-class, `Simplify`, that implements such a transformation. Our functions override `Copy` with specific cases where the current `BExp` can be simplified based on recursive results. A `Neg` instance can be simplified when its recursive `inner` result is a `Lit` by returning its negation, or when its recursive result is a `Neg` by returning the `inner` simplified `BExp`. Instances of `And` and `Or` have a number of cases that can be simplified when at least one of the recursive results is a `Lit`. The first case for each uses a shorter signature, ignoring the recursive result from its `right` field, since it is not needed. In other cases, the original `And` or `Or` can be replaced by the simplified results from its `left` or `right` field.

In cases where the specific *combine* methods from `Simplify` do not match, the methods from `Copy` are used to rebuild the structure. The `Traversal` gives us the added benefit of implicit recursion, so our transformation applies to the entire data structure. This kind of transformation is so common that DemeterF provides a function-class, named `TP` for *type-preserving* [44, 47], that generically implements `Copy` for all structures. We will discuss `TP` and other generic function-classes in Section 2.7.

[11] It is not exactly the same, since list `map` is shallow and our traversals are deep.

```
class Simplify extends Copy{
   Lit  combine(Neg n, True t){ return new False(); }
   Lit  combine(Neg n, False f){ return new True(); }
   BExp combine(Neg n, Neg e){ return e.inner; }

   Lit  combine(And a, False l){ return l; }
   Lit  combine(And a, BExp l, False r){ return r; }
   BExp combine(And a, True l,  BExp r){ return r; }
   BExp combine(And a, BExp l,  True r){ return l; }

   Lit  combine(Or o, True l){ return l; }
   Lit  combine(Or o, BExp l,  True r){ return r; }
   BExp combine(Or o, False l, BExp r){ return r; }
   BExp combine(Or o, BExp l, False r){ return l; }

   BExp simplify(BExp e)
   { return new Traversal(this).traverse(e); }
}
```

Listing 2.17: BExp simplification, using Copy

2.6 Mutual Recursion

Previously, our example data structures have only been *self recursive*, where recursive occurrences within concrete subclasses of BExp are all of type BExp. Mutually recursive types can make processing instances more complicated, particularly when visitors [60] or folds [64] are used to implement operations. DemeterF traversals, however, handle mutual recursion just like self recursion. Since the Traversal selects the most specific matching *combine* method from the given function-object, the grouping of methods or types to which they apply is handled by our multiple dispatch.

As an example, we can extend our BExp structures to include a class that represents variable binding. Listing 2.18 shows our new structures. We add

```
// Add Let to BExp definition
BExp = Lit | Neg | And | Or | Var | Let.

// Variable bindings
Let  = "let" *s <bind> Bind *s
       "in" *s <body> BExp.
Bind = <id> ident *s "=" *s <e> BExp.
```

Listing 2.18: Mutually recursive structures

a new BExp subclass, Let, that contains a Bind and a body BExp. A binding is represented with an ident and a BExp. The types BExp and Bind are considered mutually recursive since a Let is a BExp and has a Bind, which in turn has a BExp.

We can reuse our previous example, Simplify, to handle our new structures by adding Let and Bind cases to our Copy function-class, and extending

Simplify.[12] Listing 2.19 shows the function-class extensions.

```
// Extend copy for Let and Bind
class Copy extends FC{
   /* ... Others from Listing 2.16... */

   Let combine(Let l, Bind b, BExp e){ return new Let(b,e); }
   Bind combine(Bind b, ident id, BExp e){ return new Bind(id,e); }
}

// Extend Simplify for Let
class SimplifyWLet extends Simplify{
   BExp combine(Let l, Bind b, Lit e){ return e; }
}
```

Listing 2.19: Copy additions and Simplify extension for Let

Our new function-class, SimplifyWLet, adds a *combine* method for the new structure that simplifies a Let when its body can be simplified to a Lit, since the binding is unnecessary given our pure interpretation of BExps. Because each case is handled separately, the presence of mutual recursion does not affect our traversal: *combine* methods are still applied as usual. Our previous simplify method does not need to be redefined, it works as expected when called on an instance of our new class:

```
new SimplifyWLet().simplify(a_bexp_wlet)
```

And, of course, the function-class still operates on instances without our new Let and Bind structures.

2.7 Generic Programming

We have shown several examples of traversal-based functions over data structures. While we developed them for our particular BExp data structures, many of them are written with a degree of genericity. Because the traversal adapts the *combine* methods to a data structure, the function-class itself can, in many cases, avoid mentioning certain parts of the data structures. For instance, the ToString function-class from Listing 2.9 relies on three pieces of information: the names of the concrete classes mentioned, the *number* of parameters/fields, and the return types of their respective subtraversals. In this section we take a closer look at the generic aspects of traversal-based programming with DemeterF.

2.7.1 Generic Function-Classes

The spectrum of generic functions can (usually) be divided into two different kinds: type-unifying and type-preserving [44, 47].

[12] Multiple inheritance would be very useful in this case to extend both Copy and Simplify simultaneously.

2.7.1.1 Type-Unifying Functions

Type-unifying (TU) functions are those that sum a specific property over a data-structure. This category includes counting or collecting instances of a certain type within a larger data structure, or calculating the size of a structure. DemeterF supports the writing of generic TU functions with a parametrized function-class, `TU<X>`, that sums a property of type `X` over a structure. The function declares two abstract methods that the client must implement: a default *`combine`* method that takes no arguments, and a `fold` method that folds together two results of type `X`.

For example, Listing 2.20 shows a function-class, `UsedVars` that collects the used variable names within a `BExp` instance. Our function-class **`extends`**

```
class UsedVars extends TU<Set<ident>>{
   Set<ident> combine()
   { return Set.<ident>create(); }
   Set<ident> fold(Set<ident> a, Set<ident> b)
   { return a.union(b); }

   Set<ident> combine(Var v){ return Set.create(v.id); }
}
```

Listing 2.20: Collect used variables in a `BExp` using `TU`

`TU<Set<ident>>`, in order to collect the `Set` of names, `idents`, of used variables within a `BExp`.[13] We provide an implementation of a default *`combine`* method that returns the empty `Set`, and a `fold` method that returns the `union` of two `Sets`. The final *`combine`* method creates a singleton set from the `id` within a `Var` instance, *i.e.*, a *used* variable.

When an instance of `UsedVars` used over a traversal, the default *`combine`* is called whenever a leaf of the structure is reached. When a compound object is traversed, its recursive results are `folded` together (if necessary) into a single result by the methods inherited from `TU` . In the actual implementation of DemeterF we extend `TU` in order to collect the type definitions from the tree of included CD files.

2.7.1.2 Type-Preserving Functions

Type-preserving functions include transformations or functional updates to a particular part of a structure. This category includes functions like substitution or variable index calculations. DemeterF provides a class, `TP`, that rebuilds the data structure it traverses. Each *`combine`* method simply calls the corresponding constructor of its first argument. Clients implement specific *`combine`* methods for the part of the structure to be transformed and the rest of the methods automatically reconstruct.

[13] We use a functional implementation of `Set` from the DemeterF library, so all methods return a new `Set`, rather than using mutation.

Listing 2.21 shows an example function-class, `Invert`, that inverts `True` and `False` literals within a `BExp` instance. When a literal, `True` or `False`,

```
class Invert extends TP{
   False combine(True l){ return new False(); }
   True combine(False l){ return new True(); }
}
```

Listing 2.21: Invert `True`/`False` instances using `TP`

is reached, one of our *combine* methods will be called. Otherwise, the inherited methods of `TP` rebuild compound `BExp`s (or `Bind`s if we have them) using the results of recursive subtraversals to create a new instance. In the implementation of DemeterF we extend `TP` to implement type parameter substitution and to push global CD properties into local type definitions.

2.7.2 Generating Function-Classes

Many of our earlier functions are specific to our `BExp` datatypes (*e.g.*, `ToString`), but more general function-classes use implementations of `TU` and `TP` to generically adapt to a data structure. DemeterF allows programmers to write functions over the *structure* of CDs that generate function-classes to be used with a traversal.[14] Though our implementation is complicated by parametrized types, we essentially traverse the abstract syntax tree of a CD to produce a function-class with specialized *combine* methods. In this section we give abstract specifications of our generation (*i.e.*, *compilation*) of generic function-classes from CD definitions by way of simple *rewrite* rules.

2.7.2.1 Abstract CDs

At runtime our structures are only made up of concrete classes, so generated function-classes depend only on the structure of concrete classes. Before generating function-classes, DemeterF transforms more complex CDs into a simpler representation by pushing common fields from abstract classes down into concrete subtypes. For the purpose of generating function-classes it is usually enough to view a CD as a list of concrete class definitions of the form:

$$C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n$$

Where each type, T_i, can be either abstract or concrete. The field names, f_i, are actually not important, but we use them to keep the names of method parameters consistent. Since fields of abstract definitions are taken into

[14] It is possible to write external function-classes to be loaded at runtime, but the author of this dissertation has provided a number of useful classes that are easy to use. So client extensions are rarely necessary.

account by concrete subclasses, we view abstract classes simply as a list of bar separated subtypes:

$$A = T_1 \mid \cdots \mid T_n$$

In this section we use these simplified definitions to describe rewrite rules for generating function-classes from the definitions of a particular CD.

2.7.2.2 Printing: `Show`

Printing in various forms has typically been a generated function in Adaptive Programming tools, *e.g.*, DemeterJ [66]. In DemeterF we define the generation of a CD-based function-class as a function from concrete definitions to *combine* methods. As an example of a print related function-class, we demonstrate the generation of `Show`, a common derivable type class in Haskell [34]. We will use templates to describe the format of our resulting function-classes.

The template for `Show` is given in Listing 2.22. The template simply pro-

```
class Show extends FC{
   // Convert primitives
   string combine(int p){ return ""+p; }
   /*... Other primitive types ... */

   // Generate the rest with GenShow
   ∀C ∈ CD . GenShow(C)
}
```

Listing 2.22: `Show` generation template

vides a class definition and *combine* methods for primitives that convert each into a `String`. The rest of the body of `Show` is generated by GenShow, using a simple rewrite rule mapped to each concrete definition from the CD:

$$\begin{aligned}&\textsc{GenShow}(\, C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\,) \rightsquigarrow \\ &\quad \texttt{String } \textit{combine}(C\ \texttt{\_h, String } f_1\texttt{, } \cdots\texttt{, String } f_n) \\ &\quad \{\ \textbf{return } \texttt{"}C\texttt{("+}f_1\texttt{+",",+}\cdots\texttt{+",",+}f_n\texttt{+")"; }\ \}\end{aligned}$$

For each concrete definition with n fields we create a *combine* method with $n + 1$ arguments. The first is of type C, the defined type, and the rest are of type `String`. During the traversal of an object using an instance of `Show`, the field traversals will recursively convert the fields into strings before calling the matching *combine*. Within each method, the return `String` is constructed by concatenating the separating the recursive field results wit commas, wrapping them in parentheses, and prefixing the `String` with the class name, C.

Listing 2.23 gives a portion of the generated `Show` function-class for our `BExp` CD.

```
class Show extends FC{
   /* ... */
   String combine(Neg _h, String inner)
   { return "Neg("+")"; }
   String combine(And _h, String l, String r)
   { return "And("+l+","+r+")"; }
   /* ... */
}
```

Listing 2.23: `Show` generated for `BExps`

2.7.2.3 Type Unifying Functions

While the generic (reflective) `TU` class works for all structure, we can use the concrete class definitions in a CD to generate the equivalent function-class that does not require the use of reflection. We provide a template that is parametrized by the eventual return type, `X`, shown in Listing 2.24. We de-

```
class TU<X> : FC{
   // Methods to override
   abstract X fold(X a, X b);
   abstract X combine();

   // Primitives call default
   X combine(int p){ return combine(); }
   /*... Other primitive types ...*/

   // Generate the body with GenTU
   ∀C ∈ CD.GenTU(C)
}
```

Listing 2.24: `TU` generation template

clare the abstract methods for producing the default result (*combine()*) and folding together two recursive results, respectively. Primitive *combine* methods can be overridden, but initially return the default result. Our generation rule for concrete definitions is a generalization of that for `Show`:

$$\begin{aligned}&\textsc{GenTU}(\, C = \langle f_1\rangle\ T_1 \cdots \langle f_n\rangle\ T_n\,) \rightsquigarrow\\ &\quad \texttt{X } \textit{combine}(C\ \texttt{\_h, X } f_1\texttt{, } \cdots\texttt{, X } f_n)\\ &\quad \{\ \textbf{return } \texttt{fold}(f_1\texttt{, fold}(f_2\texttt{, } \cdots))\texttt{;}\ \}\end{aligned}$$

Each generated *combine* method accepts $n+1$ parameters: again the first of type C, but the rest are of our type parameter `X`. If necessary, the return result is computed by nested calls to `fold`. Listing 2.25 shows the resulting `TU` class, specialized for our `Exp` CD. The generated version of is a direct replacement for the generic/reflective version used in Listing 2.20. The generated function-class gives us much better performance, especially when we can inline traversals [19].

```
class TU<X> extends FC{
   /* ... */
   X combine(Neg _h, X inner){ return inner; }
   X combine(And _h, X l, X r){ return fold(l,r); }
   /* ... */
}
```

Listing 2.25: TU generated for BExps

2.7.2.4 Type Preserving Functions

Our last function-class generation example is probably the most useful. We use it often to do recursive functional updates and transformations over different types. Since *combine* methods are optional for primitive types we leave them out of our template, shown in Listing 2.24. Our generation rule

```
class TP extends FC{
   // Generate the body with GenTP
   ∀C ∈ CD . GenTP(C)
}
```

Listing 2.26: TP generation template

creates a *combine* method that simply reconstructs a new C instance from the recursive traversals results.

$$\begin{aligned}&\textsc{GenTP}(\, C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\,) \rightsquigarrow \\ &\quad C\ \texttt{combine}(C\ \texttt{\_h},\ T_1\ f_1,\ \cdots,\ T_n\ f_n) \\ &\quad \{\ \texttt{return new}\ C(f_1,\ \cdots,\ f_n);\ \}\end{aligned}$$

Because the transformation is type preserving, each field result type is the same as its defined type, T_i. The resulting generated TP class for our Exp CD is shown in Figure 2.27.

```
class TP extends FC{
   /* ... */
   Neg combine(Neg _h, BExp inner)
   { return new Neg(inner); }
   And combine(Add _h, BExp l, BExp r)
   { return new And(l, r); }
   /* ... */
}
```

Listing 2.27: TP generated for BExps

2.8 Errors and Assumptions

Having seen several examples of our DemeterF library and implemented operations, it is worth going over the assumptions that DemeterF makes and the different errors that can occur when using and writing traversal-based functions. As with any Java-based library, programmers can raise a traditional `RuntimeExceptions` during the execution of a traversal and within *combine* methods. DemeterF does not attempt to interact with Java's exception mechanism, so programmer raised errors behave as expected.

DemeterF assumes a bit more about the structures that will be traversed. While class definitions generated from a CD do not (by default) support mutation, Java will still allow local mutation and mutation of hand-written classes, which allows programmers to construct cyclic instances. DemeterF assumes that traversed structures are acyclic, but traversal based functions can be written for cyclic structures by using `Control` to avoid infinite recursion.[15]

All the function-classes presented thus far have been *type-correct* and *complete* with respect to the structures being traversed. In each case the *combine* method signatures have handled all possible cases, including recursive results. However, when this is not the case DemeterF raises a `RuntimeException` during method selection, when a suitably typed *combine* method cannot be found.

A simple example of an incomplete function-class is shown in Listing 2.28. Within `TypeError` we have a *combine* method that handles the `Lit` case, but

```
class TypeError extends FC{
   String combine(Lit l){ return "Lit"; }

   String error(){
      BExp e = new And(new True(), new False());
      return new Traversal(this).traverse(e);
   }
}
```

Listing 2.28: Function-class that causes a dispatch error

not one that handles `And`. Calling the `error` method of a `TypeError`:

```
new TypeError().error()
```

Results in a DemeterF runtime error:

```
DemeterF: Did not find a match for:
    TypeError.combine(And, String, String)
```

Stating that a matching *combine* method for the signature `(And, String, String)` could not be found in the given function-object. In this case the

[15] The traversal of shared structures behaves as expected, though a shared instance may be traversed multiple times.

problem is easy to fix by adding a new case for `And`, but what we want is to be certain that a `Traversal` will *never* raise such an error for a combination of data structure and function-class. Modeling DemeterF traversals in order to statically eliminate such dispatch errors (*i.e.*, ensuring *safety*) is the main topic of the next chapter.

CHAPTER 3

A Model of DemeterF

Now that we have discussed the features of our DemeterF implementation, in this chapter we formally describe the syntax and semantics of a simplified model, which we refer to as AP-F. AP-F captures the key aspects of DemeterF's CD definitions, adaptive generic traversal, and type-based multiple dispatch. We use the model to give our traversals and dispatch a precise semantics. With the given semantics we then define a type system, which guarantees that well-typed traversals are free from dispatch errors. We provide a proof of type soundness, then complete the chapter with a discussion of extensions to the model that would bring it in line with the implementation of DemeterF.

3.1 Syntax

We begin by giving a minimal description of our minimal syntax, which embodies the key aspects of DemeterF CDs, traversals, and function-classes. Our model syntax is shown in Figure 3.1. Aside from general Java features like classes and local definitions, our most notable emissions are base types and field names. Our syntactic categories are partitioned into variable names, x, concrete type names, C, and abstract type names A. An AP-F program, P, is a sequence of data structure definitions (`abstract` and `concrete` types) followed by an expression. Abstract and concrete types correspond to abstract (*i.e.*, no fields) and concrete classes in a DemeterF CD. Concrete type definitions mention only the types of their "fields", since functions will be used to rename structural elements during traversal.

Expressions, e, are either variable references, constructor calls (`new`), or traversals. We model the simplest form of DemeterF traversal, representing the traversal of a structure instance using a given a functions-class. Function sets, F, and functions, f, represent DemeterF function-classes and *`combine`* methods respectively. A function set, `funcset`, is a sequence of functions, each of which is a sequence of type/argument pairs followed by a `return` and body expression in Java-like syntax. Function return types are left out, since they can be inferred from the argument types and body expression.

$$
\begin{array}{rcl}
x & ::= & \text{variable names} \\
C & ::= & \text{concrete type names} \\
A & ::= & \text{abstract type names} \\
T & ::= & C \mid A \\
P & ::= & D_1 \ldots D_n \;\; e \\
 & & \\
D & ::= & \texttt{concrete}\ C\ \texttt{=}\ T_1 * \ldots * T_n\ . \\
 & \mid & \texttt{abstract}\ A\ \texttt{=}\ T_0 \mid \ldots \mid T_n\ . \\
e & ::= & x \mid \texttt{new}\ C\,\texttt{(}\,e_1\texttt{,}\ \ldots\texttt{,}\ e_n\,\texttt{)} \mid \texttt{traverse(}\,e_0\texttt{,}\ F\,\texttt{)} \\
F & ::= & \texttt{funcset\{}\ f_1 \ldots f_n\ \texttt{\}} \\
f & ::= & \texttt{(}T_0\ x_0\texttt{,}\ \ldots\ \texttt{,}\ T_n\ x_n\texttt{)}\ \texttt{\{ return}\ e\texttt{; \}}
\end{array}
$$

Figure 3.1: AP-F Model Language Syntax

3.1.1 Subtyping

Based on the definitions in a program, we define a subtype relation, $\leq$, as the reflexive, transitive closure of the immediate subtype relationship from `abstract` definitions. Our definition is given by three rules, shown in Figure 3.2.

$$
\text{[S-REFL]}\ \frac{}{T \leq T} \qquad \text{[S-DEF]}\ \frac{\texttt{abstract}\ A\ \texttt{=}\ T_0 \mid \ldots \mid T_n\ . \in P}{T_i \leq A}
$$

$$
\text{[S-TRANS]}\ \frac{T \leq T'' \qquad T'' \leq T'}{T \leq T'}
$$

Figure 3.2: Subtyping Rules

The subtype relation will be used primarily to our define our multiple-dispatch, but we will also use it in our type system to relate the type of a data structure to the types of possible return values, when an instance is traversed with a `funcset`.

3.1.2 Example

Our model does not include base types, but our basic boolean expression structures (from chapters 1 and 2) can still be defined. The `BExp` CD-like definitions are shown in Listing 3.1. To complete the program definition we construct a simple `BExp` in the body of the program representing $(\mathit{true} \wedge \neg \mathit{false})$.

```
// ASTs for boolean expressions
abstract BExp = Lit | Neg | And | Or.
abstract Lit  = True | False.
concrete True = .
concrete False = .
concrete Neg = BExp.
concrete And = BExp * BExp.
concrete Or  = BExp * BExp.

// Simple program body
new And(new True(),
        new Neg(new False()))
```

Listing 3.1: Model Example: Boolean expression structures

3.2 Well Formed Rules

In order to avoid purely syntactic problems in our semantics, we restrict syntactically valid programs with a few *well-formed* rules. They check the sanity of a program's definitions and allow us to focus on the key issues of our semantics.

TYPESONCE(P): Each type must only be defined once.

COMPLETETYPES(P): Each type used in the right-hand side of a definition must itself be defined.

NOSELFSUPER(P): Each `abstract` type must not occur in the right-hand side of its own definition.

SINGLESUPER(P): Each type should occur in the right-hand side of at most one `abstract` definition.

The first two rules check for the existence and completeness of a program's definitions: TYPESONCE ensures that each type is *defined* only once, and COMPLETETYPES makes sure each type *use* corresponds to a defined type. The rules do not restrict recursion in the data structures or the shapes that can be defined, since they only require that a definition exists and is unique.

Our SINGLESUPER rule enforces a simplifying assumption on our type hierarchies, which restricts types to a form of single inheritance. Together with NOSELFSUPER, the rules ensure a linear supertype relation: each type may only have one supertype. Linearizing supertypes gives us a total ordering on function signatures: each `abstract` type can have multiple subtypes, but only one supertype. We requiring a total order on function signatures in order to simplify our dispatch semantics and avoid the usual *diamond problem* when multiple inheritance and multiple dispatch interact [56, 21].

3.3 Semantics

We use a (small-step) reduction semantics to model DemeterF traversals. We begin with a description of values, v, runtime expressions, e, and evaluation contexts, E, described in Figure 3.3. Values are constructor calls in which all

$$
\begin{array}{rcl}
v & ::= & \texttt{new}\ C\,(\,v_1\texttt{,}\ \ldots\texttt{,}\ v_n\,) \\
e & ::= & \cdots \\
 & \mid & \texttt{dispatch(}\,F\texttt{,}\ v_0\texttt{,}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\,\texttt{)} \\
 & \mid & \texttt{apply(}\,f\texttt{,}\ v_0\texttt{,}\ v_1\texttt{,}\ \ldots\texttt{,}\ v_n\,\texttt{)} \\
 & & \\
E & ::= & [\,] \\
 & \mid & \texttt{new}\ C\,(\,v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)} \\
 & \mid & \texttt{traverse(}\,E\texttt{,}\ F\,\texttt{)} \\
 & \mid & \texttt{dispatch(}\,F\texttt{,}\ v_0\texttt{,}\ v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)}
\end{array}
$$

Figure 3.3: Values, runtime expressions, and evaluation contexts

sub-expressions are also values. Runtime expressions (`dispatch` and `apply`) are not part of our surface syntax, but are used to model structural recursion and function application respectively. The use of `apply` is mainly cosmetic in order to avoid complicating eventual rules involving `dispatch`. Evaluation contexts account for our reduction strategy. Reduction can occur under the empty context, $[\,]$ constructor application, the left argument of a traversal expression, or under a `dispatch` expression. Overall our evaluation contexts ensure that our reduction strategy is deterministic and *left-most/inner-most*.

Figure 3.4 contains definitions of our reflective meta-functions and substitution. The function *types* is used to return the concrete types of a list of sequence of values. Others functions, *argtypes* and *functions*, are simply convenient accessors for converting between abstract syntax and meta representations.

We denote the substitution of a value, v, for a variable, x, within an expression, e, by $e[v/x]$. Substitution is defined over all terms, including functions and function sets. Within function definitions, substitution only occurs when the variable, x, is free in the function body. Since only values can be substituted, and functions are not first-class, α-conversion or renaming is not necessary to avoid capture.

Figure 3.5 completes our meta-functions with signature comparison and type-based function selection implemented by *choose*. The helper function *chooseOne* selects the most specific applicable function in a `funcset`, given the actual argument types. *possibleFs* filters the function set, returning only the functions that are *possible* to apply to the given types. *possible* returns true if all arguments are element-wise related, since a function may be applied to subtypes of its argument types or when actual arguments are refined

$$
\begin{array}{l}
\mathit{types}(\texttt{new } C_0\texttt{(}\cdots\texttt{)}, \ldots, \texttt{new } C_n\texttt{(}\cdots\texttt{)}) \;=\; (C_0 \ldots C_n) \\
\mathit{argtypes}(\texttt{(}T_0\ x_0\texttt{, } \ldots \texttt{, } T_n\ x_n\texttt{)\{ return } e\texttt{; \}}) \;=\; (T_0 \ldots T_n) \\
\mathit{functions}(\texttt{funcset\{ } f_1 \ldots f_n \texttt{ \}}) \;=\; (f_1 \ldots f_n)
\end{array}
$$

$$
\begin{array}{ll}
x[v/x] & = v \\
x'[v/x] & = x' \quad \text{if } x' \neq x \\
\texttt{new } C\texttt{(}e_1\texttt{, } \ldots \texttt{, } e_n\texttt{)}[v/x] & = \texttt{new } C\texttt{(}e_1[v/x]\texttt{, } \ldots \texttt{, } e_n[v/x]\texttt{)} \\
\texttt{traverse(}e_0\texttt{, } F\texttt{)}[v/x] & = \texttt{traverse(}e_0[v/x]\texttt{, } F[v/x]\texttt{)} \\
\texttt{dispatch(}F\texttt{, } v_0\texttt{, } e_1\texttt{, } \ldots \texttt{, } e_n\texttt{)}[v/x] & = \texttt{dispatch(}F[v/x]\texttt{, } v_0\texttt{, } e_1[v/x]\texttt{, } \ldots \texttt{, } e_n[v/x]\texttt{)} \\
\texttt{apply(}f\texttt{, } v_0\texttt{, } v_1\texttt{, } \ldots \texttt{, } v_n\texttt{)}[v/x] & = \texttt{apply(}f[v/x]\texttt{, } v_0\texttt{, } v_1\texttt{, } \ldots \texttt{, } v_n\texttt{)} \\
\texttt{funcset\{ } f_1 \ldots f_n \texttt{ \}}[v/x] & = \texttt{funcset\{ } f_1[v/x] \ldots f_n[v/x] \texttt{ \}} \\
\texttt{(}T_0\ x_0\texttt{, } \ldots\texttt{) \{ return } e\texttt{; \}}[v/x] & = \texttt{(}T_0\ x_0\texttt{, } \ldots\texttt{) \{ return } e\texttt{; \}} \quad \text{if } x \in \overline{x_i} \\
\texttt{(}T_0\ x_0\texttt{, } \ldots\texttt{) \{ return } e\texttt{; \}}[v/x] & = \texttt{(}T_0\ x_0\texttt{, } \ldots\texttt{) \{ return } e[v/x]\texttt{; \}} \quad \text{if } x \notin \overline{x_i}
\end{array}
$$

Figure 3.4: Reflection and Substitution Definitions

$$
\mathit{choose}(F, (C_0 \ldots C_n)) \;=\; \mathit{chooseOne}(\, \mathit{possibleFs}(F, (C_0 \ldots C_n)), (C_0 \ldots C_n))
$$

$$
\begin{array}{rl}
\mathit{chooseOne}((), (T_0 \ldots T_m)) = & \texttt{error} \\
\mathit{chooseOne}((f_0\ f_1 \ldots f_n), (T_0 \ldots T_m)) = & \mathit{best}(f_0, (f_1 \ldots f_n), (T_0 \ldots T_m))
\end{array}
$$

$$
\begin{array}{rl}
\mathit{best}(f, (), (T_0 \ldots T_m)) = & f \\
\mathit{best}(f, (f_0\, f_1 \ldots f_n), (T_0 \ldots T_m)) = & \texttt{if } \mathit{better}(\mathit{argtypes}(f_0), \mathit{argtypes}(f)) \\
& \texttt{then } \mathit{best}(f_0, (f_1 \ldots f_n), (T_0 \ldots T_m)) \\
& \texttt{else } \mathit{best}(f, (f_1 \ldots f_n), (T_0 \ldots T_m))
\end{array}
$$

$$
\begin{array}{rl}
\mathit{better}((), ()) = & \texttt{false} \\
\mathit{better}((T_0\ T_1 \ldots T_n), (T'_0\ T'_1 \ldots T'_n)) = & ((T_0 \not\equiv T'_0 \ \wedge\ T_0 \leq T'_0) \vee \\
& (T_0 \equiv T'_0 \ \wedge\ \mathit{better}((T_1 \ldots T_n), (T'_1 \ldots T'_n))))
\end{array}
$$

$$
\mathit{possibleFs}(F, (T_0 \ldots T_n)) \;= \mathit{filter}(\lambda\, f.\, \mathit{possible}(\mathit{argtypes}(f), (T_0 \ldots T_n)), \mathit{functions}(F))
$$

$$
\begin{array}{rl}
\mathit{possible}((), ()) = & \texttt{true} \\
\mathit{possible}((), (T'_0 \ldots T'_m)) = & \texttt{false} \\
\mathit{possible}((T_0 \ldots T_n), ()) = & \texttt{false} \\
\mathit{possible}((T_0\ T_1 \ldots T_n), (T'_0\ T'_1 \ldots T'_m)) = & (T'_0 \leq T_0 \ \vee\ T_0 \leq T'_0) \wedge \\
& \mathit{possible}((T_1 \ldots T_n), (T'_1 \ldots T'_m))
\end{array}
$$

Figure 3.5: Function Selection Meta-functions

from supertypes. At runtime however, the actual argument types will always be **concrete** and without subtypes, so the second check, $T_0 \leq T'_0$, is irrelevant. This check only becomes important when we use *possibleFs* with approximate types, as is necessary during type checking. *chooseOne* uses *best* to select the most specific function in the filtered set, using *better* to compare function signatures. For simplicity we compare only functions with the same number of arguments, though dispatch in our DemeterF implementation is more flexible, allowing functions to ignore later arguments.

Finally, Figure 3.6 gives a relation, $\rightarrow$, which completes our small-step

semantics with a *notion of reduction*, *i.e.*, with axioms or contraction rules. The left-hand side of each rule represents a potential reducible expression,

[R-TRAV]
$\texttt{traverse}(\, v_0\,,\, F\,)$
$\quad \rightarrow \texttt{dispatch}(\, F\,,\, v_0\,, \texttt{traverse}(\, v_1\,,\, F\,)\,,\, \ldots\,, \texttt{traverse}(\, v_n\,,\, F\,))$
where $v_0 = \texttt{new}\ C(\, v_1\,,\, \ldots\,,\, v_n\,)$

[R-DISPATCH]
$\texttt{dispatch}(\, F\,,\, v_0\,,\, v_1\,,\, \ldots\,,\, v_n) \rightarrow \texttt{apply}(\, f\,,\, v_0\,,\, v_1\,,\, \ldots\,,\, v_n\,)$ if $f \neq \texttt{error}$
where $f = \mathit{choose}(F,\, \mathit{types}(\, v_0\ v_1\ \ldots\ v_n))$

[R-APPLY]
$\texttt{apply}(\, (T_0\ x_0\,,\, \ldots\ ,\, T_n\ x_n)\, \{\, \texttt{return}\ e\,;\, \}\,,\, v_0\,,\, v_1\,,\, \ldots\,,\, v_n\,) \rightarrow e[\overline{v_i/x_i}]$

Figure 3.6: Reduction Rules

or *potential redex*. If a potential redex can be contracted then it is considered an actual redex, *i.e.*, no longer potential.

A `traverse` expression with a constructed value as its first argument can be contracted (R-TRAV) producing a `dispatch` expression. We include the function set, F, the original value, v_0, and wrap each field of the value in a `traverse` expression that uses the same function set. A `dispatch` expression containing only values can be contracted (R-DISPATCH) to an `apply` expression, when the result of *choose* is not **`error`**. A `dispatch` expression that violates the side condition is considered *stuck*, *i.e.*, a potential but not actual redex. Any expression that contains a nested stuck expression is itself considered stuck, since contraction cannot occur. A stuck expression represents a runtime dispatch error from a DemeterF traversal. Our last rule (R-APPLY) is an extension of R-DISPATCH, substituting the given values for the formal parameters of the selected function. We use overbar notation, $e[\overline{v_i/x_i}]$, to represent repetitive substitutions: $(e[v_0/x_0]\,[v_1/x_1]\ \cdots)$.

3.3.1 From Reduction to Evaluation

Following Danvy's lecture notes at APF'08 [23], a one-step reduction function can be defined that decomposes non-value expression into an evaluation context, E, and a potential redex. If the potential redex can be contracted, then the resulting contractum can be recomposed with (plugged into) the evaluation context resulting in a reduced program. Figure 3.7 gives sketches of the functions *reduce*, *decmp*, and *recmp* that implement the one-step reduction function of our semantics.

We define *reduce* as decomposition followed by contraction and recomposition, when one of our reduction rules applies. The function *decmp* traverses an expression while accumulating an evaluation context. Expression cases

$$
\begin{aligned}
\mathit{reduce}(v) &= v \\
\mathit{reduce}(e) &= \texttt{let}\ \langle e', E\rangle = \mathit{decmp}(e, [\,]) \\
&\qquad \texttt{in}\ \mathit{recmp}(e'', E) \\
&\quad \text{if}\ e' \rightarrow e''
\end{aligned}
$$

$$
\begin{aligned}
\mathit{decmp}(\texttt{new}\ C\,\texttt{(}\,v\ \ldots\texttt{,}\ e_0\texttt{,}\ e\ \ldots\,\texttt{)}, E) &= \mathit{decmp}(\texttt{new}\ C\,\texttt{(}\,v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)}, e_0) \\
\mathit{decmp}(\texttt{traverse(}\,e_0\texttt{,}\ F\,\texttt{)}, E) &= \mathit{decmp}(\texttt{traverse(}\,E\texttt{,}\ F\,\texttt{)}, e_0) \\
\mathit{decmp}(\texttt{dispatch(}\,F\texttt{,}\ v\ \ldots\texttt{,}\ e_0\texttt{,}\ e\ \ldots\,\texttt{)}, E) &= \mathit{decmp}(\texttt{dispatch(}\,F\texttt{,}\ v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)}, e_0) \\
\mathit{decmp}(e, E) &= \langle e, E\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{recmp}(e, [\,]) &= e \\
\mathit{recmp}(e_0, \texttt{new}\ C\,\texttt{(}\,v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)}) &= \mathit{recmp}(\texttt{new}\ C\,\texttt{(}\,v\ \ldots\texttt{,}\ e_0\texttt{,}\ e\ \ldots\,\texttt{)}, E) \\
\mathit{recmp}(e_0, \texttt{traverse(}\,E\texttt{,}\ F\,\texttt{)}) &= \mathit{recmp}(\texttt{traverse(}\,e_0\texttt{,}\ F\,\texttt{)}, E) \\
\mathit{recmp}(e_0, \texttt{dispatch(}\,F\texttt{,}\ v\ \ldots\texttt{,}\ E\texttt{,}\ e\ \ldots\,\texttt{)}) &= \mathit{recmp}(\texttt{dispatch(}\,F\texttt{,}\ v\ \ldots\texttt{,}\ e_0\texttt{,}\ e\ \ldots\,\texttt{)}, E)
\end{aligned}
$$

Figure 3.7: Functions for one-step reduction

that match evaluation contexts are handled explicitly by recurring on the inner, left-most non-value expression. Other expressions, *e.g.*, `apply`, match the final case returning a pair of the potential redex and inverted context. *recmp* does the reverse, building an expression and composing evaluation contexts until the empty context, $[\,]$, is reached.

Our one-step reduction function can be used to iteratively define an evaluation function, as shown in Figure 3.8. The function *evaluate* implements

$$
\begin{aligned}
\mathit{evaluate}(v) &= v \\
\mathit{evaluate}(e) &= \mathit{evaluate}(\,\mathit{reduce}(e)\,) \\
&\quad \text{if}\ e\ \text{is not}\ \textit{stuck}
\end{aligned}
$$

Figure 3.8: Reduction-based Evaluation Function

the iteration of the one-step reduction function from Figure 3.7. This definition can be 'refocused' into an abstract machine, and further transformed resulting in a more typical big-step evaluation function [23, 24], but the version of Figure 3.8 is sufficient for our purposes. For reasons of efficiency our actual DemeterF implementation is, of course, based on big-step evaluation.

3.3.2 Example

With our example definitions of Listing 3.1, we can add a simple traversal and function set that implements (*strict*) `BExp` evaluation, shown in Listing 3.2. Again, without base types, we construct an expression representing $(\mathit{true} \wedge \neg \mathit{false})$ and `traverse` it using a `funcset`. Our function set is similar to the `Eval` function-class from Listing 2.12 in Section 2.3. The traversal of the expression produces a `Lit`, representing a result of `True` or `False`. Similar to

```
// ... Definitions from Listing 3.1 ...

traverse(new And(new True(),
                 new Neg(new False())),
         funcset{
            (Lit l){ return l; }
            (Neg n, True t){ return new False(); }
            (Neg n, False f){ return new True(); }
            (And a, True l, True r){ return r; }
            (And a, Lit l, Lit r){ return new False(); }
            (Or o, False l, False r){ return r; }
            (Or o, Lit l, Lit r){ return new True(); }
         })
```

Listing 3.2: Model Example: Boolean expression evaluation

the DemeterF example, multiple dispatch is used to match interesting cases during traversal. For `Neg` this means matching `True` or `False` and returning its negation; for `And` or `Or` this means capturing the all-true and all-false cases respectively. The other two cases for `And` and `Or` are handled by more general signatures using `Lit`.

3.4 Type System

Like regular Java programs, those written using our DemeterF system can raise many different kinds of errors, unrelated to traversal. Our model has been specifically designed to eliminate all but those relating to function sets, and dispatch. In order to rule out runtime errors and predict the class of values a program may return, we impose a type system on our model. Though our type system rules out standard errors like unbound variable uses, we are mostly interested in eliminating errors resulting from function selection (*choose* and *chooseOne* in Figure 3.5).

For any type-correct program we obtain a typing derivation that constrains the return values of traversals and function sets based on the shape of the datatypes. Our judgment (*well-typed*) is separated into three mutually recursive relations; one for each of expressions, functions, and traversals. We standard variable type environments, Γ, for typing expressions and functions. For traversals we use an additional environment, $\mathcal{X}$, to track the return types of recursive datatype traversals. We represent environments as a list of pairs, with syntax shown in Figure 3.9.

$$\begin{aligned} \Gamma &::= \emptyset \mid \Gamma, x{:}T \\ \mathcal{X} &::= \emptyset \mid \mathcal{X}, T{:}T' \end{aligned}$$

Figure 3.9: Variable and Traversal Environments

In certain typing rules we will denote the set of the left-hand sides of pairs

from Γ (also $\mathcal{X}$) by $dom\ \Gamma$. New pairs will be appended to environments, and lookup, denoted $\Gamma(x)$, will occur from right to left, selecting the latest binding if duplicate names exist.

3.4.1 Functions

We begin with the simplest of our typing rules. Since functions are not first-class values, type-checking a function depends only on the type of its body expression when parameters are bound to the types given in its signature. Our rule for $\vdash_F$ is shown in Figure 3.10.

$$\text{[T-FUNC]}\quad \frac{(\Gamma,\ x_0{:}T_0,\ \ldots,\ x_n{:}T_n)\ \vdash_e e_0 : T}{\Gamma \vdash_F \texttt{(}T_0\ x_0\texttt{, } \ldots \texttt{ , } T_n\ x_n\texttt{)\{ return } e_0\texttt{; \}} : T}$$

Figure 3.10: Function Typing Rule

3.4.2 Expressions

Figure 3.11 shows our typing rules for expressions ($\vdash_e$). Variables must be

$$\text{[T-VAR]}\quad \frac{x \in dom\ \Gamma}{\Gamma \vdash_e x : \Gamma(x)} \qquad \text{[T-NEW]}\quad \frac{\texttt{concrete } C \texttt{ = } T_1 * \ldots * T_n \texttt{ .} \in P \quad \text{for } i \in 1..n \quad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i}{\Gamma \vdash_e \texttt{new } C \texttt{ (} e_1\texttt{, } \ldots\texttt{, } e_n\texttt{)} : C}$$

$$\text{[T-TRAV]}\quad \frac{\Gamma \vdash_e e_0 : T_0 \quad \Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F\rangle : T}{\Gamma \vdash_e \texttt{traverse(}\ e_0\texttt{, }\ F\ \texttt{)} : T}$$

$$\text{[T-DISPATCH]}\quad \frac{\begin{array}{c}\emptyset \vdash_e v_0 : C \\ \text{for } i \in 1..n \quad \Gamma \vdash_e e_i : T_i' \\ \text{for } f \in \mathit{possibleFs}(F, (C\ T_1' \ldots T_n')) \quad \Gamma \vdash_F f : T_f \quad T_f \leq T \\ \mathit{covers}(F, (C\ T_1' \ldots T_n'))\end{array}}{\Gamma \vdash_e \texttt{dispatch(}\ F\texttt{, }\ v_0\texttt{, }\ e_1\texttt{, }\ \ldots\texttt{, }\ e_n\texttt{)} : T}$$

$$\text{[T-APPLY]}\quad \frac{f = \texttt{(}T_0\ x_0\texttt{, } \ldots \texttt{ , } T_n\ x_n\texttt{)\{ return } e\texttt{; \}} \quad \Gamma \vdash_F f : T \quad \text{for } i \in 0..n \quad \emptyset \vdash_e v_i : T_i' \quad T_i' \leq T_i}{\Gamma \vdash_e \texttt{apply(}\ f\texttt{, }\ v_0\texttt{, }\ v_1\texttt{, }\ \ldots\texttt{, }\ v_n\texttt{)} : T}$$

Figure 3.11: Expression Typing Rules

bound to a type in the environment (T-VAR) and value construction requires

subtypes (T-NEW) for each expression (*i.e.*, *field*) of a concrete structure. Traversal expressions (T-TRAV) delegate to a more specialized judgment, $\vdash_{\mathcal{T}}$ (presented in Section 3.4.3), passing the variable environment and an empty traversal environment, $\mathcal{X} = \emptyset$. For `dispatch` expressions (T-DISPATCH) we use *possibleFs* to be sure all *possible* functions unify to a common supertype. Function application (T-APPLY) requires subtypes of a function's formal parameter types.

One subtle (but key) aspect of the T-DISPATCH rule is the use of the meta-function, *covers*. Its properties will be discussed in Section 3.4.4, but the main idea of *covers* is to verify that a function set, F, contains a possible function for each possible argument sequence of `concrete` types that are subtypes of the given sequence. In T-DISPATCH, this means that F has at least one function that can be applied to possible *values* of the given types. The use of *covers* in this rule corresponds to our typing rules for `concrete` traversals, which is discussed in the next section.

3.4.3 Traversals

Traversal expressions are typed using a specific judgment, $\vdash_{\mathcal{T}}$, that takes into account the types of functions in the set and the program's data structure definitions. The two rules, one for each of `abstract` and `concrete` types, are shown in Figure 3.12.

[T-ATRAV]

$$\frac{\begin{array}{c}\texttt{abstract}\ A\ \texttt{=}\ T_0 \mid \ldots \mid T_n\ \texttt{.} \in P \\ \text{for } i \in 1..n \quad T_i \in dom\,\mathcal{X} \ \Rightarrow\ T_i' = \mathcal{X}(T_i) \\ \text{for } i \in 1..n \quad T_i \notin dom\,\mathcal{X} \ \Rightarrow\ \Gamma; \mathcal{X},\ A{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \\ \text{for } i \in 1..n \quad T_i' \leq T\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T}$$

[T-CTRAV]

$$\frac{\begin{array}{c}\texttt{concrete}\ C\ \texttt{=}\ T_1 * \ldots * T_n\ \texttt{.} \in P \\ \text{for } i \in 1..n \quad T_i \in dom\,\mathcal{X} \Rightarrow\ T_i' = \mathcal{X}(T_i) \\ \text{for } i \in 1..n \quad T_i \notin dom\,\mathcal{X} \Rightarrow\ \Gamma; \mathcal{X},\ C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \\ \text{for } f \in possibleFs(F,\ (C\ T_1' \ \ldots\ T_n')) \quad \Gamma \vdash_F f : T_f \quad T_f \leq T \\ covers(F,\ (C\ T_1' \ \ldots\ T_n'))\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

Figure 3.12: Traversal Typing Rules

We read $\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle T, F\rangle : T'$ as follows:

> *In type environment Γ with traversal types $\mathcal{X}$ the traversal of a value of type T with function set F returns a value of type T'.*

Γ is the standard variable type environment. $\mathcal{X}$ is an environment of traversal return types for (possibly recursive) types that may depend on the traversal return of T. The function set F is constant for a given expression, and is passed throughout a derivation.

The typing of the traversal of an `abstract` type proceeds by typing each of its elements T_i separately. If a binding for T_i exists in $\mathcal{X}$ (*i.e.*, $T_i \in dom\,\mathcal{X}$) then the result, T_i', must be the same as the bound result type, which we denote $\mathcal{X}(T_i)$. Otherwise, we calculate the result type by adding $A : T$ to $\mathcal{X}$ using the same function set, F. The final line of the premise constrains the result type for the `abstract` type to be a common supertype of the traversal the individual elements.

The rule for `concrete` types is more involved, due to function selection. Similar to `abstract` types, for *field* types bound in $\mathcal{X}$, must be the same as the bound result type, *i.e.*, $T_i' = \mathcal{X}(T_i)$. For unbound field types we calculate the result type of a traversal with $C : T$ added to $\mathcal{X}$ using the same function set F. Using the return types, T_i', of field traversals we can approximate the possible functions from F that can be called after traversing an instance of C. The final return type, T, is the common supertype of the *possibleFs* given the field return types. On the last line of our premise, the meta-function *covers*$(\cdot)$ is used to determine whether or not the function set is *complete* with respect to all possible value sequences corresponding to subtypes of the given types. The attributes of *covers* are quite important to the type soundness of our model and deserve a special discussion that follows.

3.4.4 Function Set Coverage

Type checking DemeterF programs infers the return types of traversal expressions, but being sure that function selection always succeeds requires an analysis of function set signatures. In particular, our asymmetric multiple dispatch implemented by *choose* means that after traversing a concrete value, any of the *possible* functions may be called based on the types of subtraversal return values. In general, we cannot know (until runtime) which concrete subtypes will be returned, so we require that all cases be handled by the function set.

In order to guarantee successful dispatch, $covers(\cdot)$ must check all concrete subtypes of the possible argument types and ensure that a *possible* function exists. Because our type hierarchies and function signatures can be arranged into trees (or at least *directed acyclic graphs*), we call the problem *leaf-covering*. The solution involves the Cartesian product of the sequence of type hierarchies, which will be discussed thoroughly in chapter 4 (Section 4.3).

The actual implementation of *covers* is not important to our soundness, we only require the specification that each concrete sequence of subtypes has a *possible* function:

$$\begin{array}{l} \mathit{covers}(F,\ (T_0\ T_1\ \ldots\ T_n)) \ \Leftrightarrow \\ \qquad \forall\, C_0,\ C_1,\ \ldots,\ C_n \text{ with } C_i \leq T_i\ .\ \mathit{possibleFs}(F,\ (C_0\ C_1\ \ldots\ C_n)) \neq () \end{array}$$

As a consequence, *covers* is preserved by subtyping. If $\forall i \in 1..n\,.\,T'_i \leq T_i$, then:

$$\mathit{covers}(F,\ (T_0\ T_1\ \ldots\ T_n)) \ \Rightarrow\ \mathit{covers}(F,\ (T'_0\ T'_1\ \ldots\ T'_n))$$

Because runtime values are made only of concrete types, *e.g.*, `(Neg (True))`, then function selection cannot fail as long as sub-traversals (at runtime) return subtypes of their expected types. Different implementations of *covers* will be examined in chapter 4, and the abstract problem of leaf-covering is *coNP-complete*. However, in practice the number of function arguments (*i.e.*, structure fields) tend to be small, and individual type hierarchies are usually tractable. In our DemeterF implementation the largest number of arguments is 13. With approximately 90 classes in all, the deepest subtype chain is 4 classes, *i.e.*, $C \leq A_1 \leq A_2 \leq A_3$.

3.4.5 Typing Example

Returning to our model example from Listing 3.2, we can assign a type to the body of our program using the T-TRAV rule. The first argument to `traverse` is given the type `Or` by successive applications of T-NEW. Since `True` and `False` have no fields, their constructions become axioms for the derivation. The second part of T-TRAV requires the use of our traversal judgment:

$$\emptyset; \emptyset \vdash_{\mathcal{T}} \langle\, \texttt{Or}, F \rangle : T$$

From the definitions in Listing 3.1, `Or` is a `concrete` type, so a derivation requires the use of T-CTRAV:

$$\frac{\begin{array}{c} \texttt{concrete Or = BExp * BExp .} \in P \qquad \emptyset; (\emptyset,\ \texttt{Or} : T_{\texttt{or}}) \vdash_{\mathcal{T}} \langle\, \texttt{BExp}, F \rangle : T_{\texttt{bexp}} \\ \text{for } f \in \mathit{possibleFs}(F,\ (\texttt{Or}\ T_{\texttt{bexp}}\ T_{\texttt{bexp}})) \qquad \emptyset \vdash_F f : T_f \qquad T_f \leq T_{\texttt{or}} \\ \mathit{covers}(F,\ (\texttt{Or}\ T_{\texttt{bexp}}\ T_{\texttt{bexp}})) \end{array}}{\emptyset; \emptyset \vdash_{\mathcal{T}} \langle \texttt{Or}, F \rangle : T_{\texttt{or}}}$$

The traversal type derivation recursively continues to the abstract types `BExp` and `Lit`, eventually coming to the applications of T-CTRAV for `True` and `False` that do not require recursion. For these types there is only one *possible* function, which simplifies the rule further. An instance for the type `True` is shown below.

$$\frac{\begin{array}{c} \texttt{concrete True = .} \in P \\ \emptyset \vdash_F \texttt{(Lit l)\{ return l; \}} : \texttt{Lit} \qquad \texttt{Lit} \leq T_{\texttt{true}} \\ \mathit{covers}(F,\ (\texttt{True})) \end{array}}{\emptyset; \mathcal{X} \vdash_{\mathcal{T}} \langle \texttt{True}, F \rangle : T_{\texttt{true}}}$$

Assigning a type to the single function and checking function set coverage is then trivial. The constraints build up as we come back through the `abstract`

definitions of `Lit` and `BExp`. Ignoring other variants of `BExp` for simplicity, we have the constraints:

$$\texttt{Lit} \leq T_{\texttt{true}} \quad \texttt{Lit} \leq T_{\texttt{false}} \quad T_{\texttt{true}} \leq T_{\texttt{lit}} \quad T_{\texttt{false}} \leq T_{\texttt{lit}} \quad T_{\texttt{lit}} \leq T_{\texttt{bexp}}$$

We can make these true by setting each of the return types to `Lit`. Other `BExp` variants (`Neg`, `And`, and `Or`) are recursive, which causes equality constraints to be generated instead.

3.5 Type Soundness

In order to prove our AP-F model sound, we construct a Wright-Felleisen [72] style proof of type-soundness, by way of *progress* and *preservation*. Our proof ultimately shows that the reduction of a well-typed AP-F program will not get *stuck*, and will result in a value of the expected type. An expression e is considered *stuck* if there does not exist an expression e' such that $e \rightarrow e'$. In particular, an expression is stuck if it is of the form:

$$E[\texttt{dispatch(}\ F\texttt{,}\ v_0\texttt{,}\ v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{)}\ ]$$

and *choose* (Figure 3.5) results in an error:

$$\mathit{choose}(F,\ \mathit{types}(\ v_0\ v_1\ \ldots\ v_n)) = \texttt{error}$$

We note that *choose* returns `error` precisely when:

$$\mathit{possibleFs}(F,\ \mathit{types}(\ v_0\ v_1\ \ldots\ v_n)) = ()$$

Meaning that F does not contain a function applicable to the given arguments.

Our proof begins with a few AP-F specific lemmas (function and traversal specialization) then moves on to more standard soundness lemmas such as substitution and well-typed contexts. In order to prove that reduction preserves the type of a program, it is necessary to start at the dispatch level and work up to expressions. We begin by proving that *possibleFs* applied to a sequence of subtypes returns a *subset* of the functions returned by *possibleFs* applied to supertypes.

Lemma 3.5.1 (Function Specialization)**.** *As a sequence of argument types is specialized through subtyping, the set of possible functions does not increase.*

If $\forall i \in 1..n\ \ T'_i \leq T_i$ *then*
$\mathit{possibleFs}(F,\ (T'_1\ \ldots\ T'_n)) \subseteq \mathit{possibleFs}(F,\ (T_1\ \ldots\ T_n))$

Proof: We argue using induction on the type sequences by case analysis of the definition of *possible* form Figure 3.5, used to filter the functions of F. Consider a single function $f \in F$ with formal argument types, $(T^f_0\ \ldots\ T^f_m)$.

Our lemma depends on a single implication that must hold of *possible*, given our subtype sequence assumption:

$$possible((T_0^f \ \ldots \ T_m^f), (T'_0 \ \ldots \ T'_n)) \ \Rightarrow \ possible((T_0^f \ \ldots \ T_m^f), (T_0 \ \ldots \ T_n))$$

The three base cases of *possible* (Figure 3.5) are simple, so we consider them together. If the first case applies, and then our implication follows immediately, while the two `false` cases are not relevant, since they only stand to decrease the set of selected functions. Proof of the lemma then hinges on showing that our implication holds for the final, inductive case of the definition. In particular, the first component of the conjunction is important. In our case this reduces to:

$$(T_0^f \leq T'_0) \vee (T'_0 \leq T_0^f) \ \Rightarrow \ (T_0^f \leq T_0) \vee (T_0 \leq T_0^f)$$

Which follows from reflexivity and transitivity of a program's subtype relation, $\leq$. Both disjunction components of the implication are immediate:

$$(T_0^f \leq T'_0) \ \Rightarrow \ (T_0^f \leq T_0) \ \text{and} \ (T'_0 \leq T_0^f) \ \Rightarrow \ (T_0 \leq T_0^f)$$

□

In order to complete the dispatch portion of preservation, we must also show that the application of a function set within a well-typed traversal expression preserves the result type, which is the subject of lemma 3.5.2.

Lemma 3.5.2 (Traversal Specialization, or Subtype Traversals Return Subtypes)**.** *As the type of an expression that is the argument of a traversal is refined, the return type of the traversal expression itself remains a subtype of its original type.*

> *For any well-typed traversal of a type T_0 with $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$. The traversal of a type $T'_0 \leq T_0$ satisfies $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T'_0, F \rangle : T'$ for some $T' \leq T$*

Proof: By induction on the traversal type derivation of $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$, we proceed by cases on the last rule of the derivation, which must be one of T-CTRAV or T-ATRAV, from Figure 3.12.

If T-ATRAV applies (`abstract` A `=` $T_0 \mid \ldots \mid T_n$ `.` $\in P$) then the rule requires that a traversal of an immediate subtype of T_0 return a subtype of the final result type, which applies inductively to all transitive subtypes of T_0, including T'_0.

If T-CTRAV applies (`concrete` C `=` T_1 `*` $\ldots$ `*` T_n `.` $\in P$) then T_0 can only have itself as a subtype ($T_0 \equiv T'_0$). Regardless of which function in F is actually applied at runtime, we know by the T-CTRAV derivation that each function returns a subtype, from the premises of the rule.
□

The final lemmas for preservation are value substitution and well-typed contexts. Substitution proves that function application preserves the type of a traversal expression:

Lemma 3.5.3 (Substitution Preserves Type)**.** *Substituting a value of a subtype for a free variable in any expression results in a subtype of the original expression's type.*

> *Suppose* $\Gamma \equiv (\Gamma', x{:}T_x)$. *If* $\Gamma \vdash_e e : T$, $\emptyset \vdash_e v : T'_x$, *with* $T'_x \leq T_x$ *then* $\Gamma' \vdash_e e[v/x] : T'$ *and* $T' \leq T$.

Proof: By induction on the derivation of $(\Gamma, x : T_x) \vdash_e e : T$. Traversal expressions require lemma 3.5.2, and dispatch expressions require lemma 3.5.1. We proceed by cases on the last rule used:

Case T-VAR $e = x'$. If $x' \neq x$ then $x'{:}T \in \Gamma'$ and $\Gamma' \vdash_e x' : T$. If $x' = x$ then $e[v/x] = v$ and $T'_x \leq T_x$ by our assumptions.

Case T-NEW $e = \texttt{new}\ C\ (\ e_1, \ \ldots\ , e_n\ )$ with $T = C$. By the induction hypothesis, for all $i \in 1..n$ $\Gamma \vdash_e e_i[v/x] : T''_i$ for some $T''_i \leq T'_i$ with $T''_i \leq T_i$ by transitivity of $\leq$. So $\Gamma \vdash_e \texttt{new}\ C\ (\ e_1[v/x], \ \ldots\ , e_n[v/x]\ ) : C$.

Case T-TRAV $e = \texttt{traverse}(\ e_0, \ F\ )$. By the induction hypothesis, $\Gamma' \vdash_e e_0[v/x] : T'_0$ for some $T'_0 \leq T_0$. By lemma 3.5.2 the traversal result is $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T'_0, F\rangle : T'$ for some $T' \leq T$, so $\Gamma' \vdash_e \texttt{traverse}(\ e_0[v/x]\ ,\ F[v/x]\ ) : T'$ and $T' \leq T$.

Case T-APPLY $e = \texttt{apply}(\ f, \ v_0, \ v_1, \ \ldots\ , v_n)$ with $f = (T_0\ x_0, \ \ldots\ , T_n\ x_n)\ \{\ \texttt{return}\ e_0; \}$. If $x \in \overline{x_i}$ then substitution has no effect and the result is T. If $x \notin \overline{x_i}$ then by the induction hypothesis, $(\Gamma', \ x_0{:}T_0, \ \ldots\ , \ x_n{:}T_n) \vdash_e e_0[v/x] : T'$ for some $T' \leq T$.

Case T-DISPATCH $e = \texttt{dispatch}(\ F, \ v_0, \ e_1, \ \ldots\ , e_n)$. By the induction hypothesis, for all $i \in 1..n$ $\Gamma \vdash_e e_i[v/x] : T''_i$ and $T''_i \leq T'_i$. By lemma 3.5.1 we know that $possibleFs(F, \ (C\ T''_1\ \ldots\ T''_n)) \subseteq possibleFs(F, \ (C\ T'_1\ \ldots\ T'_n))$, so there exists a type $T' \leq T$ such that for all $f \in possibleFs(F, \ (C\ T''_1\ \ldots\ T''_n))$ $\Gamma \vdash_F f : T_f$ with $T_f \leq T'$. The result is $\Gamma \vdash_e \texttt{dispatch}(\ F[v/x], \ v_0, \ e_1[v/x], \ \ldots\ , \ e_n[v/x]\ ) : T'$. By the implication property of *covers*:

$$covers(F, \ (C\ T'_1\ \ldots\ T'_n)) \ \Rightarrow \ covers(F, \ (C\ T''_1\ \ldots\ T''_n))$$

So our *covers* premise still holds.

Cases of substitution within functions/sets follow directly from our induction hypothesis.
□

Well-typed contexts shows that recomposition of an expression and a context also preserves the type of the outer context. The lemma and proof are similar to substitution.

Lemma 3.5.4 (Well-Typed Contexts)**.** *Substituting a closed, well-typed expression, which is a subtype of the original, into the hole of a context preserves the outer context's type.*

> *For any closed expressions* e, e', *and context* E, *if* $\emptyset \vdash_e e : T$, $\emptyset \vdash_e e' : T'$ *with* $T' \leq T$, *and* $\Gamma \vdash_e E[e] : T_0$, *then* $\Gamma \vdash_e E[e'] : T'_0$ *for some* $T'_0 \leq T_0$.

Proof: By induction on the structure of the outermost context E and the typing derivation of $E[e]$.

Case $E = [\,]$. Follows from our assumptions, since $\emptyset \vdash_e e : T$, $\emptyset \vdash_e e' : T'$ and $T' \leq T$.

Case $E = \texttt{new}\ C\ (\ v\ \ldots, E',\ e_i\ \ldots\ )$. By the induction hypothesis, replacing e with e' in E' maintains the premises of T-NEW. The result type remains C.

Case $E = \texttt{traverse(}\ E'\ ,\ F\ \texttt{)}$. In T-TRAV, by the induction hypothesis and lemma 3.5.2, the traversal of $E'[e']$ with the same function set, F, must return a subtype of the traversal result type of $E'[e]$.

Case $E = \texttt{dispatch(}\ F\texttt{,}\ v_0\texttt{,}\ v\ \ldots\texttt{,}\ E'\texttt{,}\ e_i\ \ldots\ \texttt{)}$. In T-DISPATCH, by the induction hypothesis and lemma 3.5.1, the *possible* functions with $E'[e']$ instead of $E'[e]$ remains a subset, and must unify to a common supertype, which is a subtype of that obtained with $E'[e]$. The premise of *covers* also holds, with proof similar to substitution.

□

We can now state the first half of our soundness theorem: *preservation*.

Theorem 1 (Preservation)**.** *Reduction (i.e., contraction) preserves an expression's type.*

> *If* $\Gamma \vdash_e E[e] : T$ *and* $E[e] \rightarrow E[e']$ *then* $\Gamma \vdash_e E[e'] : T'$ *with* $T' \leq T$.

Proof: Using lemma 3.5.4, our proof reduces to showing that our individual reductions preserve type. That is, we must show that $\emptyset \vdash_e e : T_e$ and $e \rightarrow e'$ *implies* $\emptyset \vdash_e e' : T'_e$ and $T'_e \leq T_e$. If we prove this implication, then by lemma 3.5.4 it is true that $\Gamma \vdash_e E[e'] : T'$ for some $T' \leq T$.

We proceed by showing the implication holds for each of our reduction rules.

Case If R-APPLY applies . Follows from substitution, lemma 3.5.3.

Case If R-DISPATCH applies . Since the function selected, f, is one of the *possible* functions ($choose(F, (T_0 \ldots T_n)) \in possibleFs(F, (T_0 \ldots T_n))$), f is used in the premise of our typing rule (T-DISPATCH). Proof follows immediately, as the rule requires that the return types of all *possible* functions be a subtype of the assigned type.

Case If R-TRAV applies . The typing derivation of the traversal expression includes both a sub-derivation for the value to be traversed, $e_0 =$ `new` C `(` v_1 `, ...,` v_n `)`, and a traversal judgment based on the definition of C. By the first sub-derivation, we know that $\emptyset \vdash_e v_i : C_i$ for some $C_i \leq T_i$ where T_i is from the definition of C. The traversal typing for each field type, T_i, contains as a sub-derivation a typing rule for C_i, which can be used to construct a traversal derivation for the expanded `traverse` term.

By lemma 3.5.1 the *possible* functions to be used in the typing derivation of the dispatch expression are a subset of those used in the traversal rule for C, and likewise unify to a common supertype (T'_e), which is a subtype of the original, T_e. The use of *covers* in the traversal rule (T-CTRAV) for C remains the same for `dispatch`.

□

While preservation itself is interesting, as important is the preservation of function set *completeness*: if a traversal expression is well typed, then *covers* holds after traversal reduction, R-TRAV.

Corollary 1 (Preservation of *covers*)**.** *The reduction of a well-typed* `traverse` *expression to a* `dispatch` *expression maintains the predicate "covers".*

> *If an expression* $e =$ `traverse(`v_0 `,` F `)` *such that* $\emptyset \vdash_e e : T$ *reduces to* $e' =$ `dispatch(`F `,` v_0 `,` e_1 `, ...,` e_n`)`*, then covers holds for the reduced expression.*

The result of the corollary is that throughout (recursive) traversal reductions *covers* is preserved, so it necessarily holds when function selection is made, and a `dispatch` expression is contracted to `apply`.

Soundness rests on progress, which in turn relies on function selection succeeding. While preservation says that our *possible* functions return the right types, progress requires that there exists a possible function for well-typed traversals.

Theorem 2 (Progress)**.** *A closed, well-typed expression is either a value, or can be reduced, i.e., is never stuck.*

> *For any expression* e *such that* $\emptyset \vdash_e e : T$*, then either* e *is a value, or* $e = E[e']$ *and* $E[e'] \rightarrow E[e'']$.

Proof: By induction on the structure e.

Case $e = x$. This case is impossible since e is closed.

Case $e = \texttt{new}\ C\ (\ e_1, \ \ldots\ , e_n\ )$. If all e_i are values, then e is also a value. Otherwise, by the induction hypothesis, we can decompose e into $E[\,e'\,]$ with $E = \texttt{new}\ C\ (\ v\ \ldots\ , E', e_i\ \ldots\ )$, for for the first non-value and some E', and e' can be reduced.

Case $e = \texttt{traverse(}\ e_0, \ F\ )$. If e_0 is a value, then R-TRAV applies. Otherwise, by the induction hypothesis we can decompose e into $E[\,e'\,]$ with $E = \texttt{traverse(}\ E', \ F\ )$, for some E', and e' can be reduced.

Case $e = \texttt{dispatch(}\ F, \ v_0, \ e_1, \ \ldots\ , e_n)$. If not all e_i are values, then by the induction hypothesis we can decompose e into $E[\,e'\,]$ with $E = \texttt{dispatch(}\ F, \ v_0, \ v\ \ldots\ , E', \ e_i\ \ldots\ )$, for some E', and e' can be reduced.

If all e_i are values, then R-DISPATCH applies. Because e is well-typed, it must be the case that $\emptyset \vdash_e v_0 : C_0$ and for all $i \in 1..n\ \emptyset \vdash_e e_i : C_i$. Our premises require that $covers(F, (C_0\ C_1\ \ldots\ C_n))$, which matches our necessary property of *covers*: $possibleFs(F, (C_0\ C_1\ \ldots\ C_n)) \neq ()$.

Case $e = \texttt{apply(}\ f\ , \ v_0, \ v_1, \ \ldots\ , v_n)$.
With $f = \texttt{(}T_0\ x_0, \ \ldots\ , \ T_n\ x_n\texttt{)}\ \{\ \texttt{return}\ \ e_0\texttt{;}\ \}$. R-APPLY is immediately applicable.

□

With preservation and progress we can now state and prove our soundness theorem.

Theorem 3 (Type Soundness)**.** *A closed, well-typed expression e is either a value, or can be reduced to another well-typed expression.*

> *For any expression e such that $\emptyset \vdash_e e : T$, then e is either a value of type T, or $e \rightarrow e'$ and $\emptyset \vdash_e e' : T'$, with $T' \leq T$.*

Proof: By PROGRESS, e is either a value or can be reduced. By PRESERVATION, if e reduces to e', then $\emptyset \vdash_e e' : T'$ and $T' \leq T$.
□

Wright and Felleisen [72] refer to this theorem as *strong* soundness, since reduction is never stuck and the type of the result is correctly predicted. The standard form of type soundness is what they call *weak* soundness:

> For any well-typed expression, e, if $e \rightarrow e'$, then e' is not *stuck*.

Proof is immediate from Theorem 3, since a stuck `dispatch` expression is not a value.

CHAPTER 4
Algorithms

The safety and performance of DemeterF-based programs rely on the implementation of a number of algorithms including method dispatch, method coverage, and inlining. In this chapter we discuss DemeterF related algorithmic problems, their implementations, and running times.

4.1 Concepts and Notation

We begin with some useful background concepts and notation. In programming languages with inheritance and subtyping one often deals with models and meta-information representing type hierarchies. In DemeterF we are primarily concerned with single inheritance (*i.e.*, C# and Java), resulting in a *tree* (a restricted graph) of types where the parent/child relationships represent both inheritance and subtyping. Two typical examples of type hierarchies are lisp-style `cons` lists and simple numerical expressions. Java class definitions and their visual tree representations are shown in Figures 4.1 and 4.2.

```
abstract class { ... }
class Cons extends List{ ... }
class Empty extends List{ ... }
```

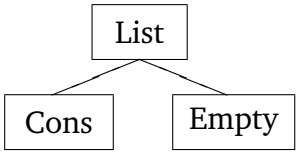


Figure 4.1: List classes and hierarchy tree

We use this more abstract, tree representation for class hierarchies in order to discuss algorithms related to multi-methods, which is particularly useful when discussing features of method selection, coverage, and static dispatch. In the rest of this section we introduce a more formal notion of trees, argument signatures, and graph Cartesian products that will be used in describing our algorithms.

```
abstract class Exp{ ... }
class Int extends Exp{ ... }
class Bin extends Exp{ ... }
class Add extends Bin{ ... }
class Sub extends Bin{ ... }
```

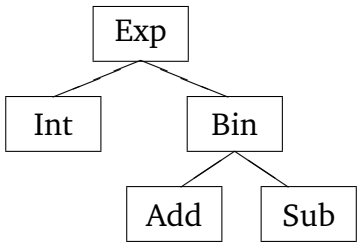


Figure 4.2: Arithmetic expression classes and hierarchy tree

4.1.1 Trees

A *tree*, $T = (\Sigma, \prec)$, is defined over an alphabet of symbols, Σ. Edges of the tree are defined by an immediate successor relation, $\prec \subseteq (\Sigma \times \Sigma)$. For two symbols $a, b \in \Sigma$, an edge exists from a to b when $b \prec a$. For simplicity we restrict the successor relation to be *injective*, modeling single inheritance. This kind of tree can also be viewed as a directed acyclic graph (DAG) where each non-root node has a unique immediate predecessor.

We use less-than, $<$, to denote the transitive closure of the immediate successor relation:

$$\forall a, b \in \Sigma \,.\, b < a \;\equiv\; b \prec a \;\vee\; \exists c \in \Sigma \,.\, b < c \;\wedge\; c < a$$

The reflexive, transitive closure of $\prec$ is denoted by less-than-or-equal, $\leq$.

Given a tree $T = (\Sigma, \prec)$, we define the function `leaves`, to return the nodes in a tree without predecessors:

$$\texttt{leaves}(T) \;\equiv\; \{\, a \in \Sigma \mid \neg \exists b \in \Sigma \,.\, b \prec a \,\}$$

And the function `succs` that returns the immediate successors of a given node in a tree:

$$\texttt{succs}(T, b) \;\equiv\; \{\, a \in \Sigma \mid b \prec a \,\}$$

The `leaves` of a tree represent the *concrete* classes in a hierarchy, and the result of `succs` represents a type's immediate subclasses.

When writing examples we will use a type/symbol, *e.g.*, `Bin`, to refer to either the symbol `Bin` or the tree with `Bin` as its root, though the meaning will be clear from context. For example, using the tree `Exp` from Figure 4.2 we get the following results:

$$\begin{aligned}
&\texttt{leaves}(\texttt{Exp}) = \{\, \texttt{Int}, \texttt{Add}, \texttt{Sub} \,\} \\
&\texttt{succs}(\texttt{Exp}, \texttt{Exp}) = \{\, \texttt{Int}, \texttt{Bin} \,\} \\
&\texttt{succs}(\texttt{Exp}, \texttt{Bin}) = \{\, \texttt{Add}, \texttt{Sub} \,\}
\end{aligned}$$

4.1.2 Signatures

To represent a method's formal and actual argument types we define a *signature* as a sequence of symbols. For simplicity we will use both vector (over-arrow) and sequence notations to denote signatures, depending on context, *e.g.*, $\vec{s} = (s_1, \ldots, s_n)$. Given a sequence of trees, $(T_1, \ldots, T_n)$, with each $T_i = (\Sigma_i, \prec)$, a signature is defined as an element of $(\Sigma_1 \times \cdots \times \Sigma_n)$. For example, using the trees from Figures 4.1 and 4.2, the signature $(\texttt{Add}, \texttt{Cons})$ could represent the formal parameter types of the method:

```
int combine(Add a, Cons c)
```

Or the types of actual arguments in the method call:

```
f.combine(new Add(...), new Cons(...))
```

For specific algorithms we will need to update/replace a specific element within a signature:

$$\texttt{update}(\vec{s},\ i,\ a) \ \equiv\ (s_1, \ldots, s_{i-1}, a, s_{i+1}, \ldots, s_n)$$

Given a sequence $\vec{s}$, an integer i, and a symbol a, the function `update` returns a new sequence with the i^{th} component of $\vec{s}$ replaced by a. The symbol a is assumed to be from the same tree as s_i: $s_i, a \in \Sigma_i$.

Sets of signatures will be used to model the argument types of methods in DemeterF function-classes. Given a sequence of trees, $()T_i, \ldots, T_n)$, we extend the immediate successor relation, $\prec_i$, from symbols to signatures to define two different comparisons: *symmetric* ($\leq$) and *asymmetric* ($\sqsubset$). The first models method and argument applicability and the second models method selection and preference/ordering.

Symmetric Comparison Applying a method to arguments requires that the types of the arguments be subtypes of the method's formal parameter types. In comparing two signatures, each parameter is given equal, or *symmetric*, treatment. We call this relation *applicable*, and write it as $\leq$. Similar to symbol/tree relations, we begin by defining an immediate successor relation, $\prec$, on signatures using the successor relations, $\prec_i$, from our n trees:

$$\vec{c} \prec \vec{a} \ \equiv\ \exists i\,.\ c_i \prec_i a_i \ \wedge\ \forall j \in [1..n]\,.\ j \neq i \implies c_j = a_j$$

Two signatures are related by $\prec$ when they differ only by their i^{th} element, and the corresponding elements are related in the i^{th} tree by $\prec_i$. We will use $<$ for the transitive closure of $\prec$ over signatures, with symmetric comparison defined as the reflexive, transitive closure of the immediate successor relation:

$$\vec{c} \leq \vec{a} \ \equiv\ \vec{c} = \vec{a} \ \vee\ \vec{c} < \vec{a}$$

We say that a signature $\vec{a}$ is applicable to a signature $\vec{c}$ if $\vec{c} \leq \vec{a}$, where $\vec{a}$ and $\vec{c}$ represent a method's formal and actual argument types respectively. This gives us a notion of methods that can be applied to a sequence of arguments, but it does not provide a total ordering. Ambiguities arise when two signatures are both applicable to the same signature, but neither is applicable to the other.

For example, consider the signatures, $(\texttt{Int}, \texttt{Int})$, $(\texttt{Exp}, \texttt{Int})$, and $(\texttt{Exp}, \texttt{Int})$. Given our tree in Figure 4.2 the following, are true:

$$(\texttt{Int}, \texttt{Int}) < (\texttt{Int}, \texttt{Exp})$$
$$(\texttt{Int}, \texttt{Int}) < (\texttt{Exp}, \texttt{Int})$$

With actual argument types of $(\texttt{Int}, \texttt{Int})$, the signatures $(\texttt{Exp}, \texttt{Int})$ and $(\texttt{Int}, \texttt{Exp})$ are both applicable, but neither is applicable to the other.[1]

Asymmetric Comparison To avoid these ambiguities when a signature representing runtime argument types has multiple applicable method signatures, we define a total ordering that provides a notion of *more specific*, which we write as $\sqsubset$:

$$\vec{a} \sqsubset \vec{s} \equiv \exists i \in [1..n] \,.\, (a_i < s_i) \wedge \forall k < i \,.\, a_k = s_k$$

The resulting relation is similar to lexicographic ordering on strings: the first element that differs defines the ordering.

Returning to our previous example, when deciding between the two applicable signatures $(\texttt{Exp}, \texttt{Int})$ and $(\texttt{Int}, \texttt{Exp})$, our asymmetric relation orders them as follows:

$$(\texttt{Int}, \texttt{Exp}) \sqsubset (\texttt{Exp}, \texttt{Int})$$

A method signature $(\texttt{Int}, \texttt{Exp})$ will be chosen over $(\texttt{Exp}, \texttt{Int})$ when both are applicable, *i.e.*, when the actual argument types are $(\texttt{Int}, \texttt{Int})$.

We use this relation to model the multiple-dispatch selection that DemeterF uses to eliminate ambiguities and corresponding errors in traversals and function-objects.

4.1.3 Graph Cartesian Products

To help visualize relations over signatures and trees we will use a *graph Cartesian product* (GCP). A GCP, $G = (V, E)$, is defined over a sequence of trees, $(T_1, \ldots, T_n)$, with each $T_i = (\Sigma_i, \prec_i)$. The vertices, V, of the graph are signatures and the edges, E, are defined by the immediate successor relation on signatures:

[1] In most statically typed multiple-dispatch systems, *e.g.*, MultiJava [22] and Fortress [8], this results in a compile-time error.

$$V = \Sigma_1 \times \cdots \times \Sigma_n$$
$$E = \{\, (\vec{a}, \vec{c}) \in V \times V \;\mid\; \vec{c} \prec \vec{a} \,\}$$

For example, the GCP of the two earlier trees of expressions and cons-lists is shown in Figure 4.3. The *root* of the GCP is the signature `(Exp, List)`.

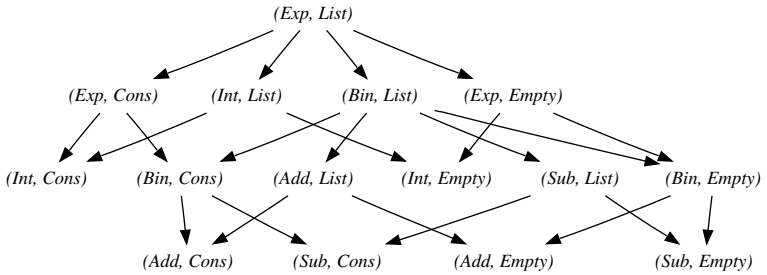


Figure 4.3: Graph Cartesian product rooted at `(Exp, List)`.

Given our definition of a GCP, it can always be characterized as a directed acyclic graph (DAG). Reachability in a GCP is defined by the *applicable* relation on signatures, $\leq$, and the `leaves` of a GCP correspond to signatures made entirely of leaves of the corresponding trees:

$$\texttt{leaves}(G) \;\equiv\; \texttt{leaves}(T_1) \times \cdots \times \texttt{leaves}(T_n)$$

The leaves of the trees correspond to concrete classes, and the leaves of the GCP correspond to concrete signatures that represent the possible runtime types of arguments passed to methods.

4.1.4 Algorithm Notation

We will present algorithms in a notation similar to the functional programming language Haskell [38], with a C/Java style calling syntax where the opening parenthesis are placed to the right of the function name. Our algorithms do not rely on any particular properties of an implementation (*e.g.*, lazy versus strict), just an intuition of its semantics and simple pattern matching forms.

Function Definitions Function definitions will consist of a type signature, followed by a list of equations. As an example, Figure 4.4 shows a function that recursively calculates the i^{th} Fibonacci number.
The function, `fib`, is declared with the type `(Int → Int)`: it takes a single integer and returns an integer. The function is defined by three equations that match argument cases. 0 and 1 used in the argument position of the equations are *patterns* that match corresponding integer literals. The pattern i matches any other integer and binds it to i in the right-hand side of

```
fib :: Int → Int
fib(0) = 0
fib(1) = 1
fib(i) = fib(i-1) + fib(i-2)
```

Figure 4.4: Notation Example: Fibonacci.

the equation, which proceeds by adding the results of two recursive calls. Variable bindings and uses will be typeset in italics so they are easily distinguishable.

Data Structures We assume datatypes for representing symbols (`sym`), trees (`tree`), and signatures (`sig`). Overloaded versions of previously defined relations and functions, *e.g.*, $\prec$, $\sqsubset$, `succs`, *etc.*, will be used in the right-hand side of equations when needed.

We use Haskell's list notation for type signatures, *e.g.*, `[Int]` is the type of integer lists, and within function definitions: the empty list is both a pattern and a value, denoted by empty square brackets, `[]`, and a non-empty list with a head of `f` and a tail of `R` will be denoted in both patterns and expressions by `(f:R)`.

When necessary we will define custom data structures using an intuitive notation for algebraic datatypes similar to Haskell and ML. Figure 4.5 shows an example data structure representing integer binary search trees, `IntBST`s, and functions for inserting an integer into a `IntBST`, and collecting a list of a `IntBST`'s elements. For simplicity we will refrain from polymorphic user-defined data structures.

```
data IntBST = IntNode(Int, IntBST, IntBST)
            | IntLeaf()

insert :: IntBST → Int → IntBST
insert(IntLeaf(), i) = IntNode(i, IntLeaf(), IntLeaf())
insert(IntNode(d, left, right), i) =
            if i ≤ d then IntNode(d, insert(left, i), right)
                     else IntNode(d, left, insert(right, i))

elements :: IntBST → [Int]
elements(IntLeaf(), i) = []
elements(IntNode(d, left, right), i) =
            append(elements(left), (d : elements(right)))
```

Figure 4.5: Notation Example: `IntBST` insertion and elements as a list.

The type `IntBST` is defined by a **`data`** definition with two value constructors, `IntNode` and `IntLeaf`, separated by a bar (|). In general any number of constructors can be defined. The constructor `IntNode` accepts three arguments, an `Int` and two `BST`s, while `IntLeaf` accepts no arguments. Defined

constructors are used as patterns in argument positions and as expressions. The variables within patterns, *e.g.*, i and *left*, are bound in the right-hand side of the equation to matching components of the structure. In the definition of `insert` we make use of an `if`-expression that decides between the recursive insertion into the *left* or *right* subtrees, constructing a new `IntNode` in both cases. The definition of `elements` uses a helper function, `append`, and Haskell's infix cons-list syntax, $(d{:}\cdots)$. If necessary we will provide definitions for more complicated helper functions along with the algorithm(s).

With notation and background in place, we now discuss particular algorithmic problems used in the implementation of DemeterF. Each section in the remainder of this chapter will give a brief background of a problem, a concise description, an algorithmic solution, and one or more implementations including a discussion of running times.

4.2 Method Selection and Dispatch

In DemeterF, the selection of function-object methods during traversal chooses the *most specific* signature based on the runtime types of its arguments. When there is only a single applicable method, this decision can certainly be made statically. If two or more are applicable to similar traversal results then at least some of the method selection decision must be deferred to runtime. This section discusses our selection algorithms for reflective and statically computed method dispatch.

4.2.1 Reflective Selection

Before applying a `combine` method during traversal, the types of actual method arguments are known. The method signatures of the function-object used can be inspected to determine the most specific method that is *applicable* to the actual argument types.

4.2.1.1 The Problem

We describe the DemeterF runtime method selection problem is follows:

> Given a non-empty signature, $\vec{c} = (c_0, c_1, \ldots, c_n)$, an implicit sequence of trees $(T_0, T_1, \ldots, T_n)$ such that $c_i \in \texttt{leaves}(T_i)$, and a set of signatures, $S = \{\, \vec{s}_1, \ldots, \vec{s}_m \,\}$, compute the most specific signature, $\vec{s}_i$, that is applicable to $\vec{c}$:

$$\texttt{select}(\vec{c}, S) \;\equiv\; \vec{a} \in S \,.\, \vec{c} \leq \vec{a} \;\wedge\; \forall \vec{s} \in S \,.\, \vec{a} = \vec{s} \;\vee\; \vec{a} \sqsubset \vec{s}$$

The set of signatures, S, represents the formal argument types of a function-object's `combine` methods. The signature $\vec{c}$ represents the types of runtime

arguments, with c_0 being the type of the object that was traversed and $c_1, \ldots, c_n$ being the result types of the recursive traversal of the original object's fields.

4.2.1.2 Solution

The definition of the problem admits a direct algorithm shown in Figure 4.6. We use this implementation as the definition of function-object dispatch: selecting the most specific applicable signature given runtime argument types.

```
select :: sig → [sig] → sig
select(c⃗, []) = error
select(c⃗, (s⃗:S)) = if (c⃗ ≤ s⃗)
                    then best(s⃗, c⃗, S)
                    else select(c⃗, S)

best :: sig → sig → [sig] → sig
best(a⃗, c⃗, []) = a⃗
best(a⃗, c⃗, (s⃗:S)) = if (c⃗ ≤ s⃗ ∧ s⃗ ⊏ a⃗)
                    then best(s⃗, c⃗, S)
                    else best(a⃗, c⃗, S)
```

Figure 4.6: Reflective Selection Algorithm.

Our implementation is split into two functions. The function `select` accepts a signature, $\vec{c}$, and a list of signatures, S, and searches for a signature that is applicable to $\vec{c}$. The first applicable signature is passed to `best`, which finds the most specific signature applicable to $\vec{c}$ starting with the initial guess, $\vec{a}$.

4.2.1.3 Running Time

The comparison of signatures using $\leq$ runs in time $O(t \cdot n)$, where t is a bound on the size of the trees and $n = |\vec{c}|$ is the number of arguments. The running time of `select` is as follows:

$$\texttt{select}(\vec{c},\ S) \in O(\, t \cdot n \cdot |S|\,)$$

The implementation of `select` depends on the number of method signatures in the function-object. Each time a dispatch is required the list must be searched, which can dominate the running time of a data structure traversal. Rather than doing a full search we can reorganize signatures based on their argument types and reduce the number of comparisons that must be made at runtime.

4.2.2 Static Selection and Residue

Inefficiencies in `select` stem from two related issues: (1) the function performs a linear search through the signatures, and (2) it works independently without knowing anything about the context in which a method will be selected. When more information is available about the types of recursive traversals, the number of possibly applicable signatures can be statically reduced and a specialized decision structure can be generated.

4.2.2.1 Related Signatures

Static decisions about signature selection must deal with less information. Given a signature $\vec{c}$ representing the static types of traversal results, it is possible at runtime to select a signature that is applicable to $\vec{c}$, *i.e.*, $\vec{c} \leq \vec{s}$, but selecting a more specific signature, *i.e.*, $\vec{s} \sqsubset \vec{c}$, is also possible.

Since both situations may occur, we use a broader relation, $\bowtie$, to describe signatures that might be selected at runtime. We define $\bowtie$ as follows:

$$\vec{s} \bowtie \vec{c} \;\equiv\; \vec{s} \leq \vec{c} \;\vee\; \vec{c} \leq \vec{s}$$

The signature $\vec{c}$ represents a static approximation of traversal result types and $\vec{s}$ represents a method signature. We call this relation *related*, since it relates two signatures that have components related in the corresponding tree. A signature $\vec{s}$ is related $\vec{c}$ if one is applicable to the other, in either order.

4.2.2.2 Residual Dispatch

With more information about the types of values to which signatures will be applied, the set S can be reduced by filtering out unrelated signatures. Because the remaining signatures are related, we can use argument subtype relationships to construct a decision tree that selects the most specific signature using a minimal number of runtime type tests. In many cases a dynamic decision is not required, when there is only one related signature. When the number of related signatures is greater than 1 we refer to the remaining dispatch decision as *residue*.

In the case of DemeterF, type checking a function-class over a data structure statically provides the approximate types of values returned from subtraversals. This can be used to determine the methods that might be applied at each point in the traversal, *i.e.*, related signatures. We use this information to generate data structure specific traversals with inlined dispatch residue to be executed at runtime. The residue takes the form of a decision tree of argument type tests, which is interpreted at runtime using $\leq$, or `instanceof` in Java.

4.2.2.3 The Problem

We describe the residual dispatch problem as follows:

> Given a non-empty signature, $\vec{c} = (c_0, c_1, \ldots, c_n)$, an implicit sequence of trees $(T_0, T_1, \ldots, T_n)$ such that $c_i \in \Sigma_i$, and a set of signatures, $S = \{ \vec{s}_1, \ldots, \vec{s}_m \}$, compute the residual dispatch tree, D, that determines the most specific signature, $\vec{a}_i$, to be applied to a runtime signature, $\vec{a} \leq \vec{c}$.
>
> The result is decision tree, D, built from two relations, *left* and *right*. Interior nodes of the decision tree are labeled with a pair, (i, t), representing a type test of the i^{th} parameter against the given symbol, t. A node's *left* and *right* children represent sub-decisions for a test result of `true` or `false`, respectively. The leaves of the decision tree are labeled with signatures from S.

The signatures, S, represent the argument signatures of function-class methods, and $\vec{c}$ represents the static types of expected traversals results for method dispatch. Our dispatch tree, D, represents a decision procedure that performs type/instance tests on the return values of subtraversals, and leaves of the tree describe the selected method's signature.

4.2.2.4 Solution

Our solution to the dispatch residue problem is shown in Figure 4.7. A dispatch decision tree, `Dec`, is created by one of two value constructors. `IF` encodes the test of a particular argument position at particular type, and branches to another `Dec` when the test succeeds or fails. `CALL` represents a selected signature, once the necessary number arguments have been inspected.

The function `residue` constructs a `Dec` beginning with the first argument position, 1, given the static signature $\vec{c}$ and a list of signatures S. If all argument types have been tested ($i > |\vec{c}|$), then the helper function `decision` constructs a `CALL` node using `select` to determine most specific signature that is still applicable. Otherwise, we construct the set of symbols A from the i^{th} argument types of related signatures from S, sorted according to $<_i$. From A we construct a list of pairs, P, whose left component is the corresponding element of A (a_i), and right component is a `list` of the previous elements from A. P represents the type to be tested for a given set of methods, and the type tests that will have failed, and so can safely be ignored, reducing useless repetitive tests for unreachable signatures.

Signatures from S are then placed into groups, G, by their i^{th} argument type. Each group consists of a symbol a from a pair in P (in order) and a list of signatures with an i^{th} argument type that is related to a (*i.e.*, $\vec{s} \in S \mid s_i \bowtie a \ldots$), when a is also not in our list of ignored types, *ignr*. The result is a

```
data Dec = IF(Int, sym, Dec, Dec)
         | CALL(sig)

residue :: sig → [sig] → Dec
residue(c⃗, S) = decision(1, c⃗, S)

decision :: Int → sig → [sig] → Dec
decision(i, c⃗, []) = error
decision(i, c⃗, S) =
    if (i > |c⃗|) then CALL(select(c⃗, S))
    else let A = sort({ s_i | s⃗ ∈ S ∧ s⃗ ⋈ c⃗}, <_i)
             P = [(a_k, [a_1, ..., a_{k-1}]) | k ∈ [1..|A|]]
             G = [(a, [s⃗ ∈ S | s_i ⋈ a ∧ s_i ∉ ignr]) | (a, ignr) ∈ A]
         in buildDec(i, c⃗, G)

buildDec :: Int → sig → [(sym, [sig])] → Dec
buildDec(i, c⃗, []) = error
buildDec(i, c⃗, ((a, S):G)) =
    let d = decision(i + 1, update(c⃗, i, a), S)
    in if null(G) then d
       else IF(i, a, d, buildDec(i, c⃗, G))
```

Figure 4.7: Residual Selection Algorithm.

list of pairs with a symbol as their first component and a list of signatures as their second component representing a type test, and the signatures that are still possible if the test should succeed.

In `buildDec` the groupings are used to recursively construct a chain of decisions for the next argument position. If only one grouping exists, *i.e.*, `null(`G`)`, then the decision d is returned. If there are more groupings, an `IF` test for argument i of type a is constructed with d as the true branch, and the rest of the groups decision as a false branch.

4.2.2.5 Running Time

Symbol comparisons, $<$ and $\bowtie$, take time proportional to t, where t is a bound on the size of the trees. Signature comparisons, $\leq$ and $\bowtie$, run in time $O(t \cdot n)$, where n is the length of the signature, *i.e.*, $|\vec{c}|$. The worst-case running time of `residue` is as follows:

$$\texttt{residue}(\vec{c},\ S) \in O\left(b^n \cdot (t \cdot n \cdot |S| + t \cdot log(t))\right)$$

The exponential term, t^n, comes from the recursive call to `decision` from within `buildDec`, since the bound on the tree size, t, is also a bound on the length of G. The average running time of the algorithm depends on the average branching factor of the trees. If we call this factor b, then running time can be more accurately described as:

$$\texttt{residue}(\,\vec{c},\ S\,) \in O\,(\,b^n \cdot (n \cdot |S| + (|S| \cdot log(|S|)))\,)$$

In most cases this term dominates the running time, since the number of signatures, $|S|$, is small and the size of A is typically much smaller than $|S|$. The running time is interesting, but more important is the size of the resulting decision tree. Since the tree will eventually be used to make dynamic selection it should be in some sense minimal. The depth of the `Dec` produced by `residue` is at worst:

$$O(\,t \cdot n\,)$$

Since we make at most t type tests for each of the n arguments, though this also requires an exponential number of methods. This is a great improvement on the runtime performance of `select`, which depends on the number of signatures.

4.3 Method Coverage

Multi-method languages and systems like CLOS[65], MultiJava [22], JPred [55], and DemeterF rely on selecting the most specific function for runtime argument types. DemeterF (like CLOS) uses an *asymmetric* multiple dispatch strategy where the leftmost arguments are given precedence. MultiJava and JPred employ a *symmetric* strategy where all arguments are given equal weight, and method ambiguity is not allowed at runtime. In both dispatch styles it is beneficial for the system to statically ensure that certain dispatch errors are not possible, *e.g.*, *message not understood* errors.

For DemeterF this means checking that a function-class contains an applicable signature for all possible sequences of concrete argument types, corresponding to the leaves of the GCP. For example, if a method group has the signature $(Exp,\ List)$, then it suffices to check that an applicable method exists for each of the concrete combinations:

```
(Int, Cons)  (Add, Cons)  (Sub, Cons)
(Int, Empty) (Add, Empty) (Sub, Empty)
```

We call the task of checking signature coverage the `leaf-covering` problem.

4.3.1 Definition : LEAF-COVERING

> Given a sequence of trees, $(T_1,\ \ldots,\ T_n)$, we say that a set of signatures, S, *covers* the trees if S contains an applicable signature for each signature made of leaves from each T_i:

$$\texttt{covers}(S,\ (T_1,\ \ldots,\ T_n)) \ \equiv$$
$$\forall \vec{\ell} \in (\texttt{leaves}(T_1) \times \cdots \times \texttt{leaves}(T_n))\,.\,\exists \vec{s} \in S\,.\,\vec{\ell} \leq \vec{s}$$

Leaf-covering can also be defined in terms of a GCP: the root signature of our trees, *e.g.*, `(List, Exp)`, becomes the root of our GCP. Leaves in the GCP are defined as the vertices of V with *out-degree* of 0:

$$\texttt{leaves}(G) \;\equiv\; \{\, \vec{v} \in V \mid \forall\, \vec{u} \in V \,.\, (\vec{v}, \vec{u}) \notin E \,\}$$

The leaves of the GCP are the same as the signatures made up of the leaves from each of the trees, T_i. Given a GCP, the task of `covers` is to check that each leaf signature of the GCP has an ancestor in S.

4.3.2 LEAF-COVERING is coNP-Complete

Before describing solutions to leaf-covering, we first show that the problem is actually *coNP-Complete*. The problem is in *coNP*: to show that a set of signatures, S, does not cover all leaves we simply provide a leaf that is not covered by S. The witness can easily be chosen non-deterministically and checked in time $O(\,t \cdot n \cdot |S|)$. Leaf-covering can then be shown to be coNP-Complete by reducing DNF validity, *i.e.*, *tautology* checking, to leaf-covering.[2]

Reducing DNF to LEAF-COVERING Consider a formula, F, in disjunctive normal form, where each clause consists of literals, $l_{i,j}$, which are either the positive or negative assertion of a *variable, e.g.*, a or $\neg a$:

$$F \;\equiv\; (l_{1,1} \wedge \cdots \wedge l_{1,n_1}) \;\vee\; \cdots \;\vee\; (l_{m,1} \wedge \cdots \wedge l_{m,n_m})$$

With an ordering on the variables used in F, *e.g.*, alphabetic, we create a sequence of trees with variable names as roots and the special symbols *true* and *false* as leaves. We then encode the clauses of the formula as signatures of S containing a symbol from each of the trees in order. For each clause we encode a positive literal as *true*, a negative literal as *false*, and an unused variable as the root of its corresponding tree.

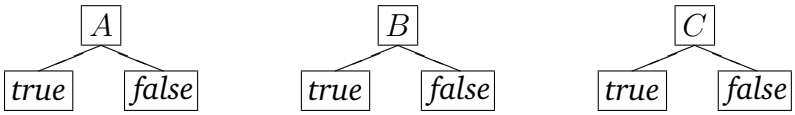
The cross product of the leaves of the trees (or the set of leaves of the GCP) contains all assignments of *true* and *false* to the variables of F. If the elements of S cover all the leaves, then all *concrete* assignments are covered by the clauses of the formula, meaning F is a tautology. If not, then one of the uncovered leaves represents an assignment that does not satisfy the formula.

As a complete example, consider the following formula:

$$F = (a \;\wedge\; \neg b) \;\vee\; (\neg a \;\wedge\; c) \;\vee\; (\neg b \;\wedge\; c) \;\vee\; (\neg a \;\wedge\; \neg c)$$

To convert the validity of this formula into a leaf-covering problem, we order the variables as $(a,\, b,\, c)$ and construct three corresponding trees:

[2]Thanks to Yannis Smaragdakis for suggesting and detailing this reduction.



The root of our GCP is the triple $(A,\ B,\ C)$, and our set S encodes the clauses of F as triples:

$$S = \{\, (true,\ false,\ C),\ (false,\ B,\ true),\\ (A,\ false,\ true),\ (false,\ B,\ false)\,\}$$

The leaf signatures (*i.e.*, leaves of the GCP) include all triple permutations of *true* and *false*. In this case the signatures, S, constructed from the formula answer in this case is *no*; the leaf signatures that are not covered, $(true,\ true,\ true)$ and $(true,\ true,\ false)$. The corresponding assignments to a, b, and c respectively do not satisfy F: *i.e.*, F is not valid.

4.3.3 Solutions

In this section we discuss two different solutions to leaf-covering. The first is a simple brute-force approach that directly implements the specification of the problem. The second is a more involved solution that uses tree intersections and counting.

4.3.3.1 Solution 1: Brute-Force

The definition of `covers` admits a straightforward solution: compute all the possible leaf signatures and check that each leaf, $\vec{\ell}$, has an applicable signature: $\exists \vec{s} \in S\,.\ \vec{\ell} \leq \vec{s}$. The simple brute-force algorithm is show in Figure 4.8. We first create the Cartesian-product of the `leaves` of each of the trees, then

```
covers :: [sig] → [tree] → Bool
covers(S, Ts) =
    let lfs = cross(map(leaves, Ts))
    in all(lfs, S)

all :: [sig] → [sig] → Bool
all([], S) = true
all((ℓ⃗ : lfs), S) = one(S, ℓ⃗) ∧ all(lfs, S)

one :: [sig] → sig → Bool
one([], ℓ⃗) = false
one((s⃗ : S), ℓ⃗) = ℓ⃗ ≤ s⃗ ∨ one(S, ℓ⃗)
```

Figure 4.8: Brute-Force Leaf-Covering Algorithm.

iterate to check that `all` leaf signatures, $\vec{\ell}$, is covered by at least `one` signature in S.

Running Time For a set of signatures, S, and a sequence of trees, $(T_1, \ldots, T_n)$, the algorithm has the following running time:

$$\texttt{covers}(S, (T_1, \ldots, T_n)) \in O\left(|S| \cdot \prod_{i=1}^{n} |\,\texttt{leaves}(T_i)\,|\right) = O(\,|S| \cdot t^n\,)$$

Where t is a bound on the size of the trees. If a leaf signature is without a corresponding applicable signature in S then it can be used as a witness of incomplete coverage.

This solution runs in time exponential in n, *i.e.*, the number of trees, but for a fixed n the running time becomes polynomial. Problems of this type are termed *fixed parameter tractable*, as they are only exponential in part of their input. In fact, this solution for the decision problem is polynomial (in $|S|$) for fixed number of trees.

4.3.3.2 Solution 2: Inclusion-Exclusion

A second solution to leaf-covering involves tree intersection and the inclusion-exclusion principle. Taking a close look at the GCP example in Figure 4.3 shows that multiple interior vertices have edges that reach a the same leaf signature. This overlap of signatures can be used to calculate the *size* of the union of the leaves covered by S, without having to generate the leaf signatures themselves. This is done by calculating the number of overlapping leaves of two or more signatures and using the set inclusion-exclusion principle to calculate the size of their union.

We begin by defining another version of `leaves` for a tree given a starting symbol:

$$\texttt{leaves}(T, a) \;\equiv\; \{\, b \in \texttt{leaves}(T) \mid b \leq a \,\}$$

Which returns the leaves of T that are also successors of a. This function can be used to compute the number of overlapping leaves of a set of signatures, $S = \{\vec{s}^1, \ldots, \vec{s}^{|S|}\}$, by calculating the product of the sizes of the individual (point-wise) intersections:

$$\texttt{overlap}(\, S, (T_1, \ldots, T_n)\,) \;\equiv\; \prod_{i=1}^{n} \left| \bigcap_{k=1}^{|S|} \texttt{leaves}(T_i, s_i^k) \right|$$

The intersection of leaves is calculated independently using the i^{th} tree and the corresponding elements of each signature, s_i^k.

The total number of leaf signatures in a sequence of trees is the product of the number of leaves in the individual trees:

$$\texttt{total}(T_1, \ldots, T_n) \;\equiv\; \prod_{i=1}^{n} |\,\texttt{leaves}(T_i)\,|$$

For two signatures, $\vec{s}$ and $\vec{a}$, determining the number of unique leaf signatures covered can be calculated by adding the total leaves covered by each and subtracting the number of overlapping leaves:

$$\prod_{i=1}^{n} |\,\texttt{leaves}(T_i,\, s_i)\,| + \prod_{i=1}^{n} |\,\texttt{leaves}(T_i,\, s_i)\,| - \texttt{overlap}(\,\{\,\vec{s},\, \vec{a}\,\},\, (T_1,\, \ldots,\, T_n)\,)$$

If this result is the same as the `total` number of leaves then the trees are are fully covered by the two vertices, $\vec{s}$ and $\vec{a}$. If not, then an uncovered leaf signature must exist.

This provides a way to calculate the size of the union of the covered leaves directly from the number of leaves at the intersection of the signatures. For an arbitrary set of signatures, S, the set inclusion-exclusion principle is used to calculate the size of the union of all covered leaves using `overlap`:

$$\begin{aligned}&\texttt{inclu\_exclu}(\,S,\, (T_1,\, \ldots,\, T_n)\,)\\ &\qquad \equiv\ \textstyle\sum_{S \supseteq M \neq \emptyset} \left[(-1)^{|M|-1}\, \texttt{overlap}(M,\, (T_1,\, \ldots,\, T_n)\,)\right]\end{aligned}$$

The complete implementation of $\texttt{covers}_{inex}$ compares the `total` number of leaves to the number of covered leaf signatures calculated by `inclu_exclu`:

$$\begin{aligned}&\texttt{covers}_{inex}(\,S,\, (T_1,\, \ldots,\, T_n)\,)\\ &\qquad \equiv\ \texttt{total}(T_1,\, \ldots,\, T_n) = \texttt{inclu\_exclu}(\,S,\, (T_1,\, \ldots,\, T_n)\,)\end{aligned}$$

The benefit of this second implementation is that the number of leaves covered by S is calculated over the individual trees, eliminating the need to inspect or construct the leaf signatures.

Running Time The running time of `inclu_exclu` relies heavily on `overlap`, which has the following running time:

$$\texttt{overlap}(\,S,\, (T_1,\, \ldots,\, T_n)\,) \in O(\,t \cdot n \cdot |S|\,)$$

The calculation of the number of overlapping leaves for a set of signatures is efficient, since individual symbols are compared over a single tree.

The inclusion-exclusion procedure itself runs in time that is exponential in the size of S, rather than the number of trees:

$$\texttt{inclu\_exclu}(\,S,\, (T_1, \ldots,\, T_n)\,) \in O\left(t \cdot n \cdot \sum_{k=1}^{|S|} \binom{|S|}{k}\right) = O(\,t \cdot n \cdot 2^{|S|})$$

Where t is again a bound on the size of each tree. The exponential factor, $2^{|S|}$, is in contrast to the brute-force solution, where running time depends on the

exponential factor t^n. Determining the best solution depends on the number of signatures versus the number trees (or the length of the signatures), which correspond to the number of methods and the number of method arguments respectively.

4.3.4 Fixed Parameter Tractability

The running times of the two solutions to leaf-covering we have presented, *i.e.*, `covers` and $\texttt{covers}_{inex}$, are not exponential in all of their inputs. Both algorithms become polynomial when part of their input is of a fixed size. If the trees, $(T_1, \ldots, T_n)$, are fixed then t^n is bounded by a constant, K_n. The brute-force algorithm's running time becomes:

$$\texttt{covers}(S, (T_1, \ldots, T_n)) \in O(\,|S| \cdot K_n\,)$$

If instead the signatures, S, are fixed, then $2^{|S|}$ is bounded by a constant, K_S and the running time of `inclu_exclu` becomes:

$$\texttt{inclu\_exclu}(\,S, (T_1, \ldots, T_n)\,) \in O(\,t \cdot n \cdot K_S\,)$$

In the case of DemeterF the inclusion-exclusion solution is more attractive since the set of signatures, S, is fixed while checking a traversal.

4.3.5 Decision Versus Search

The brute-force solution to leaf-covering answers both the decision problem and the search (or *function*) problem. While we check each leaf signature, if the leaf is uncovered then we immediately have a witness.

While the inclusion-exclusion solution provides an answer to the decision problem, it does not immediately produce an uncovered leaf signature. There is a standard, well-known sequence of reductions for NP-complete problems that converts a decision solution into a search solution to decision for [63]. We are also aware of the work of Bellare and Goldwasser [11], which provides a proof and a general algorithm showing that for all NP-complete problems, search reduces to decision ([10], Theorem 4.5). However, in this section we discuss a more efficient alternative to the standard reduction that uses the `inclu_exclu` implementation to determine an uncovered leaf signature.

In order to find an uncovered leaf signature, we consider edges of the GCP, represented by our successor relation, $\prec$. Clearly the signature, $\vec{r}$, made only of the roots of the trees must cover all leaves:

$$\vec{r} = (r_1, \ldots, r_n) \text{ where } r_i \in \Sigma_i \ \wedge\ \neg\exists a \in \Sigma_i \,.\, r_i \prec_i a$$

To find an uncovered leaf, $\vec{\ell}$, if it exists, we exploit the fact that S covers fewer leaves than if S included the signature $\vec{\ell}$. In fact, if $\vec{\ell}$ is uncovered,

then any predecessor of $\vec{\ell}$ can be combined with S to cover more leaves than just S:

$$\begin{aligned}\neg\texttt{covers}(S, (T_1, \ldots, T_n)) &\implies \forall \vec{s}\,.\,\vec{\ell} \leq \vec{s} \\ &\implies \texttt{inclu\_exclu}(S, (T_1, \ldots, T_n)) \\ &\quad < \texttt{inclu\_exclu}((S \cup \{\vec{s}\}), (T_1, \ldots, T_n))\end{aligned}$$

This presents us with an algorithm that uses the immediate successor relationship to explore signatures that will increase the coverage of S until we reach a leaf. Figure 4.9 shows our algorithm that searches for an uncovered leaf using `inclu_exclu`. We begin by calculating the number of leaves that

```
uncoveredSig :: [tree] → [sig] → sig
uncoveredSig(Ts, S) =
    let sCov = inclu_exclu(S, Ts)
    in down(sCov, roots(Ts), S, Ts)

down :: Int → sig → [sig] → [tree] → sig
down(mCov, ℓ⃗, S, Ts) =
    let ss = succs(ℓ⃗, Ts)
    in if null(ss) then ℓ⃗
       else across(sCov, ss, S, Ts)

across :: Int → [sig] → [sig] → [tree] → sig
across(mCov, [], S, Ts) = error "No Uncovered Leaf"
across(mCov, (ℓ⃗ : ss), S, Ts) =
    let cov = inclu_exclu((ℓ⃗ : S), Ts)
    in if cov > sCov then down(sCov, ℓ⃗, S, Ts)
       else across(sCov, ss, S, Ts)
```

Figure 4.9: Search for an uncovered leaf using `inclu_exclu`

S covers. We then start with the signature made up of the roots of our trees, and move `down` and `across`, in analogy to the GCP. The function `down` steps *down* the GCP to select an uncovered signature from the successors, *ss*, of $\vec{\ell}$. If the signature has no successors, `null(`*ss*`)`, then it is an uncovered leaf. Otherwise, `across` iterates through the successors to find the first one that, when included with S, covers more leaves. If found, then we can step `down` the GCP and continue searching. If none of the successors is the ancestor of an uncovered leaf, then S actually covers all leaves, which we signal with an **`error`**.

Additionally, we can explore until we find a signature that does not overlap with S:

$$\begin{aligned}\texttt{inclu\_exclu}((S \cup \{\vec{s}\}), (T_1, \ldots, T_n)) = \texttt{inclu\_exclu}(S, (T_1, \ldots, T_n), S) \\ + \texttt{inclu\_exclu}(\{\vec{s}\}, (T_1, \ldots, T_n))\end{aligned}$$

Which provides us with the first signature that covers only uncovered leaves, *i.e.*, the ancestor of a subset of uncovered leaves, which may prove more helpful to programmers.

4.3.5.1 Running Time

The running time of our search remains polynomial in the running time of `inclu_exclu`, since the maximum number of successors of a given signature is $O(t \cdot n)$: one for each of the immediate successors in each tree. Since t is a bound on the sizes of our trees, it also bounds their depths, so the maximum number of iterations of our search is $O(t^2 \cdot n^2)$. The overall running time of *uncoveredSig* using `inclu_exclu` becomes:

$$\texttt{uncoveredSig}(S, (T_1, \ldots, T_n)) \in O(\, t^3 \cdot n^3 \cdot 2^{|S|})$$

Similar to the `inclu_exclu` based decision solution, `uncoveredSig` is also fixed parameter tractable. When the set of signatures, S, is fixed we have a running time that is cubic in t and n.

CHAPTER 5
Performance

The last component of this thesis is that function-objects over data structure traversal perform well. In this chapter we discuss the performance aspects of DemeterF, what features might inhibit performance, and how we solve these issues. We give experimental results that compare our traversal-based approach to other implementation methods.

5.1 Performance Factors

Function-classes in DemeterF based programs modularize interesting computation. For many traversal-like functions, handwritten implementations share this code with DemeterF implementations, though it is spread throughout different classes. Providing an efficient function-object/traversal-based implementation relies on efficiently replacing the handwritten boiler-plate code. The implementation of a DemeterF `Traversal` can be divided into two parts: the recursive traversal of a data structure, and dispatching to an appropriate *`combine`* method.

5.1.1 Traversal

The implementation of adaptive traversal in DemeterF uses Java reflection to dynamically walk a data structure. This involves inspecting the object when it is traversed and discovering its `Class`. From the class we get the declared fields (transitively) and recursively traversal each of the field values. Other traversal features, *i.e.*, control and contexts, can also add to the inefficiencies since they also require reflection, but there are ways to speed up reflection, like caching the the results according to the `Class` of the object.

Our main approach to speeding up traversals is to generate inlined code that performs the traversal for a specific function-class, control, and context type. For abstract instances we use **`instanceof`** checks to select between subclasses. Once the traversal of a concrete instance's fields is complete, the return results of subtraversals are used to select the appropriate *`combine`* method. With the traversal inlined for a particular function-class, it is then safe to use more specific implementations of dispatch.

5.1.2 Dispatch

As discussed in chapter 4, DemeterF implements two different kinds of dispatch. With our reflective traversal we use a reflective dispatch. When a `Traversal` is created we also reflectively collect the signatures of *`combine`* methods in the given function-object. When a *`combine`* method needs to be called, the method signatures are compared to the recursive result types using an algorithm similar to `select`, in Section 4.2.1.

When a particular function-class and data structure are fixed, we can minimize our method selection by limiting our choices to only the related method signatures and computing a decision tree to select the correct method, similar to our algorithm `residue`, in Section 4.2.2.

5.2 Generating Traversals

One of the major benefits of our separation of function-classes and traversals is that we can provide different (but equivalent) implementations. Our generic traversal can adapt a function-object's `combine` methods to different structures, but we can also replace reflection with static information from a specific CD.

5.2.1 Traversal Inlining

Similar to our generative descriptions in Section 2.7, we describe our traversal generation using a template, which is shown in Listing 5.1. As expected,

```
class Traversal{
   FC fobj;
   Traversal(FC f){ fobj = f; }

   // Generate traversal methods
   ∀A ∈ CD .GenTrav(A)
   ∀C ∈ CD .GenTrav(C)
}
```

Listing 5.1: Traversal generation template

the generated `Traversal` class accepts a function-object. Though only concrete classes exist at runtime, the body of the `Traversal` requires the CD's abstract definitions in order to decide between subclasses. Our traversal generation rule, GENTRAV, is shown below. First for abstract, then concrete definitions.

GENTRAV($A = T_1 \mid \cdots \mid T_n$) $\rightsquigarrow$

```
R traverse<R>(A _h){
  if (_h instanceof T1) return this.<R>traverse((T1)_h);
      ...
  if (_h instanceof Tn) return this.<R>traverse((Tn)_h);
  throw new Exception("Unknown A Subtype");
}
```

For abstract classes we create a simple chain of **if** statements that selects the appropriate recursive `traverse` method for the given instance.[1] In order for the `Traversal` to work with different function-classes/objects, we parametrize each traversal method with the return type, R. For abstract types the parameter is carries through to recursive calls.

The generation rule for concrete definitions is bit more complex:

GENTRAV($C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n$) $\rightsquigarrow$

```
R traverse(C _h, T1 f1, ..., Tn fn){
  Object f1 = this.<Object>traverse(_h.f1);
      ...
  Object fn = this.<Object>traverse(_h.fn);
  return this.<R>apply(fobj,new Object[]{_h, f1, ..., fn});
}
```

For each of a class' fields we recursively call `traverse` and store the result in a local variable. Since our traversal can be used with any function-object, we assume nothing about the return types by using **`object`**. Once all the instance's fields have been traversed we `apply` our function object, `fobj` to an array of the results, including the original object as its first element. The elided `apply` determines the types of the arguments and dynamically dispatches to `fobj`'s most specific `combine` method.

5.2.2 Dispatch Inlining

When we specialize a traversal for a particular function-class, we can replace `Object` in our generated traversals, and `apply` with a calculated decision tree. For our `BExp` structures and `Simplify` function-class from Section 2.5, Listing 5.2 shows the generated traversal method with inlined dispatch for the `Neg` class. The method first recursively calls the general `BExp` traversal method on the instance's `inner` field, then proceeds to select the appropriate method based on the type of the recursive result.

[1] Java will statically resolve the overloaded `traverse` calls because of casting.

```
BExp traverse(Neg _h){
   BExp inner = traverse(_h.inner);
   if(inner instanceof Neg)
      return fobj.combine(_h, (Neg)inner);
   else
      if(inner instanceof False)
         return fobj.combine(_h, (False)inner);
      else
         if(inner instanceof True)
            return fobj.combine(_h, (True)inner);
         else
            return fobj.combine(_h, inner);
}
```

Listing 5.2: `Simplify` traversal method for `Neg`

Note that only the four *combine* methods (from Listing 2.17) that could possibly apply to a `Neg` instance could be called. If all the tests fail, then the method with the most general signature, `(Neg, BExp)` is called. We also note that the declared type of the local variable, `inner`, matches the least upper bound of the return of the possible methods.

When mutual recursion is involved, the situation is similar, though the return types of recursive results will likely be different. Listing 5.3 shows the merged traversal method and dispatch for `Let`. In this case, the recur-

```
BExp traverse(Let _h){
   Bind _bind = traverse(_h.bind);
   BExp _body = traverse(_h.body);
   if(_body instanceof Lit)
      return func.combine(_h, _bind, (Lit)_body);
   else
      return func.combine(_h, _bind, _body);
}
```

Listing 5.3: `Simplify` traversal method for `Let`

sive traversals return different types of results, since `Bind` and `BExp` are not related by subtyping. The dispatch only requires a single test, since the third argument, `_body`, is the only difference between the two possible *code* methods from `Simplify`.

5.2.3 Parallel Traversal

The main benefit of separating traversals and function-objects is that we can replace our traversal without changing the results. The benefit of purely functional (*e.g.*, side-effect free) traversal-based functions is that the order in which subcomponents are traversed is irrelevant to the final result. When it may improve the performance of a particular traversal, we can also perform subtraversals in separate threads. We do this by generating a subclass of

thread to perform a particular traversal in a separate thread, providing a service similar to MultiLisp's `future` annotation [31].

Listing 5.4 shows an **`interface`** `Result` that represents a subtraversal `result` of the type `R`. We use this interface to implement a *possibly* parallel

```
interface Result<R>{
   R result();
}
```

Listing 5.4: Traversal `result` interface

traversal with classes that perform a pending subtraversal either immediately, or in a separate thread.

Listing 5.5 shows `ParTrav`, a thread subclass that is used to implement a separate (parallel) subtraversal. `ParTrav` is also parametrized by the sub-

```
abstract class ParTrav<R> extends Thread implements Result<R>{
   R res = null;
   ParTrav(Traversal t){ this.start(); }

   abstract R traverse();

   public void run(){ setRes(traverse()); }
   synchronized void setRes(R r){ res = r; this.notify(); }
   synchronized R result(){
      if(res == null)this.wait();
      return res;
   }
}
```

Listing 5.5: Synchronized parallel traversal

traversal return type, and has an **`abstract`** method, `traverse`, that is responsible for executing the subtraversal. When a `ParTrav` instance is created, it immediately `starts` itself. The Java runtime will eventually begin executing the `run` method, which will execute the traversal and store the result in the local variable `res`.

For single-threaded traversals we use a simple `Result` implementation that wraps the subtraversal value. Listing 5.6 shows a simple class, `Trav` that is used to unify `Results` for sequential traversals. When a sequential

```
class Trav<R> implements Result<R>{
   R res;
   Trav(R r){ res = r; }
   R result(){ return res; }
}
```

Listing 5.6: Sequential traversal wrapper

traversal is performed we execute the traversal immediately and store it in a `Trav` instance.

In order to execute only specific subtraversals in parallel, we introduce an integer `weight` parameter to every `traverse` method. When a threshhold is reached we create new `ParTrav` threads for subtraversals with a `traverse` that performs the field's subtraversal. If the threshhold is not reached, then a sequential `Trav` is created to hold the result after it is immediately traversed. The `result` methods are used during instance checks for dispatch.

Listing 5.7 shows the generated traversal method for `And` that traverses an instance's `left` and `right` fields in different threads. We first create a

```
BExp traverse(And _h, int weight){
   final Traversal trav = this;

   Result<BExp> left = ((weight != THRESHOLD)?
       new Trav<BExp>(traverse(_h.left, weight+1)):
       new ParTrav<BExp>(){
         BExp traverse()
         { return trav.traverse(_h.left, weight+1); }
       });

   Result<BExp> right = ((weight != THRESHOLD)?
       new Trav<BExp>(traverse(_h.right, weight+1)):
       new ParTrav<BExp>(){
         BExp traverse()
         { return trav.traverse(_h.right, weight+1); }
       });

   if(left.result() instanceof False)
   /*... The rest of dispatch ...*/
}
```

Listing 5.7: Parallel traversal method for `And`

`final` local variable, `trav`, which can be referenced from within our **`new`** anonymous classes. For each field we either create a `Trav` storing the sequential traversal result, or an anonymous subclass of `ParTrav` that implements the `traverse` method by calling the recursive traversal, `trav`, when the thread is eventually `run`. After the wrappers have been created, we begin dispatch by calling the `result` methods of the `Results`, which will either immediately return a result, or wait for the subtraversal to complete. In our inlined implementations we replace the local `Traversal` with the specific traversal we are implementing. In this way we can limit the number of threads created, but at the expense of a bit of extra allocation.

5.3 Experiments and Results

The rest of this chapter presents and discusses a performance comparison and results. We compare DemeterF-based implementations of our `BExp` functions to visitor and hand-written versions. Since DemeterF is implemented

in DemeterF, we also compare different traversal implementations of our class generator for relative performance results.

All experiments were conducted on a Dell Optiplex GX 970 running Ubuntu Linux with two Intel Core 2 Duo 3 Ghz CPUs and 4 Gb of memory. We used Java OpenJDK Runtime (IcedTea6 1.6.1), gave each Java process 35 Mb of heap space (*i.e.*, "`-Xms35M`"), and disabled class garbage collection (*i.e.*, "`-Xnoclassgc`").

5.3.1 Boolean Expressions

In order to demonstrate the performance of traversal-based implementations using function-objects, we implemented functions from chapter 2 by hand and using visitors. The hand written functions are in object-oriented style (*i.e.*, similar to Listing 1.2). The visitors are functional, similar to those illustrated in Section 1.3.2.1, but with traversal implemented in the visitor methods instead of the structures. DemeterF-based implementations use the same function-classes implemented in chapter 2 using a number of different traversals.

Table 5.1 shows performance results for the first four of our seven `BExp` functions. Each sub-table contains the average timing results of 10 different

`ToString`		
Hand	472 msec	1.00
Visitor	482 msec	1.02
Inline	449 msec	.95
Static Trv	1224 msec	2.60

`StrictEval`		
Hand	832 msec	1.00
Visitor	812 msec	.97
Inline	865 msec	1.04
Static Trv	29367 msec	35.34

`Eval`		
Hand	14209 μsec	1.00
Visitor	19866 μsec	1.39
Inline	42230 μsec	2.97
Static Trv	598767 μsec	42.14

`NegNormalize`		
Hand	170 msec	1.00
Visitor	192 msec	1.13
Inline	222 msec	1.30
Static Trv	23162 msec	136.25

Table 5.1: Performance results for `BExp` functions (1)

runs. We called the given function 15 times on a large `BExp` instance, calling Java's garbage-collection (`System.gc()`) between each execution. The columns give (1) the implementation used, (2) the time in milliseconds (microseconds for `Eval`), and (3) the slowdown compared to hand-written methods, *i.e.*, (*time / hand-written-time*). Note that a slowdown of less than 1 is actually a *speedup*.

'Hand' stands for hand-written, 'Visitor' is a functional visitor solution with traversal implemented within the `visit` methods, 'Inline' is DemeterF inlined traversal and dispatch (*i.e.*, `residue` from Figure 4.7), and 'Static Trv' is DemeterF inlined traverasl with a dynamic dispatch (*i.e.*, `select` from Figure 4.6).

Results for `ToString` are relatively even for all implementations, presumably because the concatentation of strings accounts for most of the running time. This is a good example for when the task to be performed by the function is more time consuming than the data structure traversal. DemeterF inlining does slightly better on average in this situation, since we can inline some `combine` selections rather than calling another `traverse` method (*e.g.*, selection for `Lit` can be moved one level up, since neither `True` nor `False` require subtraversals). In the hand-written and visitor solutions the leaf methods must be called separately, *e.g.*, `t.accept(this)`, in order to differentiate instances.

`StrictEval` is a good test of both traversal and dispatch. Since the functionality done at each node is relatively simple, it is mainly the data structure traversal and case differentiation that are stressed, which is evident in the StaticTrv result. The visitor solution performs a bit better, presumably due to locality, and DemeterF inline. It is worth noting that the `BExp` instance used for `StrictEval` and `Eval` is extremely large[2], so this represents a reasonable worst case for all implementations.

`Eval`, on the other hand, represents a best case for hand-written and visitor-based traversals, since the short-cutting recursive case can be caught inline. The DemeterF implementations must dispatch to a method in order to decide whether or not to continue. This increases the stack, in many cases doubling it, and can interfere with garbage collection, which accounts for the near 3 times slowdown. Inline generates a traversal with inlined control (*i.e.*, no tests), but StaticTrv has the additional burden of dynamically checking for `bypass`ing fields.

`NegNormalize` excersizes method arguments, traversal, and dispatch. The DemeterF implementations dispatch to both `combine` and `update` methods, which is evident in the StaticTrv case's poor performance. Vistor and DemeterF Inline implementations are a bit slower than hand-written, but within 30%. The `combine` and `update` selections perform reasonably well, though not as well as single and double dispatch in the hand-written methods and visitors.

Table 5.2 shows performance results for the rest of our `BExp` functions in the same format. `Simplify` is similar in functionality to `NegNormalize`, without the need to pass and `update` traversal arguments (*i.e.*, context). The Visitor and DemeterF Inline implementations consistently perform better than hand-written functions on this task, though not overly so. The Visitor improvement is likely due again to locality again. The performance

[2]The `BExp` in the file is over 15Mb of text, and takes a few seconds to parse.

Simplify		
Hand	250 msec	1.00
Visitor	213 msec	.85
Inline	198 msec	.79
Static Trv	9195 msec	36.78

UsedVars		
Hand	2372 msec	1.00
Visitor	2427 msec	1.02
Inline	2536 msec	1.06
Static Trv	13069 msec	5.51

Invert		
Hand	202 msec	1.00
Visitor	160 msec	.79
Inline	194 msec	.96
Static Trv	9807 msec	48.55

Table 5.2: Performance results for `BExp` functions (2)

of DemeterF inlined version is due to the simpler `combine` method selection for `Neg` and `Let`, which require minimal instance checks rather than multiple method calls.

The results of `UsedVars` is similar to `ToString`, with most of the work being done in the methods, rather than exercizing the data structure traversal. In each of the implmenentations for binary cases (*e.g.*, `And` and `Or`) we must compute the `union` of two `Sets`. The DemeterF inlined version suffers a bit from the extra generality of its `fold` method. The calls to `fold` in the `TU combine` methods for `And`, `Or`, `Bind`, and `Let` account for the slight slowdown. The StaticTrv implementation is slower due to the number of `combine` methods in the function-class, which are eventually passed to an implmentation of `select` (Figure 4.6).

5.3.1.1 Parallel Traversals

To gauge the feasability of parallel traversals using function-objects we ran separate tests comparing our implementations using generated multi-threaded traversals. Table 5.3 shows a comparison of hand-written implementations (as before) with muti-threaded traversals for each of our `BExp` functions. The first column of the table is the function name, *e.g.*, `ToString`, and the second is the time for the hand-written Java implementation in milliseconds, the same as the first columns of Tables 5.1 and 5.2. The third and fourth columns, *1-Thd* and *SD-Hand*, are the execution times for our muti-threaded traversal using a single thread, and its slowdown with respect to the hand-written version respectively. The last three columns are execution times for multi-threaded traversal with 3 threads (1 master, 2 slaves) and its slow-

Function	*Hand*	*1-Thd*	*SD-Hand*	*3-Thd*	*SD-Hand*	*SD-1-Thd*
`ToString`	472	486	1.03	405	.86	.83
`StrictEval`	832	1697	2.04	1272	1.53	.75
`Eval`	14.2	32.7	2.30	149.4	10.51	4.57
`NegNormalize`	170	294	1.73	278	1.63	.95
`Simplify`	250	272	1.09	195	.78	.72
`UsedVars`	2372	2680	1.13	1731	.73	.65
`Invert`	202	268	1.33	254	1.26	.95

Table 5.3: Parallel performance results for `BExp` functions

down as compared to hand-written implementations and the single thread versions respectively.

Our single-threaded version is relatively competative with the hand-written implementations. The slowdown in the first *SD-Hand* column is the result of keeping track of our threshhold/weight parameter and wrapping the sequential traversal results in a `Trav` instance, as in Listing 5.7. Constant factor slowdowns of up to 2.3 is reasonable, considering the extra allocations to wrap almost all recursive subtraversals.

In the final three columns we see the different functions that are most amenable to multi-threading in this manner. Most of the *3-Thd* implementations improve on the *1-Thd* case. In particular, multi-threaded `ToString`, `Simplify`, and `UsedVars` versions *improve* on both the hand-written and *1-Trd* implementations. The functions improve hand-written implementations by 14%, 22%, and 27% while speeding up the single-thread case by 17%, 27%, and 35% respectively.

These examples show that the functions that perform the most work within their `combine` methods are the easiest to improve, though Java allocation can become a multi-threading bottleneck. `Eval` is an extreme case where the function is inherently sequential, and as such does not perform well with multiple threads. The `Eval` function itself takes such little time to complete that it's difficult for other implementations to compete, espesially when extra allocations are involved.

5.3.2 DemeterF

5.3.2.1 DemFGen CD Structures

5.3.2.2 .NET CLI Abstract Syntax

CHAPTER 6

Related Work

6.1 Demeter Tools and Generators

Adaptive (Object-Oriented) Programming (AP) [51] combines datatype descriptions with a domain specific language that selects specific paths of an object instance, over which an impoerative visitor is executed. The two major implementations of adaptive programming, DJ [61] and DemeterJ [66], are similar to DemeterF's dynamic/reflective and static/generated traversals, respectively. DemeterJ uses a similar class dictionary syntax to generate Java classes, a parser, and various default visitors. Ideas from both DemeterJ and DJ have flown into the design of DemeterF, but with a purely functional flavor. DemeterF improves on those tools with safe traversals, extensive support for generics, improved parser generation, and customizable datatype-generic function-class generation.

XML-based generational tools like JAXB [6], XMLBeans [4], and Eclipse Modelling Framework (EMF) [5] can also be used to generate Java classes and XML parsers from data structure *schemas*. Though the design of the created classes attempts to enforce good programming practices, *e.g.*, forcing the use of a factory classes and the separation of class implementations from interfaces, the tools seem to have little support for other generic or generative features, and do not support any notion of parametrized structures. EMF has other features that allow programmers to annotate source files, rather than writing XML schemas, with more generator options.

Parser generators like JavaCC [3] and ANTLR [2] have built in support for generating code for tree based traversals. JavaCC includes a tool JJTree that provides support for writing automatic visitor methods, and ANTLR provides similar functionality with *tree parsers*, but dispatching on the types of nodes is limited and typically must be done by the client in an adhoc mannar.

6.2 Visitors and Multi-methods

The visitor pattern [26] is most commonly used in object-oriented languages to implement functions over datatypes without requiring instance checks or casts. Typical implementations employ double dispatch as shown in chapter 1, though reflection has also been used [62, 61]. The visitor pattern has a sound type-theoretic background [15, 68], and has been at the center of discussions of extensible functions [42] and the expression problem [67, 60, 59]. There is an opinion that multi-methods [22, 20] eliminate the need for the visitor pattern, but visitors can still be used to abstract traversal code, similar to the `Walkabout` [62] and Runabout [30] visitors. In DemeterF we use multiple dispatch to support both abstraction and specialization within function-objects. Our use of multiple-dispatch in function-objects over traversal novel, and in many ways gives us the best of both worlds (*i.e.*, mutiple-dispatch and visitors).

DemeterF's multiple-dispatch and traversal type checking is related to work on static checking of multi-methods [56]. Though Millstien and Chambers are more concerned with balancing modularity and expressiveness, they do focus on eliminating problems associated with multi-method overloading and subclassing across modules. Agrawal *et al*. [7] focus on a simple model of dynamic dispatch and reduce the type checking problem to (1) checking the consistency of overlapping signatures, and (2) confirming that call sites are correct.

Chambers and Leavens [21] eliminate overloading ambiguities by requiring that every combination of argument types have a *most specific* method signature to dispatch to. Their goal is to catch such errors at compile-time, rather than raising a runtime *method ambiguous* exception.

DemeterF dispatch is more like CLOS [65], in that we have an implicit total ordering of applicable method signatures (including shorter signatures), which avoids ambiguities. We are more interested in the possible return types during traversal when using an instance of a given function-class, and making sure that every case has an applicable function.

6.3 Generic and Strategic Programming

Our view of generic programming is influenced by many different projects ranging from generalized folds [64, 54], light-weight functional approaches [44, 45, 47], and visitors [42, 60] to full-fledged generic programming [37, 33], attribute grammars [41], and multi-methods [21, 7].

The notion of traversals that we use is closest to Sheard and Fegaras' work on generalized folds [64], drawing inspiration from Meijer *et al*. [54]. Our traversal function is similar to Sheard's general functor, E, which he uses to implement fold, though we group functions into a class/object, rather than passing them as arguments. Our single `traverse` function takes the

place of a number of very complex functions, one for each value constructor.

The external library and code generation approaches of DemeterF provide significantly more flexibility in both traversal implementation and typing, but requires us to formulate soundness separately. More heavy-weight generic programming [53, 37] can be used to write general traversal functions for any shape data structure, but the level at which functions are written (*e.g.*, over a universal datatype) makes it difficult to integrate hogher level notions like traversal contexts and control.

The benefits of a single traversal function become more pronounced when dealing with mutually recursive types, where fold functions can become difficult to manage. Rather than fixing calls to a particular function argument, our type-based dispatch allows function-classes to abstract multiple cases into one, or overload a case based on argument types.

Library and combinator approaches by Lämmel *et al*. [45, 44] and the *Scrap Your Boilerplate* series of papers [47, 48, 49] support solutions to similar problems using traversal combinators and Haskell's type classes [38]. When the typical *everywhere* traversal is not sufficient, these solutions control recursion using a one-step traversal. Type safety is provided by definition within their implementation language.

Strategic programming (SP) [46, 43] extends combinator approaches, using basic, composable strategies to build reusable traversal schemes. Lämmel *et al*. [50] provides a good overview and comparison to AP traversal "*strategies*". While both SP and AP the benefit from the reusable stretgy components, SP provides different forms of control on transform ordering and short-cutting, while AP supports more goal-based (or *milestone*) traversal control. The strategic approach has also been extended to composable visitors [69], where visitors take the place of basic strategy combinators. The visitors are used to do in-place transformations (*i.e.*, using side-effects).

DemeterF has adopted a simple form of AP strategies that is less goal-based. We use a simple *`bypass`*ing form that allows the traversal to be short-cutting, but eliminates the need for a traversal automata to track milestones. Our style of function-objects may fit nicely into using strategy combinators to define a traversal scheme, since our multiple dispatch is relatively independent of our traversal definition.

6.4 Attribute Grammars

Our traversals, folds, and contexts are similar to an implementation of *attribute grammars* [41]. In Knuth's original description, each attribute is defined by functions over the productions of a context free grammar. In DemeterF CDs, **`abstract`** and **`concrete`** definitions are similar to non-terminals of a context free grammar. In DemeterF, traversing a data structure instance using a function-object corresponds to the evaluation of an attribute's functions over a derivation of the grammar.

The `combine` methods of a function-class correspond to a *synthesized* attribute, with contexts corresponding to an *inherited* attribute. Knuth mentions that attribute grammars can be used to compute arbitrary functions over a derivation of a grammar, and later papers discuss the complexity of checking attribute dependencies and evaluating functions [25]. In DemeterF functions can can be arbitrarily complex, but function-objects without hand-coded recursion correspond to *one-pass* (or *one-visit*) attribute grammars, that can be evaluated left-to-right in a single traversal [13]. Our traversal control also allows the application of functions to be limited to a particular portion of the data structure, though it may be possible to encode similar ideas within attribute functions.

6.5 Language Models

Our model, type system, and soundness builds on simpler ideas from an earlier paper [17], with a much more detailed account in [18], and our approach has been influenced by work on aspect-oriented semantics [71].

Though we maintain a functional approach, our original motivations for separating traversal from other concerns stems from adaptive programming [51] and other visitor based approaches [42, 68, 69]. More recent functional visitor approaches [60, 59] have focused on safety and modularization, but can be mainly categorized as design patterns whereas our aim is to provide a useful library for writing flexible and generic traversal-based functions.

CHAPTER 7

Conclusions

The development of complex software requires the implementation of complex operations over recursively defined data structures. Complex data structures lead to an increase of boilerplate code dealing with structure access and navigation, which makes programs tedious to develop, difficult to maintain, prone to errors, and entangles important functionality resulting in a loss of clarity. This dissertation has proposed a new approach to developing structure-based functions. It is my thesis that this approach is useful, safe, and performs well.

7.1 Contributions

In support of this thesis I have developed DemeterF, a Java based library and set of tools for writing traversal-based functions. The system supports the development of function-classes, facilities for generic programming, a type checker, and generative tools for better traversal performance.

The flexibility of function-objects over traversals, asymmetric multiple dispatch, removal of boilerplate code, and generic programming possibilities make our approach extremely useful for writing functions over data structures, both large and small. DemeterF traversals adapt function-objects to a data structure in order to implement deep, flexible folds. A single function-class/object can handle multiple and mutually recursive structures, and can return results with limited restrictions.

DemeterF's type system and type checker allow functions and data structures to be proven free from dispatch errors, making programs safe. The type checker calculates the return types of a traversal with a specific function-class. It uses a notion of method signature coverage to prove that our multiple dispatch algorithm will succeed for possible recursive return values.

The types of function-classes and traversals can be used to generate traversals for a specific data structure and function-class. We use the structures to generate traversal methods that implement efficient structural recursion. Where the traversal must call to the function-object, we a inline residue decision that implements our multiple dispatch. Because our ap-

proach is side-effect free, generated subtraversals can be executed in parallel. Altogether, we can replace reflective traversals with implementations that in many cases perform as well as hand-written Java functions.

7.2 Future Work

The DemeterF system is relatively complete and has been used in several courses at Northeastern, but there are several extensions and improvements for the future.

7.2.1 Improve Usability

While the DemeterF tools for generic programming and traversal generation have been used to implement DemeterF, there are some parts that are not quite ready for novice end users. In particular the generation of datatype-generic programming (DGP) functions and traversals could be integrated into the CD/BEH languages to allow clients to create traversals as part of the generated classes.[1] We have begun adding syntax to describe the DGP functions to be created for the classes defined in a CD, but the implementation requires a little more work to integrate them fully.

7.2.2 Language Implementation of AP-F

While implementing function-classes and traversals makes them portable, some features (namely traversal control and contexts) reveal our implementation to the client programmer. Implementing a language for DemeterF programs would allow us to integrate features directly and generate safe code, without relying on knowledgeable clients. A complete implementation could also improve performance and enforce side-effect free function-classes and data structures.

7.2.3 Type System Enhancements

The only notable DemeterF features missing from out AP-F model are traversal control and contexts. Finding a way to integrate them into a model would give us a more complete type system and more realistic (though probably messy) proof of type soundness. A statement of type soundness has been absent from Adaptive Programming, and describing soundness in a functional setting would certainly take us a step closer in that direction.

[1] DGP functions are currently given as a command line argument, and loaded on demand. Traversals are described by a separate command and file format.

Bibliography

[1] *Independently Extensible Solutions to the Expression Problem*. ACM, 2005.

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

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

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

[5] Eclipse Modelling Framework. Website, 2010. `http://www.eclipse.org/modeling/emf/`.

[6] JAXB reference implementation. Website, 2010. `https://jaxb.dev.java.net/`.

[7] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *OOPSLA '91*, pages 113–128, New York, NY, USA, 1991. ACM.

[8] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2008.

[9] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[10] Mihir Bellare. Decision versus search. 2004. `http://cseweb.ucsd.edu/~mihir/cse200/decision-search.pdf`.

[11] Mihir Bellare and Shafi Goldwasser. The complexity of decision versus search. *SIAM J. Comput.*, 23(1):97–119, 1994.

[12] Richard S. Bird, Oege de Moor, and Paul F. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[13] Gregor V. Bochmann. Semantic evaluation from left to right. *Commun. ACM*, 19(2):55–62, 1976.

[14] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[15] Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electr. Notes Theor. Comput. Sci.*, 155:309–329, 2006.

[16] Bryan Chadwick. DemeterF: The functional adaptive programming library. Website, 2010. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

[17] Bryan Chadwick and Karl Lieberherr. A Type System for Functional Traversal-Based Aspects. In *AOSD '09, FOAL Workshop*, pages 1–6. ACM, 2009.

[18] Bryan Chadwick and Karl Lieberherr. Functional Traversal-Based Generic Programming. 2010. Submitted to *Higher-Order and Symbolic Computation*, Festscrift for Mitch Wand `http://www.ccs.neu.edu/home/chadwick/files/mitchfest.pdf`.

[19] Bryan Chadwick and Karl Lieberherr. Weaving Generic Programming and Traversal Performance. In *AOSD '10*, pages 61–72. ACM, 2010.

[20] Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92*, pages 33–56. Springer-Verlag, 1992.

[21] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *TOPLAS '95*, 17(6):805–843, November 1995.

[22] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00*, pages 130–145, 2000.

[23] Olivier Danvy. From reduction-based to reduction-free normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in LNCS, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer.

[24] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Info. Proc. Let.*, 106(3):100 – 109, 2008.

[25] Joost Engelfriet and Gilberto Filé. Passes, sweeps, and visits in attribute grammars. *J. ACM*, 36(4):841–869, 1989.

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

[27] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[28] Michael J. C. Gordon. On the power of list iteration. *Comput. J.*, 22(4):376–379, 1979.

[29] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[30] Christian Grothoff. The runabout. *Softw. Pract. Exper.*, 38(14):1531–1560, 2008.

[31] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[32] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[33] Ralf Hinze. A new approach to generic functional programming. In *POPL '00*, pages 119–132, New York, NY, USA, 2000. ACM.

[34] Ralf Hinze and Simon Peyton Jones. Derivable type classes. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[35] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[36] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *TOPLAS*, pages 132–146, 1999.

[37] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97*, pages 470–482. ACM Press, 1997.

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

[39] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242. Springer-Verlag, 1997.

[41] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[42] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113, London, UK, 1998. Springer Verlag.

[43] R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, 2004.

[44] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *PADL '02*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[45] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[46] Ralf Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003. Also available as arXiv technical report cs.PL/0205018.

[47] 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. TLDI '03.

[48] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04*, pages 244–255. ACM Press, 2004.

[49] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05*, pages 204–215. ACM Press, September 2005.

[50] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In *AOSD '03*, pages 168–177, New York, NY, USA, 2003. ACM Press.

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

[52] 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.

[53] Andres Loeh, Johan Jeuring (editors); Dave Clarke, Ralf Hinze, Alexey Rodriguez, and Jan de Wit. Generic haskell user's guide. Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

[54] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA '91*, volume 523, pages 124–144. Springer Verlag, Berlin, 1991.

[55] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *ACM Trans. Program. Lang. Syst.*, 31(2):1–54, 2009.

[56] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99*, pages 279–303, London, UK, 1999. Springer-Verlag.

[57] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[58] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[59] Bruno C. Oliveira. Modular visitor components. In *ECOOP '09*, pages 269–293. Springer-Verlag, 2009.

[60] Bruno C. D. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08*, pages 439–456, 2008.

[61] Doug Orleans and Karl J. Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001*, Kyoto, Japan, September 2001. Springer Verlag.

[62] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98*, Washington, DC, USA, 1998.

[63] Christos H. Papadimitriou. *Computational Complexity*, chapter 10, section 10.3. Addison Wesley, December 1993.

[64] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *FPCA '93*, pages 233–242. ACM Press, New York, 1993.

[65] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

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

[67] Mads Torgersen. The expression problem revisited. In *ECOOP '04*, pages 123–143, 2004.

[68] Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. *FOOL '04*, January 2004.

[69] Joost Visser. Visitor combination and traversal control. In *OOPSLA '01*, pages 270–282. ACM, October 2001.

[70] Philip Wadler. The Expression Problem, 1998. Discussion on the Java-Genericity mailing list.

[71] Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.

[72] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[73] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01*, pages 241–252, New York, NY, USA, 2001. ACM.