Domain-specific languages (DSLs) are now ubiquitous. New DSLs are needed and existing DSLs are evolving all the time. However, creating and maintaining DSLs is hard! There is a lot of engineering effort involved in the creation and the maintenance of a DSL.

One way to address these difficulties is to have language components with high reusability and extensibility. Reusable components reduce the initial effort. Instead of developing everything from scratch, a new DSL is developed through reusing existing components. High extensibility reduces the effort of maintenance, making it easy to customize these components. DSLs, or programming languages in general, share a lot of features. Unfortunately, it is hard to materialize the conceptual reuse into software engineering reuse due to the lack of good modularization techniques. Syntactic modularization techniques based on meta-programming and code generation have been used for code reuse. However, more semantic aspects of modularity, such as the ability to do separate compilation and modular type-checking, are typically missing.

Programming languages have been investigated for seeking proper semantic modularization mechanisms. Object-oriented languages seem to be a good choice as they are designed for extensibility and reuse in the first place. The fundamental features of object-oriented programming (OOP), such as inheritance and subtyping, are vital for developing reusable and extensible components. Combining these features with some classic design patterns, namely the Visitor and Interpreter pattern, a certain degree of extensibility can be achieved. However, they are still insufficient to solve the so-called Expression Problem: one dimension of extensibility is traded by the other. New solutions to the Expression Problem have been proposed, making it possible to develop fully extensible components.

In this thesis we argue that object-oriented languages, equipped with powerful semantic modularization techniques, are suitable for developing extensible DSLs. In the first part of the thesis we develop EVF, an extensible and expressive Java Visitor framework, for facilitating external DSL development. To illustrate the applicability of EVF we conduct a case study, which refactors a large number of non-modular interpreters from the “Types and Programming Languages” book. The resulting interpreters are modular and reusable, sharing large portions of code and features. In the second part of the thesis, we show the close relationship between shallow embeddings and OOP and how OOP improves the modularity of internal DSLs. (378 words)
Extensible Domain-Specific Languages in Object-Oriented Programming

A thesis submitted in partial fulfillment of the requirements for the Degree of Master of Philosophy at The University of Hong Kong

Weixin Zhang

January 2017
Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Weixin Zhang
January 2017
# Contents

## Declaration iii

## List of Figures ix

## List of Tables xi

1 Introduction 1
   1.1 Motivation ............................................. 1
      1.1.1 Language Components ............................. 1
      1.1.2 Language Workbenches ............................ 1
      1.1.3 Internal DSLs ..................................... 2
      1.1.4 Boilerplate Traversals ............................ 2
      1.1.5 Object-Oriented Programming ....................... 3
   1.2 Contributions .......................................... 3
   1.3 Outline ................................................. 4

2 Preliminaries 5
   2.1 The Expression Problem ................................. 5
   2.2 The Interpreter Pattern ................................ 6
   2.3 The Visitor Pattern .................................... 6
   2.4 Expression Problem Trivially ......................... 8
   2.5 Object Algebras ....................................... 9
      2.5.1 Limitations ...................................... 10

3 EVF: An Extensible and Expressive Visitor Framework 13
   3.1 Introduction ........................................... 13
   3.2 Overview ............................................... 14
      3.2.1 Untyped Lambda Calculus: A Running Example .... 15
      3.2.2 A Summary of the Implementations and Results .. 16
      3.2.3 An Implementation with the Visitor Pattern .... 16
      3.2.4 An Implementation with Object Algebras ......... 17
      3.2.5 An Implementation with EVF ....................... 19
   3.3 Extensible Visitors .................................... 22
      3.3.1 Generalized Object Algebras ....................... 22
      3.3.2 Instantiating Generalized Object Algebras as Visitors 23
      3.3.3 Extensibility .................................... 25
## CONTENTS

3.3.4 Object Algebra Interface Generalization ........................................... 25
3.4 Modular Dependencies ............................................................................. 27
3.5 Boilerplate Traversals ............................................................................. 28
  3.5.1 Queries with Default Values ........................................................... 28
  3.5.2 Queries by Aggregation ................................................................... 29
  3.5.3 Transformations ............................................................................... 29

4 Extensible Pattern Matching in EVF .......................................................... 31
  4.1 Introduction ........................................................................................... 31
  4.2 Example: Structural Equality ................................................................. 32
    4.2.1 A Scala Implementation ............................................................... 32
    4.2.2 A Java Implementation Using the Visitor Pattern ...................... 34
    4.2.3 A Java Implementation Using EVF ............................................ 34
    4.2.4 A Final Solution Using EVF with Pattern Matching ............... 38
    4.2.5 Discussion .................................................................................... 39
  4.3 The Design of Pattern Matching EDSL ................................................. 40

5 Case Study and Performance Measurements ............................................ 43
  5.1 Overview ............................................................................................... 43
    5.1.1 Concrete Syntax ........................................................................... 44
    5.1.2 Composable Language Implementations ..................................... 45
    5.1.3 Nontrivial Operations ................................................................... 45
    5.1.4 Multiple Sorts .............................................................................. 46
  5.2 Evaluation .............................................................................................. 46
  5.3 Performance Measurements ................................................................. 47

6 Shallow Embedding and Object-Oriented Programming ......................... 51
  6.1 Introduction ........................................................................................... 51
  6.2 Shallow Object-Oriented Programming .............................................. 52
    6.2.1 SCANS: A DSL for Parallel Prefix Circuits .................................. 52
    6.2.2 Shallow Embeddings and OOP ................................................... 53
  6.3 Interpretations in Shallow Embeddings ............................................... 55
    6.3.1 Multiple Interpretations ............................................................... 55
    6.3.2 Dependent Interpretations ............................................................ 56
    6.3.3 Context-Sensitive Interpretations .............................................. 57
    6.3.4 Adding New Constructs ............................................................... 58
    6.3.5 Discussion .................................................................................... 59

7 Related Work and Conclusion .................................................................... 61
  7.1 Related Work ......................................................................................... 61
    7.1.1 The Visitor Pattern ...................................................................... 61
    7.1.2 Object Algebras ............................................................................ 62
    7.1.3 Solutions to the EP in Functional Programming ....................... 62
    7.1.4 Component-Based Development .............................................. 62
List of Figures

2.1 The Interpreter pattern ........................................... 6
2.2 The Visitor design pattern ...................................... 7
2.3 An object algebra interface for the expression language .......... 9
2.4 Client code for Object Algebras .................................. 10
2.5 Non-modular dependent operations in Object Algebras .......... 11

3.1 Untyped lambda calculus with the Visitor pattern. ............... 17
3.2 Untyped lambda calculus with Object Algebras. .................. 18
3.3 The complete code for the untyped lambda calculus with EVF. .. 19
3.4 Untyped lambda calculus with EVF ................................. 20
3.5 Instantiation and client code for untyped lambda calculus .......... 20
3.6 An internal visitor implementation for GExpAlg ................. 23
3.7 An external visitor implementation for GExpAlg .................. 24
3.8 Generating generalized object algebra interface .................. 26
3.9 Modular dependent operation in EVF ............................... 27

4.1 Structural equality using case classes ............................ 33
4.2 Structural equality using the Visitor pattern ..................... 35
4.3 Structural equality using EVF ...................................... 36
4.4 Extended structural equality using EVF ............................ 37
4.5 Structural equality using EVF with pattern matching ............ 38
4.6 An overview of pattern matching EDSL ......................... 41

5.1 Package dependency graph ....................................... 44
5.2 Small-step evaluator for arith .................................... 45
5.3 Linked list and sum operation implementations ................ 48

6.1 The Brent-Kung parallel prefix circuit of width 4 ............... 53
6.2 Adding new interpretations ....................................... 56
List of Tables

4.1 Pattern matching support in sealed case classes, open case classes, Visitor, 
**Evf**, and **Evf** with patterns: fully supported ●, partially supported ○, not supported ○ ......................................................... 39

5.1 SLOC statistics **Evf** vs OCaml: A package perspective. ................... 47
5.2 SLOC comparison **Evf** versus OCaml: A component perspective ........ 47
5.3 Performance measurements ............................................................... 48
Chapter 1

Introduction

1.1 Motivation

Domain-specific languages (DSLs) are now ubiquitous. DSLs, as the name suggests, are programming languages tailored for specific domains, which build the gap between domain experts and programmers. Examples of widely used DSLs include SQL for database queries, VHDL for hardware description, and MATLAB for numerical computing. New DSLs are needed and existing DSLs are evolving all the time. However, creating and maintaining DSLs is hard! There is a lot of engineering effort involved in the creation and the maintenance of a DSL. To develop a DSL, various components need to be implemented, such as syntactic and semantic analyzers, a compiler or interpreter, and tools that are used to support the development of programs in that language. To maintain a DSL, bugs have to be fixed, and new features have to be implemented.

1.1.1 Language Components

One way to address the difficulties in developing and maintaining DSLs is to have language components with high reusability and extensibility. High reusability reduces the initial effort. Instead of developing everything from scratch, a new DSL is developed through reusing existing components. High extensibility reduces the effort of maintenance, making it easy to customize these components. DSLs, or programming languages in general, share a lot of features. For example, although VHDL and MATLAB are designed for different domains, they have many features in common. Both of them provide mechanisms to declare variables, support basic arithmetic operations as primitives, have loop constructs, etc. Moreover, nearly all new languages will "copy" many features from existing languages, rather than having a completely new set of features. Therefore, there is an inherent conceptual reuse in programming languages. Unfortunately it is hard to materialize the conceptual reuse into software engineering reuse due to the lack of good modularization techniques.

1.1.2 Language Workbenches

Language workbenches are aimed at lowering the amount of effort required to develop new DSLs. They provide rapid prototyping of languages and related programming
tools, including IDEs, debugging tools, as well as other editor services. Examples of modern, mature language workbenches include: Xtext [15], MPS [20] and Spoofax [33]. At the moment some language workbenches provide support for language components through syntactic modularization techniques based on meta-programming and code generation. Such techniques allow components to be specified in separate files. However, more semantic aspects of modularity, such as the ability to do separate compilation and modular type-checking are typically missing in meta-programming approaches. The lack of separate compilation means that compilation time can be quite high, which limits the scalability of such approaches. Furthermore, it also prevents a business model where companies would sell the binaries of the language components to customers. For companies this is an attractive business model because it allows customers to build customized products, without giving these customers access to the source code (and to the company’s intellectual property). The lack of modular type-checking makes the development process more complex and brittle. Without modular type-checking, many errors can only be detected at very late stages in the development, or even go undetected.

1.1.3 Internal DSLs

Although language workbenches simplify the development of DSLs, they still require a considerable amount of engineering effort and expert knowledge in language design. A more lightweight way to implement a DSL is to embed it into an existing mature programming language. DSLs implemented in this way are called internal or embedded DSLs [21, 29]. Different from an external DSL which has its own infrastructure, an internal DSL reuses the whole well-developed ecosystem of the host language, minimizing the engineering effort needed for developing a DSL. Examples of well-known internal DSLs include LINQ [40] embedded in .NET languages for data queries and Rake [69], a build tool embedded in Ruby. However, without a stand-alone parser, the syntax of an internal DSL is restricted by the host language and hence may not be as natural as that of an external DSL.

1.1.4 Boilerplate Traversals

Developing the semantics of a DSL also requires a lot of engineering effort. Abstract Syntax Tree (AST) is the core data structure in modeling a language, serving as the intermediate representation of a program. Operations that analyze or transform a program, such as type-checking and optimizations, are defined against to an AST. However, these operations often contain a large amount of boilerplate code [36] in traversing the AST. Typically, for a particular operation, only a small set of the language constructs is interesting, where programmers should write code that does the computation. However, for those boring cases, programmers still have to write code for tree traversals. Such boilerplate is tedious to write and problematic to maintain. The more constructs a language has, the more severe the situation is. Approaches like Adaptive Object-Oriented Programming [37] and Strategic Programming [64] try to eliminate boilerplate traversals. However, these approaches use meta-programming techniques, losing modular type-checking and...
separate compilation.

1.1.5 Object-Oriented Programming

Programming languages have been investigated for seeking proper semantic modularization mechanisms so that modular type checking and separate compilation can be preserved. Object-oriented (OO) languages seem to be a good choice as they are designed for extensibility and reuse in the first place. The fundamental features provided by OO languages, such as inheritance and subtyping, are vital for developing reusable and extensible components. Inheritance and overriding allow extensibility on components. Multiple-inheritance supported by some of OO languages, for example C++ multiple inheritance, Scala mixin-inheritance and Java default methods, goes further, allowing multiple components to be unified. Subtyping and late binding enable flexible reuse of components. Combining these features with some classic design patterns [22], namely the Visitor and Interpreter pattern, a certain degree of extensibility can be achieved. However, they are still insufficient to solve the so-called Expression Problem (EP) [67]: one dimension of extensibility is traded by the other.

New solutions to the EP have been proposed, making it possible to build fully extensible components. Oliveira and Cook proposed Object Algebras [46], a design pattern closely related to the Visitor pattern, using generics. To address the problem of boilerplate traversals, the Object Algebras based Shy framework [74] generates traversal templates while preserving modular type-checking and separate compilation. Unfortunately, due to the bottom-up nature of Object Algebras, it is awkward or inefficient to implement operations that need flexible traversal strategies in Shy. Wang and Oliveira [68] gave a simpler solution based on the Interpreter pattern without using generics. But the simplicity comes at the cost of expressiveness: binary operations or transformations cannot be modeled modularly.

1.2 Contributions

In this thesis we argue that OO languages, equipped with powerful semantic modularization techniques, are suitable for developing extensible internal and external DSLs. We show how to modularize language components while retaining modular type-checking and separate compilation in OOP. In the first part of the thesis, we develop EVF, an extendible and expressive Java Visitor framework, for facilitating external DSL development. EVF brings extra expressiveness to Object Algebras while retaining extensibility. To illustrate the applicability of EVF we conduct a case study, which refactors a large number of non-modular interpreters from the “Types and Programming Languages” (TAPL) book [51]. The resulting interpreters are modular and reusable, sharing large portions of code and features. In the second part of the thesis, we show the close relationship between shallow embeddings and OOP and how OOP improves the modularity of internal DSLs. We make our argument by refactoring the DSL presented in Gibbons and Wu’s paper [23] using Wang and Oliveira’s solution. Nevertheless, this design pattern can be applied to develop
shallow embedded DSLs using their approach.

In summary, the contributions of the thesis are:

- **A new approach to extensible generic visitors:** We present a novel technique to support extensible external visitors that works in Java-like languages. The new technique allows modular dependencies to be expressed using standard OO techniques.

- **Generalized generic queries and transformations:** EVF overcomes the bottom-up limitations of generic queries and transformations of Shy, and supports top-down traversals as well.

- **Extensible pattern matching with defaults:** EVF has a fluent interface style embedded DSL that emulates extensible algebraic datatypes with defaults.

- **Code generation for visitor boilerplate code:** Using an annotation processor, EVF generates large amounts of boilerplate code related to ASTs and AST traversals. Users only need to specify an annotated Object Algebra interface to trigger code generation.

- **Implementation and TAPL case study:** We illustrate the practical applicability of EVF with a large case study that refactors a nontrivial and non-modular OCaml code base into modular and reusable Java code.¹

- **Exposing the close connection between shallow embedding and OOP:** We show that procedural abstraction used in shallow embedding is also the essence of OOP.

## 1.3 Outline

The rest of the thesis is organized as follows: Chapter 2 gives some preliminaries of this thesis. Chapter 3, 4 and 5 form the first part of the thesis, which introduce EVF as a framework for developing external DSLs. Specifically, in Chapter 3, we introduce the basic infrastructure of EVF, including the generalized Object Algebras and traversal templates; in Chapter 4, we add pattern matching support to EVF; Chapter 5 is a case study illustrating the applicability of EVF. Chapter 6 itself forms the second part of the thesis, which discusses extensible internal DSLs in OOP. Chapter 7 compares our work with the state-of-the-art and concludes the thesis.

¹Code for EVF and the case studies is available via [https://github.com/wxzh/evf](https://github.com/wxzh/evf)
This chapter provides some preliminaries of this thesis. We first introduce the Expression Problem (EP) - the fundamental problem to solve for developing modular and extensible software components. Then we review two design patterns in OOP, namely the Interpreter and Visitor pattern, and explain why they fail to solve the EP. Next, we present two newly proposed solutions to the EP that are closely these two design patterns and discuss their respective limitations.

2.1 The Expression Problem

Reynold [55] firstly pointed out that procedural abstraction and user-defined types are complementary in terms of extensibility. Cook [13] further clarified the extensibility problem and argued that procedural abstraction is the essence of OOP. Krishnamurthi et al. [35] gave the first solution to the extensibility problem based on the Visitor pattern. Wadler [67] coined the term “Expression Problem” to the challenge of extensibility.

To illustrate, consider evolving a simple arithmetic expression language. The initial system contains two types of expressions, integer literals and subtractions, and one operation, evaluation. The problem is to add a new type of expression, e.g. negations, and a new operation e.g. pretty printing. This example precisely captures the challenge occurred in software evolution - the need of both adding new variant and new operation. A proper solution to EP should meet the following requirements:

- **Extensibility in both dimensions**: A solution allows both new data variants and new operations to be added. Also, existing operations can be extended to support new variants.
- **Strong static type safety**: Applying an operation to a data variant that it cannot handle is
- **No modification and duplication**: Existing code must neither be modified nor duplicated.
- **Separate compilation and type-checking**: The original datatype and existing operations should not be type-checked and compiled again when extensions are added.

And a fifth requirement added by Odersky and Zenger [44]:

- **Separate compilation and type-checking**: The original datatype and existing operations should not be type-checked and compiled again when extensions are added.
• **Independent extensibility:** It should be possible to combine independently developed extensions, so that they can be used jointly.

### 2.2 The Interpreter Pattern

```java
interface Exp {
    int eval();
}
class Lit implements Exp {
    int n;
    Lit(int m) { n = m; }
    public int eval() {
        return n;
    }
}
class Sub implements Exp {
    Exp x, y;
    Sub(Exp l, Exp r) { x = l; y = r; }
    public int eval() {
        return x.eval() - y.eval();
    }
}
```

Figure 2.1: *The Interpreter pattern*

The Interpreter pattern [22] is specifically designed for implementing languages. The Interpreter pattern uses the Composite pattern [22] to represent recursive structure using hierarchical classes in OOP. Figure 2.1 models the expression language using the Interpreter pattern. The interface `Exp` describes all the supported operations for an expression. Class `Lit` and `Sub` implement the `Exp` interface and are concrete expressions, representing integer literals and subtractions respectively. We can easily add a new variant by re-implementing the `Exp` interface:

```java
class Neg implements Exp {
    Exp x;
    Neg(Exp e) { x = e; }
    public int eval() {
        return -x.eval();
    }
}
```

However, adding a new operation becomes difficult. We have to modify the `Exp` hierarchy to insert the definition of that operation, breaking the no modification requirement of the EP. Hence the Interpreter pattern is commonly used for modeling structures on which operations are known in advance.

### 2.3 The Visitor Pattern

The Visitor design pattern [22] separates an operation from an object structure. It emulates functional decomposition in OO languages. Figure 2.2 models the expression language using the Visitor pattern. The expression structure is organized using a way similar to the Interpreter pattern. The difference is that `Exp` in the Visitor pattern declares
interface Visitor<O> {
  O Lit(int n);
  O Sub(Exp x, Exp y);
}

interface Exp {
  <O> O accept(Visitor<O> v);
}

class Lit implements Exp {
  int n;
  Lit(int m) { n = m; }
  public <O> O accept(Visitor<O> v) {
    return v.Lit(n);
  }
}

class Sub implements Exp {
  Exp x, y;
  Sub(Exp l, Exp r) { x = l; y = r; }
  public <O> O accept(Visitor<O> v) {
    return v.Sub(x, y);
  }
}

class Eval implements Visitor<Integer> {
  public Integer Lit(int n) {
    return n;
  }
  public Integer Sub(Exp x, Exp y) {
    return x.accept(this) - y.accept(this);
  }
}

only one accept() method, which takes a Visitor to traverse itself. The Visitor interface abstracts over the operations that can be applied to Exp. For each type of expressions there is a corresponding method declared inside Visitor. Eval is a concrete implementation of the Visitor interface, which defines an evaluation operation on expressions.

The Visitor pattern makes it easy to add new operations through re-implementing the Visitor interface. We can, for example, implement a pretty-printing operation for the expression language:

class Print implements Visitor<String> {
  public String Lit(int n) {
    return String.valueOf(n);
  }
  public String Sub(Exp x, Exp y) {
    return "(" + x.accept(this) + "," + y.accept(this) + ")";
  }
}

However, adding variants becomes a problem. To add a variant, the Visitor interface as well as all existing implementations have to be modified for dealing with that new variant, which again breaks the no modification requirement of the EP.

Internal and External Visitors Implementations of the Visitor pattern can be further classified. Depending on who controls the traversal, Buchlovsky and Thielecke [9] classified visitors as *internal* or *external*. In internal visitors the traversal strategy is hard-encoded into the structure and hence expose no direct control over traversal to operations.
On the other hand, in external visitors, traversal is delegated to the operation. As a result, internal visitors are simpler to use whereas external visitors are more flexible in traversal. Figure 2.2 is an example of external visitors, where the traversal on sub-expressions is controlled by explicitly calling the accept() inside Eval.

### 2.4 Expression Problem Trivially

Recently, Wang and Oliveira [68] propose a simple solution to the EP based on the Interpreter pattern. To model the expression language using this approach, most of the code shown in Figure 2.1 remains the same except for composite structures such as Sub:

```java
abstract class Sub implements Exp {
    abstract Exp x();
    abstract Exp y();
    public int eval() {
        return x().eval() - y().eval();
    }
}
```

Sub-expressions of Sub are now captured by abstract getters x() and y() rather than fields and hence the class is abstract. This change allows covariant type refinements on the sub-expressions, which is vital for extensibility.

As Sub is an abstract class, an additional instantiation phase is needed:

```java
class SubImpl extends Sub {
    Exp x, y;
    Exp x() { return x; }
    Exp y() { return y; }
    SubImpl(Exp l, Exp r) { x = l; y = r; }
}
```

The concrete class SubImpl contains fields and implements the abstract getter using these fields.

**Adding New Variants** This approach retains the simplicity of adding variants in the Interpreter pattern. We can define Neg in a way similar to Sub:

```java
abstract class Neg implements Exp {
    abstract Exp x();
    public int eval() {
        return -x().eval();
    }
}
```

**Adding New Operations** The following code shows how to add a pretty-printing operation:

```java
interface ExpExt extends Exp {
    String print();
}
class LitExt extends Lit implements ExpExt {
    LitExt(int n) { super(n); }
    public String print() {
        return String.valueOf(n);
    }
}
abstract class SubExt extends Sub implements ExpExt {
    abstract ExpExt x(); // return type refined!
}
```
To support new operations, a new class hierarchy is needed. The interface `ExpExt` extends `Exp` and introduces a new method `print()`. Class `LitExt` and `SubExt` extend their respective base class and implement the new interface. Note that the return type of the getters are refined to `ExpExt`, so that we can call `print()` method on sub-expressions.

**Limitations** If the structure itself is used as an argument or a return value in an operation, then the operation cannot be modeled modularly with this approach. Such operations include transformations and binary methods. These operations can be made modular with advanced type system features, for example Scala’s virtual types [73]. However, these features are not available in mainstream OO languages like Java. Also the encoding will become complicated afterwards.

## 2.5 Object Algebras

Object Algebras [46] are a solution to the EP closely related to the Visitor pattern. We give a brief introduction of Object Algebras and summarize their limitations here.

**Object Algebra Interfaces** An object algebra interface represents an algebraic signature [26]. Figure 2.3 models the abstract syntax of the expression language using an object algebra interface. The language contains only two constructors (i.e. methods declared in `ExpAlg`) representing integer literals and subtraction expressions.

**Object Algebras** An object algebra is an operation defined on an object algebra interface. To do so, one should implement that interface by instantiating the type parameter list with concrete types accordingly. For example, the following code implements the evaluation operation for the expression language:

```java
class Eval implements ExpAlg<Integer> {  
  public Integer Lit(int n) {  
    return n;  
  }  
  public Integer Sub(Integer x, Integer y) {  
    return x - y;  
  }  
}  
```
Adding New Operations  Defining new operations over the algebra interface is trivial. Similar to `Eval` we can define a pretty printer for expressions:

```java
class Print implements ExpAlg<String> {
    public String Lit(int n) {
        return String.valueOf(n);
    }
    public String Sub(String x, String y) {
        return "(" + x + "-" + y + ")";
    }
}
```

Adding New Variants  We can also add new variants by extending the base algebra interface and putting new variants in the extended algebra interface. For instance, the interface `ExtAlg` illustrates how to introduce addition expressions (`Add`) to the expression:

```java
interface ExtAlg<Exp> extends ExpAlg<Exp> {
    Exp Neg(Exp x);
}
```

All algebras defined for the base algebra interface (`ExpAlg`) are retroactive, in the sense that we only need to inherit the base algebra and complement the new cases:

```java
class EvalExt extends Eval implements ExtAlg<Integer> {
    public Integer Neg(Integer x) {
        return -x;
    }
}
```

```java
<Exp> Exp mkExp(ExpAlg<Exp> alg) {
    return alg.Sub(alg.Lit(2), alg.Lit(1));
}
mkExp(new Print()); // "(2-1)"
mkExp(new Eval()); // 1
```

Figure 2.4: Client code for Object Algebras

Client Code  Figure 2.4 shows how to construct an expression and apply different operations to it. The generic method `mkExp` uses constructors defined on an object algebra instance to create an object. `Print` and `Eval` are both concrete object algebras defined for `ExpAlg`, thus their instances can be passed to `mkExp`.

2.5.1 Limitations

Object Algebras are indeed a simple variant of internal visitors, which bypass the data structure and directly construct objects using operations. Although Object Algebras provide type-safe extensibility, they come with some limitations which we summarize as follows:

Fixed Traversal Pattern  Object Algebras are essentially Church encodings [46], which fix the traversal strategy to be **bottom-up** and will always traverse the structure entirely. It is inefficient or awkward to define certain operations. For example, checking whether an expression is a literal should ideally be an \(O(1)\) operation by inspecting the root node only, but the bottom-up nature of the traversal makes it an \(O(n)\) operation, where \(n\) is
interface IExp {
    String print();
    boolean isLit();
}

class MPrint extends ExpAlg<IExp> {
    public IExp Lit(int n) {
        return new IExp() {
            public String print() {
                return String.valueOf(n);
            }
            public boolean isLit() {
                return true;
            }
        };
    }
    public IExp Sub(IExp x, IExp y) {
        return new IExp() {
            public String print() {
                return x.print() + "-" + (y.isLit() ? y.print() : "(" + y.print() + ")");
            }
            public boolean isLit() {
                return false;
            }
        };
    }
}

Figure 2.5: Non-modular dependent operations in Object Algebras

the total number of nodes. The reason is that sub-expressions will always be checked beforehand, unnecessarily.

No Concrete Representation In Object Algebras there is no data structure that persists the AST. Rather, the AST is virtually formed as constructor applications. To apply different operations to the same AST, a mkExp-like method has to be called multiple times with different concrete algebras as Figure 2.4 shows. As Gouseti et al. [24] noticed, this requires reparsing the whole input program for every operation applied to the AST. Although Gouseti et al. attempted to solve this problem through delayed instantiation and caching [24], a better solution would be simply to have a concrete representation.

Difficult to Express Dependencies Object Algebras must be self-contained. That is, they cannot reuse other peer algebras defined for the same algebra interface. This makes it hard to modularize dependent operations. For example, Figure 2.5 illustrates how to define a dependent operation in Object Algebras. To print an expression with minimal parentheses, the print() operation depends on not only itself but also the isLit() operation, for checking whether an expression is a literal. Parentheses are put around a subtrahend only when it is not a literal. The solution given in Figure 2.5 is not modular because the dependent operation [print()] has to be defined together with what it depends on (isLit()) by implementing a pair-like interface (IExp). In principle, a more modular implementation will define isLit() separately, as an object algebra say IsLit. And then define MPrint in terms of IsLit. This way, IsLit can be reused elsewhere and the implementation of MPrint will be shortened dramatically. However, the lack of proper reusing mechanisms hinders such a modular implementation and results in bloated algebras that are hard to use and
Chapter 2. Preliminaries

maintain.
In this chapter, we introduce **EVF**, an extensible and expressive Visitor framework. The extensibility and expressiveness of **EVF** owe to a generalization of Object Algebras and the Visitor pattern. The key advance in **EVF** is a novel technique to support extensible external visitors that works in Java-like languages, which are able to control traversals on the data structure being traversed. This removes many of the limitations of internal visitors and Object Algebras. To make **EVF** practical, the framework employs annotations to automatically generate large amounts of boilerplate code related to visitors and traversals.

### 3.1 Introduction

The Visitor pattern [22] is widely used in object-oriented programming. The Visitor pattern separates a traversal algorithm from a data structure. This has the benefit that it becomes possible to add new operations to existing object structures modularly (that is, without modifying existing structures). Many traditional applications of the Visitor pattern are in language processing tools, such as compilers, interpreters or program analysis tools. In those applications, traversals over the abstract syntax of the language are typically modeled with visitors. Since in language processing tools it is quite often that new traversals over the abstract syntax are needed, the use of the Visitor pattern allows the new code to be added without modifying existing classes.

In reality, however, as software evolves extensions on both operations and variants are needed. Robbes et al. [56] conduct an empirical study on a large number of Smalltalk projects about extensions during evolution. Their result shows that extensions on operations and variants happen equally frequently. Surprisingly enough, this conclusion holds for projects that apply the Visitor pattern, which are expected to have much more operation extensions than variant extensions. But a well-known limitation of traditional forms of the Visitor pattern is the lack of *type-safe extensibility* [35, 67, 72]. If the data structure itself needs to be extended there are usually two options. One option is to modify and update existing code (including code for existing traversals using visitors). The other option is not to modify existing code and to use inheritance and casts instead to achieve
extensibility. Either way there is a trade-off: the first option is not modular/extensible; the second option is not type-safe.

Much work has been done to equip the Visitor pattern with type-safe extensibility [62, 73, 45]. But these solutions require sophisticated type system features not available in Java and are impractical for real-world applications. Recently, Object Algebras [46] have been shown to provide a solution to type-safe extensibility, and as a possible alternative to the Visitor pattern. However, extensibility in Object Algebras is achieved at the cost of some expressiveness when compared to traditional forms of the Visitor pattern. In particular, in their basic form [46], Object Algebras have the following limitations:

- It is hard to express computations that are not bottom-up;
- It is difficult to modularly define complex dependencies between operations;
- There is no persistent representation of the data structure.

These limitations preclude the use of Object Algebras in many of the traditional applications of the Visitor pattern. Although there has been some recent research addressing the first two issues in Scala [47, 54, 30], this is done using advanced language features that are not available in most OO languages. Moreover, the problem of dependencies is only solved with an encoding of delegation, instead of reusing existing OO mechanisms. Finally, the last issue has not been addressed before.

This chapter presents **EVF**: an extensible and expressive Java Visitor framework. **EVF** visitors support type-safe extensibility while retaining the expressiveness needed for many of the applications of the Visitor pattern. The support for extensibility improves on techniques used by Object Algebras, and previous work on modular visitors [45]. It is known that Object Algebras are closely related to internal visitors [48] (a simple, but less expressive, variant of visitors). The key advance in **EVF** is a novel technique to support extensible external visitors that works in Java-like languages. In contrast to internal visitors, external visitors are able to control traversals with direct access to the data structure being traversed. This allows them to remove many of the limitations of internal visitors and Object Algebras. **EVF** does not have any of the limitations listed above.

To make **EVF** practical, **EVF** employs annotations to automatically generate large amounts of boilerplate code related to visitors and traversals. In essence, a user needs only to specify an algebra interface, which describes the desired structure. **EVF** processes that interface and generates various useful interfaces and classes. Noteworthies are **EVF**’s generic queries and transformations, which generalize Shy-style traversals [74] and remove the limitation of bottom-up only traversals.

### 3.2 Overview

In this section, we introduce the **EVF** framework by modeling some operations on the untyped lambda calculus. The untyped lambda calculus example illustrates the advantages of **EVF** in terms of flexibility, modularity and reuse over traditional (non-modular) visitors and Object Algebras.
3.2 Overview

3.2.1 Untyped Lambda Calculus: A Running Example

The language we are going to model is based on untyped lambda calculus. We formalize its syntax and operational semantics, following Pierce’s definition [51].

Syntax  The syntax of the language is given below:

\[ e ::= x \quad \text{variable} \]
\[ \lambda x.e \quad \text{abstraction} \]
\[ e_1 e_2 \quad \text{application} \]
\[ i \quad \text{literal} \]
\[ e_1 - e_2 \quad \text{subtraction} \]

The language has five syntactic forms: variables, lambda abstractions, lambda applications, integer literals and subtractions. The meta variable \( e \) ranges over expressions; \( x \) over variable names; \( i \) over integers. With the syntax of the language given, its operational semantics can be defined.

Free Variables  The first operation collects the set of free variables of an expression \( e \), written as \( FV(e) \). A variable in an expression is said to be \( \text{free} \) if it is not bound by any enclosing abstractions. The formal definition is given below:

\[ FV(x) = \{x\} \]
\[ FV(\lambda x.e) = FV(e) \setminus \{x\} \]
\[ FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \]
\[ FV(i) = \emptyset \]
\[ FV(e_1 - e_2) = FV(e_1) \cup FV(e_2) \]

The definition relies on some set notations. Their meanings are: \( \emptyset \) denotes an empty set; \( \{x\} \) represents a set with one element \( x \); \( \setminus \) calculates the difference of two sets; \( \cup \) is the set union operator.

Substitution  Substitution, written as \([x \mapsto s]e\), is an operation that replaces all free occurrences of variable \( x \) in the expression \( e \) with the expression \( s \), given by:

\[ [x \mapsto s]x = s \]
\[ [x \mapsto s]y = y \quad \text{if } y \neq x \]
\[ [x \mapsto s](\lambda x.e) = \lambda x.e \]
\[ [x \mapsto s](\lambda y.e) = \lambda y.[x \mapsto s]e \quad \text{if } y \neq x \land y \notin FV(s) \]
\[ [x \mapsto s](e_1 e_2) = [x \mapsto s]e_1 [x \mapsto s]e_2 \]
\[ [x \mapsto s]i = i \]
\[ [x \mapsto s](e_1 - e_2) = [x \mapsto s]e_1 - [x \mapsto s]e_2 \]
The definition is indeed quite subtle, especially for the abstraction case. The body of an abstraction will be substituted only when two conditions are satisfied. The first condition, \( y \neq x \), makes sure that the variable to be substituted is not bound by the abstraction. The second condition, \( y \notin FV(s) \), prevents free variables in an expression \( s \) being bound after substitution. The two conditions together preserve the meaning of an expression. For example, \( [x \mapsto y](\lambda x.x) \) is not \( \lambda x:y \) because the variable \( x \) is bound and \( [x \mapsto y](\lambda y.x) \) is not \( \lambda y.y \) because \( y \) is free in the expression to substitute.

### 3.2.2 A Summary of the Implementations and Results

We implemented the untyped lambda calculus using the Visitor pattern, Object Algebras and \textbf{EVF} respectively. The following table summarizes the implementations from three perspectives: the source lines of code (SLOC), modularity and the number of cases to implement for free variables and substitution.

<table>
<thead>
<tr>
<th>Approach</th>
<th>SLOC</th>
<th>Modular</th>
<th># of cases for free variables</th>
<th># of cases for substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Visitor Pattern</td>
<td>88</td>
<td>No</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Object Algebras</td>
<td>89</td>
<td>Yes</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>EVF</td>
<td>32</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

From the table we can see that the implementation using \textbf{EVF} is best in all these three aspects. It is modular, uses less than half of the SLOC than the other two solutions, and has the least number of cases to implement for both operations. The remainder of this section explains the three implementations and the results in detail.

### 3.2.3 An Implementation with the Visitor Pattern

We first discuss an implementation with the (external) Visitor pattern presented in Figure 3.1 (full version can be found in Appendix A.1).

**Syntax** The visitor interface \texttt{LamAlg} describes the constructs supported by the language. The \texttt{Exp} interface represents the AST type. Classes that implement \texttt{Exp}, for instance \texttt{Var} and \texttt{Abs}, are concrete constructs of the language. The \texttt{LamAlg} interface declares (visit) methods to deal with these constructs, one for each. Concrete constructs use their corresponding visit method in implementing the \texttt{accept} method exposed by the \texttt{Exp} interface.

**Free Variables** Operations for the language are defined as concrete implementations of the \texttt{LamAlg} interface. A concrete visitor \texttt{FreeVars} collects free variables from an expression. \texttt{FreeVars} implements \texttt{LamAlg} by instantiating the type parameter as \texttt{Set<String>}. Since the traversal is controlled by the programmer via the \texttt{accept} method, we call \texttt{e.accept(this)} to collect free variables from the body of the abstraction.

**Substitution** Similarly, the class \texttt{SubstVar} models substitution. Substitution is a transformation over the expression structure. We hence instantiate the abstract type of \texttt{LamAlg}
3.2 Overview

```java
interface LamAlg<O> {
    O Var(String x);
    O Abs(String x, Exp e);
    O App(Exp e1, Exp e2);
    O Lit(int n);
    O Sub(Exp e1, Exp e2);
}
```

```java
interface Exp {
    <O> O accept(LamAlg<O> v);
}
```

```java
class Var implements Exp {
    String x;
    Var(String x) {
        this.x = x;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Var(x);
    }
}
```

```java
class Abs implements Exp {
    String x;
    Exp e;
    Abs(String x, Exp e) {
        this.x = x;
        this.e = e;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Abs(x, e);
    }
}
```

```java
class FreeVars implements LamAlg<Set<String>> {
    public Set<String> Var(String x) {
        return Collections.singleton(x);
    }
    public Set<String> Abs(String x, Exp e) {
        return e.accept(this).stream()
            .filter(y -> !y.equals(x))
            .collect(Collectors.toSet());
    }
    ...
}
```

```java
class SubstVar implements LamAlg<Exp> {
    String x;
    Exp s;
    SubstVar(String x, Exp s) {
        this.x = x;
        this.s = s;
    }
    public Exp Abs(String y, Exp e) {
        if (y.equals(x)) return new Abs(x, e);
        if (s.accept(new FreeVars()).contains(x))
            throw new RuntimeException();
        return new Abs(x, e.accept(this));
    }
    public Exp Var(String y) {
        return y.equals(x) ? s : new Var(x);
    }
    ...
}
```

Figure 3.1: Untyped lambda calculus with the Visitor pattern.

to the expression type Exp. Like FreeVars, we call e.accept(this) to perform substitution on children. Indeed, the argument passed to the accept method does not restrict to be this and can be an arbitrary instance of LamAlg. This allows existing peer visitors to be reused. For instance, we call s.accept(new FreeVars()) to reuse previously defined FreeVars for collecting free variables from the expression s.

The implementation with the Visitor pattern has two problems: it is not modular (i.e. does not allow new language constructs to be modularly added); and requires substantial amounts of code, including code for each of the 5 language constructs for both free variables and substitution.

3.2.4 An Implementation with Object Algebras

Next, we discuss an implementation with Object Algebras shown in Figure 3.2 (full version can be found in Appendix A.2).

**Syntax** Object Algebras bypass the concrete AST representation, making it simple to model the language. The Object Algebra interface LamAlg is similar to the visitor interface except that the recursive argument has abstract type Exp, which is the same as the return type.

**Free Variables** Operations over the language are defined as Object Algebras, which are implementations of the LamAlg interface. The Object Algebra FreeVars instantiates the
Chapter 3. **EVP**: An Extensible and Expressive Visitor Framework

### Interface LamAlg<Exp>

```java
interface LamAlg<Exp> {
    Exp Var(String x);
    Exp Abs(String x, Exp e);
    Exp App(Exp e1, Exp e2);
    Exp Lit(int n);
    Exp Sub(Exp e1, Exp e2);
}
```

### Interface IFV

```java
interface IFV {
    Set<String> FV();
}
```

### Class FreeVars<Exp>

```java
class FreeVars<Exp> implements LamAlg<IFV> {
    public IFV Var(String x) {
        return () -> Collections.singleton(x);
    }
    public IFV Abs(String x, IFV e) {
        return () -> e.FV().stream()
            .filter(y -> !y.equals(x))
            .collect(Collectors.toSet());
    }
    ...
}
```

### Interface ISubst<Exp>

```java
interface ISubst<Exp> {
    Exp before();
    Exp after();
}
```

### Class SubstVar<Exp> extends IFV

```java
class SubstVar<Exp extends IFV> implements LamAlg<ISubst<Exp>> {
    String x;
    Exp s;
    LamAlg<Exp> alg;
    SubstVar(String x, Exp s, LamAlg<Exp> alg) {
        this.x = x;
        this.s = s;
        this.alg = alg;
    }
    public ISubst<Exp> Var(String y) {
        return new ISubstVar<Exp>() {
            public Exp before() {
                return alg.Var(y);
            }
            public Exp after() {
                return y.equals(x) ? s : alg.Var(y);
            }
        };
    }
    public ISubst<Exp> Abs(String y, ISubst<Exp> e) {
        return new ISubstVar<Exp>() {
            public Exp before() {
                return alg.Abs(y, e.before());
            }
            public Exp after() {
                if (y.equals(x))
                    return alg.Abs(y, e.before());
                if (s.FV().contains(y))
                    throw new RuntimeException();
                return alg.Abs(y, e.after());
            }
        };
    }
    ...
}
```

---

**Figure 3.2**: *Untyped lambda calculus with Object Algebras.*

Object Algebra interface as LamAlg<IFV>, where IFV is the interface that each construct should implement. Since IFV is a functional interface, we hereby use Java 8 lambdas for creating its instances easily. The implementation is very much like the visitor version. The difference to the visitor version is that programmers have indirect control over the traversal due to the bottom-up nature of Object Algebras. This makes the operation definition simpler by removing accept invocations but sacrifices some expressiveness. It is worth mentioning that we can instantiate the abstract type as Set<String> to make the definition of FreeVars simpler. Using IFV, however, makes FreeVars easier to be composed and reused, which we will see later in the definition of substitution. Also, the number of cases to implement can be reduced to 2 by using the traversal template provided by the Shy framework [74].

**Substitution** Modeling substitution using Object Algebras is tricky. There are two major difficulties: 1) It is hard to express the dependency on FreeVars modularly in the definition of substitution; 2) Substitution traverses the expression structure in a flexible way, and not in a purely bottom up manner. In particular, in the abstraction case, the body may not be traversed when the variable to be substituted is captured by the binder. Substitution can still be modeled with a technique similar to that employed in the definition of the predecessor function on Church numerals.

The SubstVar class contains one more field alg, which is the successor algebra instance for reconstructing the AST. Instead of just returning the expression after substitution, we
3.2 Overview

Also keep track of the original expression. The pair-like interface ISubst is defined for such purpose. This interface is critical for the definition of Abs because the body can either be substituted or not depending on whether the condition holds. As the body e is now of type ISubst<Exp>, we can call before or after for obtaining the expression before and after substitution. Also, to be able to collect free variables from expressions, we set the upper bound of type parameter Exp as IFV. This way, we can call the FV method on the expression s for collecting free variables.

The drawback of the Object Algebra implementation of substitution is that it makes implementation inefficient and complicated. Moreover, unlike the implementation of free variables, which can benefit from the Shy framework to reduce the number of cases, the definition substitution does not fit any of the traversal templates offered by the Shy framework. Thus 5 cases are needed for substitution.

3.2.5 An Implementation with EVF

The corresponding implementation of the untyped lambda calculus with EVF is given by Figure 3.4. EVF uses a Java annotation processor for generating the boilerplate code related to AST creation and various traversal templates. The Java annotation processor uses the standard javax.annotation.processing API, which is part of the Java platform. To
@Visitor interface LamAlg<Exp> {
   Exp Abs(String x, Exp e);
   Exp App(Exp e1, Exp e2);
   Exp Var(String x);
   Exp Lit(int n);
   Exp Sub(Exp e1, Exp e2);
}

interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
   default Monoid<Set<String>> m() {
      return new SetMonoid<>();
   }
   default Set<String> Var(String x) {
      return Collections.singleton(x);
   }
   default Set<String> Abs(String x, Exp e) {
      return visitExp(e).stream().filter(y -> !y.equals(x)).collect(Collectors.toSet());
   }
}

interface SubstVar<Exp> extends LamAlgTransform<Exp> {
   String x();
   Exp s();
   FreeVars<Exp> FV();
   default Exp Var(String y) {
      return y.equals(x()) ? s() : alg().Var(y);
   }
   default Exp Abs(String y, Exp e) {
      if (y.equals(x())) return alg().Abs(y, e);
      if (FV().visitExp(s()).contains(y)) throw new RuntimeException();
      return alg().Abs(y, visitExp(e));
   }
}

class FreeVarsImpl implements FreeVars<Exp>, LamAlgVisitor<Set<String>> {}
class SubstImpl implements Subst<Exp>, LamAlgVisitor<Exp> {
   String x;
   Exp s;
   public SubstImpl(String x, Exp s) { this.x = x; this.s = s; }
   public String x() { return x; }
   public Exp s() { return s; }
   public FreeVars<Exp> FV() { return new FreeVarsImpl(); }
   public GLamAlg<Exp,Exp> alg() { return new LamAlgFactory(); }
}

public class Overview {
   public static void main(String[] args) {
      LamAlgFactory alg = new LamAlgFactory();
      Exp exp = alg.Abs("y", alg.Var("x"));
      alg.Var("x"); // \(\lambda y.x\) x
      new FreeVarsImpl().visitExp(exp); // \(x\)
      new SubstImpl("x", alg.Var("y")).visitExp(exp); // \(\lambda y.x\) y
   }
}

Figure 3.4: Untyped lambda calculus with EVF

Figure 3.5: Instantiation and client code for untyped lambda calculus
3.2 Overview

interact with **EVF**, users simply annotate the standard Object Algebra interfaces with @Visitor. The companion infrastructure code will then be automatically generated at compile-time. In a modern IDE like Eclipse or IntelliJ, usually each time the code is saved, the compilation is triggered and code is generated.

**Generalized Object Algebras** EVF supports a generalized version of Object Algebras, which can also be used as external visitors. Among the generated code, the **generalized object algebra interface** is the basis around which all other generated code is centered. However, users of **EVF** do not need to write such generalized algebra interfaces directly. Instead **EVF** allows clients to encode the algebra interfaces in the old way, as done for example in lines 1-7 of Figure 3.4. The corresponding generalized object algebra interface generated for LamAlg is as follows:

```java
interface GLamAlg<Exp, OExp> {
    OExp Var(String x);
    OExp Abs(String x, Exp e);
    OExp App(Exp e1, Exp e2);
    OExp Lit(int n);
    OExp Sub(Exp e1, Exp e2);
    OExp visitExp(Exp e); // visitX() method
}
```

Note that **GLamAlg** is parameterized by two types **Exp** and **OExp**, where **Exp** is for recursive arguments and **OExp** is for return values. It replaces the return type of constructors with **OExp** and inserts a method **visitExp()** that converts **Exp** to **OExp**. The change of algebra interfaces also alters the way of encoding algebras. **FreeVars**, for instance, instantiates **OExp** to **Set<String>** while keeps **Exp** abstract and we call **visitExp()** on inner expressions of **Abs** in line 16. The **visitExp()** method allows programmers to explicitly control recursive calls, which is something that regular Object Algebras do not directly support. This makes it simple to model operations using top-down traversals like **SubstVar**. We leave the discussion on the technical details of the generalization to Section 3.3.

**Structural Traversals** Neither **FreeVars** nor **SubstVar** extends **GLamAlg** directly. Instead, they extend the generated interfaces **LamAlgQuery** and **LamAlgTransform**. Similarly to the Object Algebras based **Shy** framework [74], **EVF** supports various traversal patterns that can be used to remove boilerplate code. Note, however, that **Shy** only supports bottom-up traversals, due to the inherited limitation from standard Object Algebras. In contrast, **EVF** does not limit the traversal strategy and traversal patterns can be used in top-down operations such as **SubstVar**. In Section 3.5 we will give formal specifications of the traversal templates and introduce more forms of traversal patterns.

**Reusable Algebras via Dependencies** The support for external visitors allow algebras to reuse their peer algebras, just like **SubstVar** reuses **FreeVars**. We express such dependency by declaring a getter that returns an instance of **FreeVars** as shown in line 23 of Figure 3.4, and use **FreeVars** by calling **visitExp()** with an instance of type **Exp** (the input type) in line 29. This simple reuse mechanism improves the modularity of algebras significantly, and can be used together with OO inheritance for modularity and extensibility. More about
dependencies will be discussed in Section 3.4.

**Instantiation and Client Code** We use interfaces and default methods to define generalized object algebras for exploiting Java 8 multiple interface inheritance. As a result, we have to instantiate them as classes in order to create objects. Notice that visitExp() remains abstract in both of the algebra definitions in Figure 3.4. EVF generates an interface LamAlgVisitor, which extends GLamAlg with visitExp() implemented. Line 1 and lines 2-10 of Figure 3.5 illustrate how to instantiate FreeVars and SubstVar as external visitors. In SubstVar dependencies must be fulfilled. For example, to meet the dependency on FreeVars, we need a FreeVarsImpl instance in line 8.

**Concrete AST Representation** Different from conventional Object Algebras, the construction and interpretation of an AST are separated in EVF. Exp, sharing the same name as the type parameter used in Figure 3.5, reifies the AST and is automatically generated by EVF. Once created, the AST will reside in memory and is able to accept different algebras to traverse itself. For example, we construct an AST of form (\(\lambda y: x\) x) in line 14 and then apply two different algebras, FreeVars and SubstVar, to the AST by invoking their visitExp() method.

### 3.3 Extensible Visitors

This section presents an extensible visitor encoding based on a generalization of Object Algebras. To reduce the burden of users, EVF automatically generates the Visitor infrastructure. We illustrate the ideas by manually re-implementing the expression language example used throughout Chapter 2. EVF visitors retain the extensibility of Object Algebras, which is illustrated by extending the expression language.

#### 3.3.1 Generalized Object Algebras

Previous work [47, 48] already noted that it is possible to generalize Object Algebras, by distinguishing between input and output types in the interfaces. Such generalization provides a way to deal with dependencies in Object Algebras using delegation. EVF uses this generalization, but it adds a new twist: introducing visitX() methods (where X stands for the input types of the algebra interface). These extra methods account for a large part of the expressive power of EVF and also allow EVF to express dependencies via OO inheritance and composition instead of delegation. Recall the object algebra interface of the expression language ExpAlg shown in Figure 2.3. Its generalized form is as follows:

```java
interface GExpAlg<Exp,OExp> {
    OExp Lit(int n);
    OExp Sub(Exp x, Exp y);
    OExp visitExp(Exp t); // new method
}
```

Differently from ExpAlg, GExpAlg is parameterized by two types Exp and OExp, for distinguishing the type of recursive argument from the result type. One additional method, visitExp(), is added to the interface for converting Exp to OExp.
interface Exp {
    <O> 0 accept(Visitor<O> vis);
}

interface Visitor<O> extends GExpAlg<O, O> {
    default 0 visitExp(O t) {
        return t;
    }
}

class Factory implements Visitor<Exp> {
    public Exp Lit(int n) {
        return new Exp() {
            public <O> O accept(Visitor<O> v) {
                return v.Lit(n);
            }
        };
    }
    public Exp Sub(Exp x, Exp y) {
        return new Exp() {
            public <O> O accept(Visitor<O> v) {
                return v.Sub(x.accept(v), y.accept(v));
            }
        };
    }
}

Figure 3.6: An internal visitor implementation for GExpAlg

With GExpAlg defined, its generalized object algebras can be given:

interface Eval<Exp> extends GExpAlg<Exp, Integer> {
    default Integer Lit(int n) { return n; }
    default Integer Sub(Exp x, Exp y) {
        return visitExp(x) - visitExp(y);
    }
}

Eval implements the evaluator for the expression language, where Exp is still an abstract type parameter while 0Exp is instantiated to Integer for representing the evaluation result. Note that visitExp() needs to be called explicitly to process recursive arguments. Another obvious change to the algebra definition is that Eval is modeled as an interface rather than a class with its constructors implemented using Java 8 default methods. This is mainly for two reasons: making algebra definitions composable via Java 8 multiple interface inheritance and more importantly, retaining extensibility for algebras, which will be discussed in Section 3.3.3.

3.3.2 Instantiating Generalized Object Algebras as Visitors

As we have discussed in Section 2.5.1, conventional object algebras are closely related to internal visitors and do not have direct control over traversals. The generalized object algebras in EVF go further, allowing themselves to be instantiated as either internal or external visitors while retaining extensibility. How to instantiate the type parameter Exp and implement the visitExp() method determine whether a generalized object algebra is an internal or an external visitor.
interface Exp {
    <O> 0 accept(Visitor<O> v);
}

interface Visitor<0>
    extends GExpAlg<Exp,O> {
    default 0 visitExp(Exp t) {
        return t.accept(this);
    }
}

class Factory implements GExpAlg<Exp,Exp> {
    public Exp Lit(int n) {
        return new Exp() {
            public <O> O accept(Visitor<O> v) {
                return v.Lit(n);
            }
        };
    }
    public Exp Sub(Exp x, Exp y) {
        return new Exp() {
            public <O> 0 accept(Visitor<O> v) {
                return v.Sub(x, y);
            }
        };
    }
    public Exp visitExp(Exp t) {
        return t;
    }
}

Figure 3.7: An external visitor implementation for GExpAlg

Internal Visitors The internal version is given in Figure 3.6. The ExpAlgVisitor interface recovers the conventional Object Algebra interface ExpAlg from GExpAlg by instantiating the type parameter 0Exp the same as Exp and implementing the visitExp as an identity function. The interface Exp, used to represent the interface of the data structure to be visited concretely, contains the accept() method. The ExpAlgFactory class is an object algebra for constructing Exp instances, where the factory methods construct literals and subtractions respectively. The encoding is internal because the traversal over subexpressions (i.e. the accept() method call) is predefined inside those factory methods.

Instantiating an generalized object algebra to an internal visitor is simple:

class EvalImpl implements Eval<Integer>, Visitor<Integer> {}

The class EvalImpl implements both Eval and ExpAlgVisitor and instantiates input types the same as their corresponding output types.

External Visitors Figure 3.7 shows the external visitor encoding for GExpAlg. The infrastructure is similar to that of the internal version with the following differences. The external ExpAlgVisitor interface fixes the input type to Exp, the data structure to be visited. With this instantiation, the visitExp() method is implemented by calling the accept() method. In other words, calls to visitExp() in a generalized object algebra are essentially calls to accept() if that algebra is instantiated as an external visitor. Thus programmers are in direct control of the traversal. We can instantiate Eval as an external visitor in a way similar to an internal visitor:

class EvalImpl implements Eval<Exp>, Visitor<Integer> {}

The only difference is that its input type is bound to Exp.
3.3 Extensible Visitors

**Client Code** The code below demonstrates how to construct an expression and then evaluate it, which applies to both internal and external visitor version.

```java
Factory alg = new Factory();
Exp e = alg.Sub(alg.Lit(2), alg.Lit(1));
new EvalImpl().visitExp(e); // 1
```

Although the style of client code shown in Figure 2.4 is still applicable to internal visitors, the new style is preferable. In this new style we construct an AST through an ExpAlgFactory instance and then apply various operations to the AST by invoking the accept() method with algebra instances. It is more convenient to construct ASTs using a ExpAlgFactory than creating a mkExp-like method every time a different AST is needed. In addition, we can switch between the internal and external visitors easily.

### 3.3.3 Extensibility

Extensibility is preserved in EVF, and the novelty is that in EVF external visitors are supported as well. We can extend GExpAlg modularly in a way shown below:

```java
interface GExtAlg<Exp,OExp> extends GExpAlg<Exp,OExp> {
    OExp Neg(Exp x);
}
interface EvalExt<Exp> extends GExtAlg<Exp,Integer>, Eval<Exp> {
    default Integer Neg(Exp x) {
        return -visitExp(x);
    }
}
```

However, we need a visitor infrastructure similar to Figure 3.7 for GExtAlg so that EvalExt can be instantiated. This code is quite painful to write manually. Fortunately, EVF automatically generates all this infrastructure for extensions as well. So, if we use EVF (rather than having a manual implementation), we can simply write:

```java
@Visitor interface ExtAlg<Exp> extends ExpAlg<Exp> {
    Exp Neg(Exp x);
}
```

**Client code** The following code illustrates how to use the generated code to construct an extended expression and evaluate it using the extended algebra:

```java
class EvalExtImpl implements EvalExt<Exp>, ExtAlgVisitor<Integer> {}
ExtAlgFactory alg = new ExtAlgFactory();
Exp e = alg.Neg(alg.Sub(alg.Lit(2), alg.Lit(1)));
new EvalExtImpl().visitExp(e); // -1
```

The client code is almost the same as that of GExpAlg except that the prefix of the visitor infrastructure is changed from ExpAlg to ExtAlg.

### 3.3.4 Object Algebra Interface Generalization

It is cumbersome for users to directly write down the generalized object algebra interfaces, especially when multiple sorts are needed. This motivates us to let EVF automatically translate a conventional object algebra interface into its generalized version. Figure 3.8 formalizes the translation.
• Syntax of object algebra interfaces

\[ L ::= \text{interface } I_0 \text{ extends } \overline{I} \{ \overline{C} \} \quad \text{object algebra interfaces} \]

\[ C ::= X \, c(\overline{T} \overline{x}) ; \quad \text{constructors} \]

\[ I ::= A \overline{x} ; \quad \text{interface types} \]

\[ T ::= X \mid \text{int} \mid \text{boolean} \mid \ldots \quad \text{argument types} \]

• Translation scheme

\[
\begin{align*}
\text{genVisitX}_{I_0}(I) &= \text{interface } I \text{ extends } I \{ \overline{C} \} \text{ genVisitX}_{\overline{I}}(I_0) \text{ methods one for each new input carrier type.} \\
\text{allX}_{\overline{I}}(\text{interface } I_0 \text{ extends } \overline{I} \{ \overline{C} \}) &= \{ \text{returntype}(C) \mid C \in \overline{C} \} \cup \bigcup_{I \in \overline{I}} \text{allX}_{\overline{I}}(\text{AT}(I)) \text{ collects all input carrier types from the algebra interface hierarchy; finally,} \\
\text{newX}_{\overline{I}}(\text{interface } I_0 \text{ extends } \overline{I} \{ \overline{C} \}) &= \text{allX}_{\overline{I}}(\text{AT}(I_0)) \setminus \bigcup_{I \in \overline{I}} \text{allX}_{\overline{I}}(\text{AT}(I)) \text{ collects input carrier types that are not introduced by super algebra interfaces; finally,} \\
\text{genVisitX}_{\overline{I}}(I) &= \text{genVisitX}_{\overline{I}}(I) \text{ generates visitX() methods one for each new input carrier type.} \\
\end{align*}
\]

Figure 3.8: Generating generalized object algebra interface

Syntax of Object Algebra Interface  We first give the grammar of conventional object algebra interfaces. The metavariable \( A \) ranges over object algebra interface names; \( X \) ranges over type parameters; \( c \) and \( x \) range over names. We write \( \overline{I} \) as shorthand for \( I_1, \ldots, I_n, \overline{x} \) for \( X_1, \ldots, X_n; \overline{C} \) for \( C_1 \ldots C_n \) (no commas in between). We abbreviate operations on pairs of sequences similarly, writing \( \overline{T} \overline{x} \) for \( T_1 x_1, \ldots, T_n x_n \), where \( n \) is the length of \( \overline{T} \) and \( \overline{x} \). Following standard practice, we assume an algebra interface table (AT) that maps algebra interface \( I \) to their declaration \( L \).

Translation Scheme  Translation rules are defined with the semantic brackets ([ ]). The bracket notation \([f(A) \mid A \in \overline{A}]\) denotes that the function \( f \) is applied to each element in the list \( \overline{A} \) sequentially to generate a new list. The curly brace notation \( \{f(A) \mid A \in \overline{A}\} \) is similar to the bracket notation except that it collects a set of elements while preserving their order.

The fundamental step of the generalization is to separate input carrier types from the type parameter list. We say that a type parameter is an input carrier type if it is a return type of any constructor from the algebra interface hierarchy. These type parameters are special because they have corresponding output type and \( \text{visitX}() \) method.

The translation scheme consists of three main steps. First, we find out all input carrier types and augment the type parameter list with their corresponding output carrier types. Second, the return types of the constructors are replaced by output carrier types. Last, the \( \text{visitX}() \) methods are generated for new input carrier types.

Auxiliary Definitions  The translation scheme relies on some auxiliary definitions: \( \text{returntype} \) gets the return type of a constructor (considered as an input carrier type); \( \text{allX}_{\overline{I}} \) collects all input carrier types from the algebra interface hierarchy; \( \text{newX}_{\overline{I}} \) collects input carrier types that are not introduced by super algebra interfaces; finally, \( \text{genVisitX}_{\overline{I}} \) generates \( \text{visitX}() \) methods one for each new input carrier type.
3.4 Modular Dependencies

An important drawback of conventional Object Algebras is that it is hard to modularly express dependencies. Fortunately, external visitors make the reuse of peer algebras easy through standard OO composition mechanisms. In this section, we explain EVF’s approach to defining modular dependent algebras.

Consider the pretty printer shown in Figure 2.5 again. Its modular version in EVF is given in Figure 3.9. We separate the original implementation into two algebras MPrint and IsLit, and makes MPrint depend on IsLit. The dependency on IsLit is expressed through declaring a getter method isLit() that returns an instance of IsLit. To use IsLit we call the getter first and then invoke the visitExp() method with an instance of Exp.

Compared to Figure 2.5 the code in Figure 3.9 is simpler and more modular. As IsLit is decoupled, it can be reused elsewhere. Moreover, MPrint in Figure 3.9 is easier to use than that of Figure 2.5. Its result is a direct String, hence an extra selection on the tuple (print()) is no longer needed.

Client Code Dependencies should be fulfilled when instantiating algebras. This could simply be done through returning corresponding instantiated algebra instances:

```java
class IsLitImpl implements IsLit<Exp>, ExpAlgVisitor<Boolean> {}
class MPrintImpl implements MPrint<Exp>, ExpAlgVisitor<String> {
    public IsLit<Exp> isLit() {
        return new IsLitImpl();
    }
}
```

Alternatively, if memory is a concern, one can declare a static field, initialize it using an instance of IsLitImpl and use that field to fulfill the dependency.

To test our printer implementation, we print out the expression 2 − 1 that subtracts itself:

```java
Exp e = alg.Sub(alg.Lit(2), alg.Lit(1));
```
Extensibility  More importantly, algebras with dependencies are still extensible. We can easily support pretty printing for the extended expression:

```java
interface IsLitExt<Exp> extends GExtAlg<Exp, Boolean>, IsLit<Exp> {
    default Boolean Neg(Exp x) {
        return false;
    }
}
interface MPrintExt<Exp> extends GExtAlg<Exp, String>, MPrint<Exp> {
    @Override IsLitExt<Exp> isLit(); // dependency refinement
    default String Neg(Exp x) {
        return "-" + visitExp(x);
    }
}
```

Both IsLit and Print are extended. Through covariant return type refinement on getter isLit(), we update the dependency from IsLit to its subtype IsLitExt. Errors would occur on algebra instantiation if one forgets to do type-refinements on dependencies in extensions. The @Override annotation makes sure that we are not adding a new dependency but refining an inherited one. As shown by Wang and Oliveira, type-refinements are often needed to achieve type-safe extensibility [68].

3.5 Boilerplate Traversals

AST traversals often contain a lot of boilerplate code. EVF deals with boilerplate traversals in Shy [74]. Notably, and unlike Shy, boilerplate traversals in EVF are not restricted to be bottom-up. We have seen how traversal templates eliminate boilerplate code in Section 3.2.5. In this section, we give formal definitions of these traversal templates and additionally introduce a novel type of traversal pattern.

3.5.1 Queries with Default Values

Inspired by wildcard patterns in functional languages, EVF supports a new type of queries with default values. This template gives each case an implementation using the client-supplied default value, which would be handy for defining algebras with a lot of cases sharing the same behavior. For example, it would be tedious if we define the IsLit algebra for the untyped lambda calculus in a way similar to Figure 3.9. Rather, we can use this template because only the Lit() case is interesting and other cases share the same behavior:

```java
interface IsLit<Exp> extends LamAlgDefault<Exp, Boolean> {
    default Zero<Boolean> m() { return () -> false; }
    default Boolean Lit(int n) { return true; }
}
```

Instead of giving each of those boring cases an implementation manually, we use the LamAlgDefault template to deal with them. With this template, we only need to supply a default value (false) once via implementing the m() method.

Now we give the template of queries with default values formally. Given an Object Algebra interface A, let $\mathcal{X}$ denote the input types of A where $\mathcal{X} = \mathcal{X}_{in}(\mathcal{A}(\mathcal{A}))$. The template
can be defined as below:

```java
interface Zero<O> {
    O empty();
}
```

```java
interface A0Default<X0,O> extends A0<X0,O> {
    Zero<O> m();
    default O c(T x) { return m().empty(); }
}
```

The functional interface `Zero` is the default value provider on which `Default` depends. `Default` implements all cases of an algebra interface simply through returning that default value. The default value is obtained by invoking `m().empty()`. The implementation of `m()` is delayed to concrete algebras that use the `Default` template, for allowing different default values to be specified.

### 3.5.2 Queries by Aggregation

Another form of query traverses the whole AST and aggregates a value. Recall the definition of `FreeVars` shown in Figure 3.4. It uses the template `LamAlgQuery`. The template for queries by aggregation is given below:

```java
interface Monoid<O> extends Zero<O> {
    O join(O x, O y);
}
```

```java
interface A0Query<X0,O> extends A0<X0,O> {
    Monoid<O> m();
    default O c(T x) {
        return m().empty();
        if \( \forall T \in T \land T \in X_0 \),
        Stream.of([\( \text{visit}(T(x)|T \in T \land T \in X_0) \)]).reduce(m().empty(),m()::join); otherwise.
    }
}
```

Query gives different implementation to a constructor according to whether it is a leaf or an internal node. If it is a leaf (i.e. no argument of any input carrier types), the result is `m().empty()`; otherwise corresponding `visitX()` methods get called on recursive arguments and their results are combined using `m().join()`.

In the definition of `FreeVars`, the generic `SetMonoid` class is used for fulfilling the `m()` dependency:

```java
class SetMonoid<T> implements Monoid<Set<T>> {
    public T empty() { return Collections.emptySet(); }
    public T join(Set<T> x, Set<T> y) {
        return Stream.concat(x.stream(), y.stream()).collect(toSet());
    }
}
```

where `empty()` returns an empty set and `join()` is the union of two sets.

### 3.5.3 Transformations

Transformations are operations that transform an AST to another AST. Transformations use a factory to construct another AST that can be further transformed or consumed.
Recall the definition of \texttt{SubstVar} shown in Figure 3.4. It uses the transformation template \texttt{LamAlgTransform} for eliminating boilerplate code. The general template for transformations is given below:

```java
interface A0Transform<X0> extends G0<X0,X0> {
    G0<X0,X0> alg();
    default X c(T x) { return alg().c(visitT(T,x)); }
}
```

In \texttt{Transform} the output carrier types are the same as input carrier types, reflecting the essence of a transformation. We need an additional auxiliary definition \texttt{visitT}, which transforms an argument only when it is of any input carrier types:

\[
\texttt{visitT}(T,x) = \begin{cases} 
\texttt{visit}(x) & \text{if } T \in \mathcal{X}, \\
    x & \text{otherwise.}
\end{cases}
\]
In the previous chapter, we presented the backbone of EVF, including the generalized Object Algebras and traversal templates. In this chapter, we introduce the pattern matching support in EVF. Pattern matching provides a concise way to implement operations that inspect and manipulate ASTs extensively. EVF generates a set of classes that provide support for a form of pattern matching. The approach emulates Zenger and Odersky's extensible algebraic datatypes with defaults using a fluent interface style embedded DSL (EDSL).

### 4.1 Introduction

Very often operations used in compilers, interpreters or program analysis, require deep case analysis on an AST structure. In functional programming languages ASTs are modeled as algebraic datatypes. With built-in support for pattern matching, one can analyze an AST and perform transformations on it in a concise way. However, algebraic datatypes as well as operations defined over them are typically not extensible, which becomes a problem when new constructs are introduced. On the other side of the coin, ASTs are represented as class hierarchies in OO languages, which are harder to explore. Nevertheless, pattern matching can be simulated through encodings in OOP. The Visitor pattern is one of the well-known approaches. Other common approaches include OO decomposition and type-tests/type-casts [16]. In contrast to OO decomposition approach which scatters the definition of an operation around the class hierarchy, the Visitor pattern centralizes the definition in one class. Neither runtime introspection nor casts are needed when using the Visitor pattern, as objects are recognized through dynamic dispatching. With these appealing properties, the Visitor pattern is often used in compiler phases. However, similar to the functional approach, the Visitor pattern suffers from the EP. Moreover, there is a high notation overhead in using the Visitor pattern.

Fortunately, the encoding of generalized Object Algebras makes it possible to introduce new variants modularly. But the verbosity of the encoding is not reduced but increased, especially when modeling operations that need nested case analysis. As one visitor performs only the top-level case analysis on an AST, nested case analysis is simulated through a series of ad-hoc visitors. Such ad-hoc visitors are normally defined anonymously so that the information extracted by existing visitors is right in scope. Un-
Chapter 4. Extensible Pattern Matching in \texttt{EVF}

Fortunately, \texttt{EVF} in its basic form does not support modular anonymous algebras. Although anonymous algebras can be simulated by named algebras with necessary scope information passed explicitly, the resulting implementation is long and complicated.

To tackle this problem, we design a pattern matching EDSL for creating modular anonymous algebras to perform nested case analysis. We integrate this EDSL in \texttt{EVF} and let \texttt{EVF} automatically generate the infrastructure during annotation processing. The EDSL is inspired from two existing works: extensible algebraic datatypes with default (EADD) \cite{EADD} and the derive4j project \cite{derive4j}. Our contribution is to combine these ideas to design an EDSL on top of extensible external visitors that emulates extensible pattern matching with defaults. Algebras defined with such infrastructure are concise and require no instantiation and fewer dependencies, significantly reducing the burden of client users.

4.2 Example: Structural Equality

In this section, we give an overview of the pattern matching support in \texttt{EVF}. Meanwhile, for comparison, we review pattern matching in Scala using case classes and in Java using the \texttt{Visitor} pattern. We use structural equality as a running example. Structural equality is interesting because it checks whether two data structures are constructed consistently, where case analysis is needed on both structures being compared.

From Section 4.2.1 to Section 4.2.4, we implement structural equality for the expression language and its extended version respectively in Scala, the \texttt{Visitor} pattern, \texttt{EVF}, and \texttt{EVF} with pattern matching respectively.

4.2.1 A Scala Implementation

Scala is a language that unifies both object-oriented and functional paradigms. There are two mechanisms to support pattern matching in Scala: \texttt{case classes} and \texttt{extractors}. Here is how we model the expression language using case classes:

```scala
trait Exp

case class Lit(n: Int) extends Exp

case class Sub(x: Exp, y: Exp) extends Exp
```

The base trait \texttt{Exp} is an abstract representation of expressions. Concrete expressions, literals and subtractions, are defined as case classes. The \texttt{case} modifier automatically generates the \texttt{apply()} and \texttt{unapply()} methods for the class it decorates, allowing convenient constructions and deconstructions on objects. Typically, base classes or traits are decorated with a \texttt{sealed} modifier for allowing Scala compiler to perform exhaustiveness checking on pattern matching. We intend not to do so on \texttt{Exp} for the purpose of extensibility, which will be discussed later.

With these definitions, we can now define structural equality, as shown in Figure 4.1. The implementation is quite straightforward. We pattern match on the two expressions being compared simultaneously using a pair. If they are of the same pattern, we recursively compare their components. The result is a \texttt{true} only when their components are also constructed in the same way. In the last clause we use the wildcard pattern (an underscore) to capture all cases when \texttt{x} and \texttt{y} are of different forms at once.
trait Eq {
  def eq(x: Exp, y: Exp): Boolean = (x, y) match {
    case (Lit(m), Lit(n)) => m == n
    case (Sub(x1, x2), Sub(y1, y2)) => eq(x1, y1) && eq(x2, y2)
    case _ => false
  }
}

Figure 4.1: Structural equality using case classes

Note that we put the definition of eq inside the trait Eq. Together with non-sealed Exp, we are able to support structural equality on an extended expression language, modularly.

For example, to support structural equality on the expression language with negations, we define a new case class Neg and implement structural equality in a new trait EqExt:

```scala
case class Neg(x: Exp) extends Exp

trait EqExt extends Eq {
  override def eq(x: Exp, y: Exp): Boolean = (x, y) match {
    case (Neg(x), Neg(y)) => eq(x, y)
    case _ => super.eq(x, y)
  }
}
```

By inheriting Eq, the overridden definition of eq in EqExt only needs to complement the comparison of two negations. The comparison of two existing forms of expressions as well as the comparison of a negation with another form of expression delegates to the super method. Indeed, the implementation follows EADDs. EADDs force a default case in each operation so that new variants are subsumed by that case. Existing operations can hence be reused for extended datatypes.

Though combining Scala case classes with EADDs seem to solve the EP, the solution has some limitations:

- **Lost exhaustiveness checking**: The exhaustiveness checking on pattern matching is traded for extensibility by removing the `sealed` modifier. As a result, a `MatchError` exception may be thrown at runtime, indicating that there is no pattern that matches the given object.

- **No good defaults**: EADDs are a nice solution when there is a good default for an operation (e.g. structural equality). But when there is not (e.g. evaluation), a default has to be invented. EADDs make these operations automatically work for extended datatypes. A problem is that in later extensions programmers may forget to override the default, and the compiler gives no warning. As a result, unexpected behavior may happen. For example, if we use Eq to compare two identical negations, the result is an unexpected `false`.

- **No distinct datatypes**: New data variants are added to the original datatype (Exp in our example). Consequently, operations like desugaring can not be precisely represented at the type level. For example, an operation that rewrites `Neg n` to `Sub 0 n` has type `Exp -> Exp`, which does not reflect the fact that all negations have been
Chapter 4. Extensible Pattern Matching in EVF

eliminated after the operation is applied. It would be much clearer if it is of type
Exp \to Exp', where Exp' contains every construct from Exp except for negations.

- **Not supported in mainstream OO languages**: Mainstream OO languages such as
Java, C++, and C# do not support first-class pattern matching.

### 4.2.2 A Java Implementation Using the Visitor Pattern

For ordinary OO languages, algebraic datatypes and pattern matching can be encoded
using the Visitor pattern. However, one visitor does only one layer of “pattern matching”.
To encode operations that need nested case analysis like structural equality, we need
extra visitors.

Figure 4.2 gives the Visitor implementation of structural equality. The return type of
the concrete visitor \( Eq \) is the functional interface \( IEq \), for capturing the second expression.
For creating instances of \( IEq \) conveniently, Java 8 lambdas are used. The internal rep-
resentation of an expression is revealed when a particular visit method is invoked. For
example, when implementing the \( \text{Sub()} \) method of \( Eq \), we know that the first expression
is a subtraction whose subexpressions are \( x_1 \) and \( x_2 \). To further deconstruct the second
expression, the lambda argument \( e \), we create an anonymous inner visitor. Then we can
compare the two expressions inside the inner visitor, as their internal representations
are both known now. When the second expression is also a subtraction, we recursively
compare its subexpressions \( y_1 \) and \( y_2 \) against to \( x_1 \) and \( x_2 \); otherwise a false is returned.
Indeed, the outer visitor and the inner visitor are mutually recursive: the outer visitor
uses the inner visitor to deconstruct the second expression and the inner visitor uses the
outer visitor to compare subexpressions.

Compared to the Scala approach, the Visitor pattern introduces some notation over-
head. The need for nested case analysis exacerbates the verbosity. Worse still, the
implementation in Figure 4.2 is not extensible. To support structural equality for the ex-
tended expression language, we have to either modify the Visitor interface and the \( Eq \) class
or define another class through copying and pasting code from \( Eq \). Neither approaches
are modular.

### 4.2.3 A Java Implementation Using EVF

Fortunately, the Visitor implementation can be made extensible with EVF. But we
cannot directly translate the code in Figure 4.2 to EVF, as extensible anonymous visitors
are not supported. Nevertheless, anonymous visitors can be simulated by top-level alge-
bras as long as information needed in scope is provided explicitly. Figure 4.3 modularizes
structural equality using basic features of EVF. Two extra top-level algebras \( EqLit \) and
\( EqSub \) are defined for simulating the functionality of anonymous visitors in Figure 4.2. \( Eq \)
implements the constructors by delegating to \( EqLit \) and \( EqSub \). Now that \( EqLit \) and \( EqSub \) are
defined outside the scope of constructor definitions in \( Eq \), components of the first expres-
sion need to be passed to the auxiliary algebras. Functional interfaces \( IEqLit \) and \( IEqSub \)
are defined for such purpose. Of course, dependencies, including mutual dependencies
interface IEq {
    boolean eq(Exp e);
}

class Eq implements Visitor<IEq> {
    public IEq Lit(int m) {
        return e -> e.accept(new Visitor<Boolean>() { // encoding case analysis
            public Boolean Lit(int n) {
                return m == n;
            }
            public Boolean Sub(Exp x1, Exp x2) {
                return false;
            }
        });
    }
    public IEq Sub(Exp x1, Exp x2) {
        return e -> e.accept(new Visitor<Boolean>() {
            public Boolean Lit(int n) {
                return false;
            }
            public Boolean Sub(Exp y1, Exp y2) {
                return x1.accept(Eq.this).eq(y1) &&
                        x2.accept(Eq.this).eq(y2);
            }
        });
    }
}

Figure 4.2: Structural equality using the Visitor pattern

between Eq and EqSub, have to be declared explicitly.

To actually use Eq, we need an additional instantiation step:

class EqImpl implements Eq<Exp>, ExpAlgVisitor<IEq<Exp>> {
    class EqLitImpl implements EqLit<Exp>, ExpAlgVisitor<IEqLit> {}
    class EqSubImpl implements EqSub<Exp>, ExpAlgVisitor<IEqSub<Exp>> {
        public EqImpl eq() {
            return new EqImpl();
        }
        public EqLit<Exp> eqLit() {
            return new EqLitImpl();
        }
        public EqSub<Exp> eqSub() {
            return new EqSubImpl();
        }
    }
}

Eq as well as its two auxiliary algebras EqLit and EqSub are instantiated as classes by mixing
in ExpAlgVisitor and fulling the dependencies.

The implementation in EVF is extensible in that structural equality for the extended
expression language can be incrementally defined without modifying the original imple-
mentation. Figure 4.4 presents the extended version built on top of what we have in
Figure 4.3. Through extending Eq, we only need to implement the Neg() method. To
compare negations with other kinds of expressions, we define a new auxiliary algebra
EqNeg. EqExt delegates the implementation of Neg() to EqNeg via the new dependency eqNeg
(). All existing auxiliary algebras, EqLit and EqSub, are also extended for the comparison
with negations. Moreover, existing dependencies should be refined to the corresponding
extended version.
interface IEq<Exp> {
    boolean eq(Exp e);
}

interface Eq<Exp> extends GExpAlg<Exp, IEq<Exp>> {
    EqLit<Exp> eqLit(); // simple dependency
    EqSub<Exp> eqSub(); // mutual dependency

    default IEq<Exp> Lit(int n) {
        return e -> eqLit().visitExp(e).eq(n);
    }
    default IEq<Exp> Sub(Exp x, Exp y) {
        return e -> eqSub().visitExp(e).eq(x, y);
    }
}

interface IEqLit {
    boolean eq(int n);
}

interface EqLit<Exp> extends GExpAlg<Exp, IEqLit> {
    default IEqLit Lit(int m) {
        return n -> m == n;
    }
    default IEqLit Sub(Exp x, Exp y) {
        return n -> false;
    }
}

interface IEqSub<Exp> {
    boolean eq(Exp x, Exp y);
}

interface EqSub<Exp> extends GExpAlg<Exp, IEqSub<Exp>> {
    Eq<Exp> eq(); // mutual dependency

    default IEqSub<Exp> Lit(int n) {
        return (y1, y2) -> false;
    }
    default IEqSub<Exp> Sub(Exp x1, Exp x2) {
        return (y1, y2) -> eq().visitExp(x1).eq(y1) && eq().visitExp(x2).eq(y2);
    }
}
4.2 Example: Structural Equality

```java
interface EqExt<Exp> extends GExtAlg<Exp, IEq<Exp>>, Eq<Exp> {
    EqNeg<Exp> eqNeg(); // new dependency
    @Override EqLitExt<Exp> eqLit(); // refined dependency
    @Override EqSubExt<Exp> eqSub(); // refined dependency

    default IEq<Exp> Neg(Exp x) {
        return e -> eqNeg().visitExp(e).eq(x);
    }
}
interface IEqNeg<Exp> {
    boolean eq(Exp e);
}
interface EqNeg<Exp> extends GExtAlg<Exp, IEqNeg<Exp>> {
    EqExt<Exp> eq();
    default IEqNeg<Exp> Lit(int n) {
        return e -> false;
    }
    default IEqNeg<Exp> Sub(Exp x, Exp y) {
        return e -> false;
    }
    default IEqNeg<Exp> Neg(Exp x) {
        return e -> eq().visitExp(x).eq(e);
    }
}
interface EqLitExt<Exp> extends GExtAlg<Exp, IEqLit>, EqLit<Exp> {
    default IEqLit Neg(Exp x) {
        return n -> false;
    }
}
interface EqSubExt<Exp> extends GExtAlg<Exp, IEqSub<Exp>>, EqSub<Exp> {
    @Override EqExt<Exp> eq();
    default IEqSub<Exp> Neg(Exp x) {
        return (y1, y2) -> false;
    }
}
```

Figure 4.4: Extended structural equality using $EVF$
interface Eq<Exp> extends GExpAlg<Exp, IEq<Exp>> {
    ExpAlgMatcher<Exp, Boolean> matcher();
}

default IEq<Exp> Lit(int m) {
    return e -> matcher()
        .Lit(n -> m == n)
        .otherwise(() -> false)
        .visitExp(e);
}

default IEq<Exp> Sub(Exp x1, Exp x2) {
    return e -> matcher()
        .Sub(y1 -> y2 -> visitExp(x1).eq(y1) && visitExp(x2).eq(y2))
        .otherwise(() -> false)
        .visitExp(e);
}

Although the implementation in EVF is extensible, it is even more verbose and less readable than the Visitor version due to the lack of support for anonymous algebras. The more constructs there are, the worse the situation is. This motivates us to develop a pattern matching EDSL, which allows us to create extensible anonymous algebras conveniently.

4.2.4 A Final Solution Using EVF with Pattern Matching

Figure 4.5 refactors structural equality in EVF using the pattern matching EDSL. Compared to Figure 4.3 the implementation is very compact. Like the Visitor implementation shown in Figure 4.2, auxiliary algebras are anonymously constructed right inside the constructor definition. Only one simple dependency on the generated ExpAlgMatcher interface is needed because all the auxiliary algebras in Eq return a Boolean value. To construct an anonymous algebra, we first call matcher(), then implement different cases using lambda expressions, and finally pass the expression to be matched to visitExp(). Now that the auxiliary algebra is created in the scope of the constructor, we can directly compare the components of the two expressions. For example, when the two expressions are both subtractions, we recursively compare their subexpressions using visitExp() without the need of mutual dependencies. Boring cases, dealt with otherwise(), return false by default. The new implementation is a big improvement regarding readability over the approach presented in Figure 4.3.

Eq implemented using the pattern matching EDSL is easier to instantiated because there is only one top-level algebra with only one dependency:

class EqImpl implements Eq<Exp>, ExpAlgVisitor<IEq<Exp>> {
    public ExpAlgMatcher<Exp, Boolean> matcher() {
        return new ExpAlgMatcherImpl<>();
    }
}

The dependency on ExpAlgMatcher is simply fulfilled by returning an instance of the generated class ExpAlgMatcherImpl.

More benefits of this style emerge when extensions are needed. We can support
4.2 Example: Structural Equality

<table>
<thead>
<tr>
<th></th>
<th>Exhaustiveness</th>
<th>Extensibility</th>
<th>Distinct Types</th>
<th>Deep Matches</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sealed case classes</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>10</td>
</tr>
<tr>
<td>Open case classes</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>10</td>
</tr>
<tr>
<td>Visitor</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>26</td>
</tr>
<tr>
<td>EVF</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>36</td>
</tr>
<tr>
<td>EVF with patterns</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 4.1: Pattern matching support in sealed case classes, open case classes, Visitor, EVF, and EVF with patterns: fully supported ●, partially supported ○, not supported □

structural equality for the extended expression language easily:

```scala
interface EqExt<Exp> extends GExtAlg<Exp, IEq<Exp>>, Eq<Exp> {
    @Override ExtAlgMatcher<Exp, Boolean> matcher();

    default IEq<Exp> Neg(Exp x) {
        return e -> matcher()
            .Neg(y -> visitExp(x).eq(y))
            .otherwise() -> false
            .visitExp(e);
    }
}
```

By reusing `Eq` in Figure 4.5, we simply complement the `Neg()` case. The return type of `matcher()` in the extension is refined to `ExtAlgMatcher<Exp, Boolean>`, which is a generated subtype of `ExpAlgMatcher<Exp, Boolean>` used in Figure 4.5. Note that the type-refinement is essential for being able to define the `Neg()` case when using `matcher()`. All existing anonymous algebras inherited from `Eq` still work, as the new case is subsumed by the otherwise clause correctly.

4.2.5 Discussion

Table 4.1 summarizes pattern matching support of sealed case classes, open case class, the Visitor pattern, EVF and EVF with patterns. We do the comparison from five perspectives: exhaustiveness, extensibility, distinct types and deep matches and SLOC (source lines of code). Exhaustiveness prevents pattern matching failure from happening at runtime; Extensibility allows new data variants to be added and existing operations to be adapted for handling new variants. Distinct types require new variants to be introduced in a separate datatype; Deep matches allow pattern matching on multiple objects simultaneously or nested pattern matching on a single object conveniently; SLOC measures the simplicity of a solution. We discuss the pattern matching support for each approach according to this table.

**Case Classes** For case classes, exhaustiveness and extensibility can not be obtained both at the same time. Scala compiler performs exhaustiveness checking only on sealed case classes by statically knowing all cases of a datatype. However, once decorating a class or a trait with the `sealed` modifier, its subclasses can no longer be defined in a separate file, breaking the separate compilation requirement of the EP. By removing `sealed` we trade exhaustiveness for extensibility. Combining with EADDs, operations using pattern matching are retroactive. But new variants are defined over a single datatype.
Scala has concise syntax for pattern matching and supports deep matches, leading to the minimal SLOC among the approaches.

**The Visitor Pattern** As we have discussed extensively, the classic Visitor pattern is not extensible. Pattern matching over multiple arguments and nested pattern matching are simulated through a series of visitors, which is not as convenient as Scala approach. Nevertheless, exhaustiveness is guaranteed - a concrete visitor cannot be a class unless it implements every method exposed by the visitor interface. Without considering extensibility, distinct types are not very interesting as we can define a new datatype and operations over it through duplicating existing code.

**EVF** The two approaches of EVF, namely top-level algebras and anonymous algebras constructed via the EDSL, are both based on external visitors, inheriting the exhaustiveness property. But different from the Visitor pattern, they are extensible. Compared to the open case class approach, EVF has three main advantages. Firstly, EADDs are used as an idiom in open case classes whereas a part of the syntax of the pattern matching EDSL in EVF. The difference is that a trailing call on `otherwise()` for setting the default value is mandatory in EVF. Otherwise the value to be matched cannot be supplied. Secondly, unlike open case classes, users of EVF have a choice when there is no good default: one can switch back to named algebras at the cost of verbosity. Thirdly, EVF supports distinct types, which differentiates the initial and extended system clearly. New variants are introduced in a separate algebra interface (e.g. `ExtAlg`) and a new set of operations is defined for this algebra interface. Therefore, the mismatch between the AST and the operation will be captured by the compiler. This separation also allows desugaring to be defined appropriately:

```java
interface Desugar<S, T> extends GEExtAlg<S, T> {
    GEExpAlg<T, T> alg();

    default T Lit(int n) {
        return alg().Lit(n);
    }
    default T Sub(S x, S y) {
        return alg().Sub(visitExp(x), visitExp(y));
    }
    default T Neg(S x) {
        return alg().Sub(alg().Lit(0), visitExp(x));
    }
}
```

The resulting expression is constructed via `GEExpAlg`, which contains no negations. The SLOC has been reduced significantly by using the pattern matching EDSL but is still larger than case classes approach since deep matches are not well supported.

### 4.3 The Design of Pattern Matching EDSL

This section presents the design and the implementation of the pattern matching EDSL. When a user annotates an object algebra interface, say `A`, with `@Visitor`, a pattern
4.3 The Design of Pattern Matching EDSL

matching infrastructure will be generated automatically after annotation processing. Figure 4.6 gives a high-level overview of the infrastructure generated for A. We will discuss the functionality and implementation details of the components shown in the figure one by one, based on the generated code for ExpAlg.

**Mapper**  A mapper interface collects signatures of constructors defined in an algebra interface and represents them as curried functions. For example, ExpAlgMapper collects constructors from ExpAlg:

```java
interface ExpAlgMapper<Exp,O> {
    Function<Integer,O> Lit();
    Function<Exp,Function<Exp,O>> Sub();
}
```

Constructors of arity greater than one such as Sub() are represented as nested Functions.

**Matcher**  A matcher interface exposes fluent APIs for constructing anonymous algebras. If one wants to use the pattern matching in an algebra definition, it is the interface that she should declare a dependency on. The matcher for ExpAlg is shown below:

```java
interface ExpAlgMatcher<Exp,O> {
    ExpAlgMatcher<Exp,O> Lit(Function<Integer,O> f);
    ExpAlgMatcher<Exp,O> Sub(Function<Exp,Function<Exp,O>> f);
    GExpAlg<Exp,O> otherwise(Supplier<O> f);
}
```

ExpAlgMatcher contains two fluent setters Lit() and Sub() and an additional method otherwise(). The implementation of ExpAlgMatcher should have states for recording the forthcoming constructor definitions. A fluent setter takes a lambda function and returns the ExpAlgMatcher itself for allowing consecutive setter calls. A trailing otherwise() call gives a default implementation to undefined constructors. This way, an anonymous algebra can be constructed from the states recorded.

**Applier**  An applier converts lambda representation of constructors into an object algebra, which implements the algebra interface by delegating to the corresponding lambda function provided by the mapper dependency:

```java
interface ExpAlgApplier<Exp,O> extends GExpAlg<Exp,O> {
    ExpAlgMatcher<Exp,O> mapper();
}
```
default O Lit(int n) {
    return mapper().Lit().apply(n);
}
default O Sub(Exp x, Exp y) {
    return mapper().Sub().apply(x).apply(y);
}

Later on, an applier will be instantiated as an external visitor whose instance is returned by otherwise().

MatcherImpl To easily fulfill the dependencies on matchers, a generic matcher implementation is generated. Indeed, it implements both a matcher and a mapper:

class ExpAlgMatcherImpl<O> implements ExpAlgMatcher<Exp,O>, ExpAlgMapper<Exp,O> {
    private Function<Integer,O> Lit = null;
    private Function<Exp,Function<Exp,O>> Sub = null;

    public Function<Integer,O> Lit() {
        return Lit;
    }
    public Function<Exp,Function<Exp,O>> Sub() {
        return Sub;
    }
    public ExpAlgMatcher<Exp,O> Lit(Function<Integer,O> f) {
        Lit = f; return this;
    }
    public ExpAlgMatcher<Exp,O> Sub(Function<Exp,Function<Exp,O>> f) {
        Sub = f; return this;
    }
    public GExpAlg<Exp,O> otherwise(Supplier<O> f) {
        if (Lit == null) Lit = n -> f.get();
        if (Sub == null) Sub = x -> y -> f.get();

    class ExpAlgApplierImpl implements ExpAlgApplier<Exp,O>, ExpAlgVisitor<O> {
        public ExpAlgMapper<Exp,O> mapper() {
            return ExpAlgMatcherImpl.this;
        }
    }
    return new ExpAlgApplierImpl();
}

ExpAlgMatcherImpl contains fields for storing constructor definitions which are initialized as null. Then getters and setters from ExpAlgMapper and ExpAlgMatcher are implemented using these fields. The otherwise() method uses the Supplier passed in to construct closures for all unset (null) fields. Then a local class ExpAlgApplierImpl is defined for creating an external visitor from the fields recorded. ExpAlgApplierImpl instantiates ExpAlgApplier and realizes the dependency on ExpAlgMapper using ExpAlgMatcherImpl. Finally, an instance of ExpAlgApplierImpl is returned from the otherwise() method on which we call the visitExp() method for pattern matching an instance of Exp.
To illustrate the applicability of \textit{EVF} we conduct a large case study. The case study refactors a large number of interpreters from the “Types and Programming Languages” (TAPL) book [51]. It is worth pointing out that the original code, written in OCaml, is non-modular: interpreters for each chapter are defined from scratch, with lots of duplicated code from previous chapters. Moreover, the code makes use of standard OCaml features such as pattern matching, and due to the small-step semantics style [53] used in the book, most code requires top-down traversals. Also many functions have non-trivial dependencies on other functions, making them quite hard to modularize. Translating code with such features into Object Algebras would pose immediate challenges. Using \textit{EVF} the translation is straightforward, and the code in \textit{EVF} is modular and reusable. Last but not least, the diversity of languages makes it a comprehensive case study that covers nearly every aspect of \textit{EVF}, revealing its expressive power. This chapter also gives the preliminary performance measurements on \textit{EVF} with respect to the standard \textit{Visitor} pattern and \textit{Runabout} [25].

5.1 Overview

Terms and types are the main data structures for modeling programming languages on which two families of operations are defined. Such operations include: interpreters and type-checkers for terms; type equality and subtype relations for types.

Starting from a simple untyped arithmetic language, TAPL gradually introduces new features and combines them with some of the existing features to form various languages. However, due to the use of algebraic datatypes in OCaml, “combining” features is actually done through copying and pasting code, causing modularity issues. \textit{EVF}, on the other hand, is equipped with modular composition mechanisms and can compose features without code duplication.

Figure 5.1 gives a bird’s-eye view of the \textit{EVF} implementation of TAPL. The interactions among languages (features) are explicitly revealed by the arrows. To enhance modularity, we decompose conceptually independent features into separate packages (represented as boxes) for further reuse. For example, \textit{bool} is one of the smaller languages extracted from \textit{arith}, representing booleans. Afterwards, both \textit{arith} and \textit{simplebool} can reuse \textit{bool} and add their own functionality instead of duplicating \textit{bool} in their respective implementa-
5.1.1 Concrete Syntax

EVF can be integrated with other tools, such as Naked Object Algebras (NOA) [24], for developing external DSLs. NOA is a framework for developing extensible syntax based on Object Algebras. NOA provides syntax annotations to decorate constructors in an object algebra interface. With modest changes to the implementation of NOA, extensible concrete syntax can be supported in EVF as well. For example, the object algebra interface of the bool language is defined as follows:

```java
@Visitor public interface TermAlg<Term> {
    @Syntax("term = 'true'")
    Term TmTrue();

    @Syntax("term = 'false'")
    Term TmFalse();

    @Syntax("term = 'if' term 'then' term 'else' term")
    Term TmIf(Term t1, Term t2, Term t3);
}
```

Each constructor of TermAlg is associated a syntax production using the @Syntax annotation. Arguments of generic type Term correspond to the nonterminal term whereas non-generic arguments map to tokens. Tokens are surrounded by single quotes, for example ‘true’. A complete grammar will be generated by collecting these annotations along the interface hierarchy. Then a parser can be generated by feeding that grammar to a parser generator like ANTLR [50]. Refer to Gouseti et al.’s paper [24] for more details.

Figure 5.1: Package dependency graph
5.1 Overview

public interface Eval1<Term> extends TermAlgDefault<Term, Term>, bool.Eval1<Term>, nat.Eval1<Term> {
    @Override TermAlgMatcher<Term, Term> matcher();
    @Override GTermAlg<Term, Term> alg();
    @Override IsNumericVal<Term> isNumericVal();
}

default Term TmIsZero(Term t) {
    return matcher()
        .TmZero(() -> alg().TmTrue())
        .TmSucc(nv1 -> isNumericVal().visitTerm(nv1) ?
            alg().TmFalse() : alg().TmIsZero(visitTerm(t)))
        .otherwise(() -> alg().TmIsZero(visitTerm(t)))
        .visitTerm(t);
}

default Zero<Term> m() {
    return bool.Eval1.super.m();
}

Figure 5.2: Small-step evaluator for arith

5.1.2 Composable Language Implementations

EVF has a good support for language composition [17]. Specifically, three forms of language composition - language extension, language unification, and extension composition - are supported. The support for language composition in EVF owes to Java 8 multiple interface inheritance.

For example, the language arith is a unification of two smaller languages, nat and bool, with an extension (TmIsZero) that supports testing whether a term is zero or not. The term definition of arith is shown below:

Visitor public interface TermAlg<Term> extends bool.TermAlg<Term>, nat.TermAlg<Term> {
    Term TmIsZero(Term t);
}

Instead of duplicating constructs from nat and bool, we reuse them by extending the TermAlg from nat and bool. The kind of composability retains on operations as well. Figure 5.2 shows the small-step evaluator for arith, which assembles the small-step evaluator from nat and bool and complements the TmIsZero case. From Figure 5.1 we can see that arith is further composed by the extension package.

5.1.3 Nontrivial Operations

Operations that use pattern matching extensively, have complex dependencies and/or require multiple dispatching are hard to model in Object Algebras. Small-step evaluator, type equality, and subtype relation are representatives of such non-trivial operations. Small-step evaluator, for example, relies on extensive pattern matching and complex dependencies. The small-step semantics is formulated by a set of evaluation rules, each of which states how a term can be rewritten in a single step. With EVF we are able to translate them, demonstrated by the TmIsZero case shown in Figure 5.2. It uses the TermAlgMatcher to do case analysis on the inner term and depends on IsNumericVal for checking whether it is a numeric value in the TmSucc subcase.
5.1.4 Multiple Sorts

The case study also illustrates how multi-sorted object algebras can be modeled using the EVF framework. The demand for multiple sorts arises when a term needs a type in its definition. For instance, typed models the typed lambda calculus, where TermAlg is a multi-sorted algebra interface:

```java
@Visitor public interface TermAlg<Term, Ty>
extends varapp.TermAlg<Term> {
    Term TmAbs(String x, Ty ty, Term t);
}
```

The abstraction (TmAbs) of typed lambda calculus requires its argument of a specific type. Here we use another type parameter Ty to loosely capture the dependency on types. And we model types using a separately defined algebra interface:

```java
@Visitor public interface TyAlg<Ty> {
    Ty TyArr(Ty ty1, Ty ty2);
}
```

The reason to separate the definition of terms and types is that they belong to different syntactic categories in typed lambda calculus on which two completely different sets of operations are defined. It would not make sense to have a small-step evaluator on types or define subtyping relations on terms. This separation makes algebras fine-grained, allowing independent extensibility on types or terms. But it would be complicated if the two algebra interfaces are mutually dependent since that requires mutual dependencies. For such cases, a single algebra interface with multiple input carrier types might be better to capture mutually dependent sorts.

5.2 Evaluation

To evaluate EVF’s implementation of the case study, we compare to the original implementation\(^1\). Table 5.1 compares the number of source lines of code (SLOC, excluding blank lines and comments) of EVF’s implementation with the original OCaml implementation, package by package. Although an OOP language like Java is considerably more verbose than a functional language like OCaml, EVF’s implementation reduces approximately 32% of SLOC counting all packages, thanks to modularity and code generation techniques. The reduction of SLOC for each original package is on average 67%. For other feature-rich languages, the reduction is even more dramatic and can be up to 84%.

The reason is that all these original packages reuse other packages more or less. If all these languages were orthogonal in features, OCaml would beat EVF in terms of SLOC without question. However, from Figure 5.1 we can see that features like lambda calculus are frequently reused by other packages directly or indirectly, which makes a great difference to the total SLOC.

The comparison of SLOC between packages may not be that straightforward: EVF implementations may have dependencies on other packages whereas OCaml implementations are stand-alone. Table 5.2 compares the two implementation from the component

---

1https://www.cis.upenn.edu/~bcpierce/tapl/
2We do not count instantiation code and demo components presented in this section.
3We count only the files core.ml and syntax.ml, excluding the parser, the REPL and etc.
5.3 Performance Measurements

<table>
<thead>
<tr>
<th>Extracted Package</th>
<th>EVF</th>
<th>Original Package</th>
<th>EVF</th>
<th>OCaml</th>
<th>% Reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>115</td>
<td>arith</td>
<td>48</td>
<td>102</td>
<td>51%</td>
</tr>
<tr>
<td>extension</td>
<td>148</td>
<td>bot</td>
<td>90</td>
<td>184</td>
<td>52%</td>
</tr>
<tr>
<td>moreextension</td>
<td>114</td>
<td>fullerror</td>
<td>152</td>
<td>366</td>
<td>59%</td>
</tr>
<tr>
<td>nat</td>
<td>123</td>
<td>fullref</td>
<td>274</td>
<td>880</td>
<td>69%</td>
</tr>
<tr>
<td>record</td>
<td>231</td>
<td>fullsimple</td>
<td>110</td>
<td>651</td>
<td>84%</td>
</tr>
<tr>
<td>top</td>
<td>113</td>
<td>fullsub</td>
<td>176</td>
<td>628</td>
<td>72%</td>
</tr>
<tr>
<td>typed</td>
<td>207</td>
<td>fulluntyped</td>
<td>71</td>
<td>300</td>
<td>77%</td>
</tr>
<tr>
<td>utils</td>
<td>116</td>
<td>redbot</td>
<td>65</td>
<td>255</td>
<td>75%</td>
</tr>
<tr>
<td>varapp</td>
<td>89</td>
<td>simplebool</td>
<td>71</td>
<td>211</td>
<td>67%</td>
</tr>
<tr>
<td>variant</td>
<td>178</td>
<td>tyarith</td>
<td>37</td>
<td>135</td>
<td>73%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>untyped</td>
<td>61</td>
<td>128</td>
<td>53%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>2605</td>
<td>3840</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 5.1: SLOC statistics EVF vs OCaml: A package perspective.

<table>
<thead>
<tr>
<th>Component</th>
<th>EVF</th>
<th>OCaml</th>
<th>% Reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatype definition</td>
<td>91</td>
<td>231</td>
<td>61%</td>
</tr>
<tr>
<td>Small-step evaluator</td>
<td>256</td>
<td>481</td>
<td>47%</td>
</tr>
</tbody>
</table>

Table 5.2: SLOC comparison EVF versus OCaml: A component perspective

The table sums the SLOC of two core components, datatype definitions and the small-step evaluator, for all packages. The results show that their SLOC are both reduced significantly, which explains why the total SLOC of EVF is reduced.

To access the correctness of our implementation, we have ported all the test programs from the original implementation. Although the pretty printing operation is implemented slightly different, all the evaluation results are effectively the same.

5.3 Performance Measurements

This section gives the preliminary performance measurements on EVF and discusses its limitations. The novel visitX() methods introduced by EVF add one more level of dispatching to the standard Visitor pattern, which causes some execution overhead. To have a rough idea on the impact of visitX methods on performance, we run a microbenchmark adapted from [49]. We compare ourselves with respect to the two variants of Visitor pattern [9]: imperative visitor and functional visitor. An imperative Visitor uses side effects to do the computation; A functional Visitor is immutable, computing the result via return values. We also compare ourselves to Runabout [25], a performant reflection-based approach for achieving extensibility.

The benchmark requires each approach to model linked lists and sum a linked list of length 2000 for 10000 times. Implementations with these four approaches are shown in Figure 5.3.

The benchmarks were compiled using Oracle JDK 1.8 and executed on the JVM in 64bit server mode on a 2.6 GHz MacBook Pro Intel Core i5 with 8GB memory. Table 5.3 summarizes the run time of each approach. The results show that the imperative visitors...
interface Visitor {
    void Nil(Nil nil);
    void Cons(Cons cons);
    void Link(Link link);
}
class Sum implements Visitor {
    int sum = 0;
    public void Nil(Nil nil) {}
    public void Cons(Cons cons) {
        sum += x.head;
        x.tail.accept(this);
    }
    public void Link(Link link) {
        link.list.accept(list);
    }
}

(a) Imperative visitor

class Sum extends Runabout {
    int sum = 0;
    public void visit(Nil nil) {}
    public void visit(Cons cons) {
        sum += cons.head;
        visitAppropriate(cons.tail);
    }
    public void visit(Link link) {
        visitAppropriate(link.list);
    }
}

(c) Runabout

interface Visitor<O> {
    O Nil();
    O Cons(int head, List tail);
    O Link(boolean color, List list);
}
class Sum implements Visitor<Integer> {
    public Integer Nil() {
        return 0;
    }
    public Integer Cons(int head, List tail) {
        return head + t.accept(this);
    }
    public Integer Link(boolean color, List list) {
        return list.accept(this);
    }
}

(b) Functional visitor

@Visitor interface ListAlg<List> {
    List Nil();
    List Cons(int head, List tail);
    List Link(boolean color, List list);
}
interface Sum>List extends ListAlgQuery<List,Integer> {
    default Monoid<Integer> m() {
        return new AddMonoid();
    }
    default Integer Cons(int head, List tail) {
        return head + visitList(tail);
    }
}

(d) EVF

Figure 5.3: Linked list and sum operation implementations

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative Visitor</td>
<td>122</td>
</tr>
<tr>
<td>Functional Visitor</td>
<td>235</td>
</tr>
<tr>
<td>Runabout</td>
<td>274</td>
</tr>
<tr>
<td><strong>EVF</strong></td>
<td><strong>259</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Performance measurements
are fastest among the four approaches, which uses half the time compared to the other three approaches. There is a minor difference between the other three approaches on execution time. In summary, EVF is slightly slower (about 10%) than the functional Visitor pattern due to the additional dispatching and is comparable to Runabout. Of course, more rigorous and extensive benchmarks need to be performed to validate the conclusion.
Chapter 6

Shallow Embedding and Object-Oriented Programming

Shallow Embedded DSL (EDSLs) use procedural abstraction to directly encode a DSL into an existing host language. Procedural abstraction has been argued to be the essence of OOP. This chapter argues that OOP abstractions (including inheritance, subtyping and type-refinement) increase the modularity and reuse of shallow EDSLs when compared to classical procedural abstraction. We make this argument by taking a recent paper by Gibbons and Wu, where procedural abstraction is used in Haskell to model a simple shallow EDSL, and we recode that EDSL in Scala. From the semantic and modularity point of view the Scala version has clear advantages over the Haskell version.

6.1 Introduction

Since Hudak’s seminal paper on Embedded DSL (EDSLs) [29], existing languages (e.g. Haskell) have been used to directly encode DSLs. Two common approaches to EDSLs are the so-called shallow and deep embeddings. The origin of that terminology can be attributed to Boulton et al.’s work [8]. The difference between these two styles of embeddings is commonly described as follows:

With a deep embedding, terms in the DSL are implemented simply to construct an abstract syntax tree (AST), which is subsequently transformed for optimization and traversed for evaluation. With a shallow embedding, terms in the DSL are implemented directly by their semantics, bypassing the intermediate AST and its traversal.[23]

Although the above definition is quite reasonable and widely accepted, it leaves some space to (mis)interpretation. For example it is unclear how to classify an EDSL implemented using the Composite or Interpreter patterns in Object-Oriented Programming (OOP). Would this OO approach be classified as a shallow or deep embedding? We feel there is a rather fuzzy line here, and the literature allows for both interpretations. Some authors working on OOP EDSLs [57, 60] consider a Composite to be a deep embedding. Some other authors [23, 5] consider implementations using tuples and/or the Composite pattern to be a shallow embedding.
To avoid ambiguity we propose defining shallow embeddings as EDSLs implemented using *procedural abstraction* [55]. Such interpretation arises naturally from the domain of shallow EDSLs being functions, and procedural abstraction being a way to encode data abstractions using functions. As Cook [14] argued, procedural abstraction is also the essence of OOP. Thus, according to our definition, the implementation of a shallow EDSL in OOP languages should simply correspond to a standard object-oriented program.

The main goal of this chapter is to show the close relationship between shallow embeddings and OOP, and argue that OOP languages have advantages for the implementation of shallow embeddings. It is perhaps partly due to those advantages that some authors have considered Composite-based implementations to be deep embeddings. However, we argue that the OOP mechanisms do not change the essence of the implementation, which is still shallow (i.e. using procedural abstraction). To understand how OOP is helpful for shallow embeddings, we should first look at the limitations of shallow embeddings commonly found in the literature - shallow EDSLs is that they only support *single* interpretation.

We show that OOP abstractions, including *inheritance, subtyping* and *type-refinement*, are helpful to address those problems. For the first problem, we can employ a recently proposed design pattern [68], which provides a simple solution to the *Expression Problem* [67] in OOP languages. Thus using just standard OOP mechanisms enables *multiple modular interpretations* to co-exist and be combined in shallow embeddings.

We make our arguments by taking a recent paper by Gibbons and Wu [23], where procedural abstraction is used in Haskell to model a simple shallow EDSL, and we recode that EDSL in Scala\(^1\). From the *modularity* point of view the Scala version has clear advantages over the Haskell version.

### 6.2 Shallow Object-Oriented Programming

This section shows that an OO approach and shallow embeddings using procedural abstraction are closely related. We use a subset of the DSL presented in Gibbons and Wu’s paper [23] as the running example. We first give the original shallow embedded implementation in Haskell and rewrite it towards an “OO style”. Then translating the program into an OO language becomes straightforward. We switch to Scala as the demonstrating OO language in this chapter because of its concise syntax. Indeed, none of Scala’s functional features is used. Essentially, the code can be trivially adapted to any OO language that supports subtyping, inheritance and type-refinements such as Java.

#### 6.2.1 Scans: A DSL for Parallel Prefix Circuits

*Scans* [27] is a DSL for describing parallel prefix circuits. Given a associative binary operator \(\bullet\), the prefix sum of a non-empty sequence \(x_1, x_2, \ldots, x_n\) is \(x_1 \bullet x_2 \bullet \ldots \bullet x_n\). Such computation can be performed in parallel for a parallel prefix circuit. Parallel prefix circuits have a of applications, including binary addition and sorting algorithms.

\(^1\)Available online: https://github.com/wxzh/shallow-dsl
Consider a DSL named Scans that models parallel circuits. Its BNF grammar is given below:

\[
\langle \text{circuit} \rangle ::= \text{'identity'} \langle \text{positive-number} \rangle \\
\quad \text{'fan'} \langle \text{positive-number} \rangle \\
\quad \text{'beside'} \langle \text{circuit} \rangle \langle \text{circuit} \rangle \\
\quad \text{'above'} \langle \text{circuit} \rangle \langle \text{circuit} \rangle \\
\quad \text{'stretch'} \langle \text{positive-numbers} \rangle \langle \text{circuit} \rangle
\]

Scans has five constructs: two primitives (identity and fan) and three combinators (beside, above and stretch). Their meanings are: identity n contains n parallel wires; fan n has n vertical wires with its first wire connected to all the remaining wires from top to bottom; beside c1 c2 joins two circuits c1 and c2 horizontally; above c1 c2 combines two circuits of the same width vertically; stretch ns c inserts more wires into the circuit c by summing up ns. For example, Figure 6.1 visualizes a circuit constructed using all these five constructs.

The construction of the circuit is explained as follows. The whole circuit can be divided into three sub-circuits, vertically: the top sub-circuit is a two fan 2 put side by side; the middle sub-circuit is a fan 2 stretched by inserting a wire on the left hand side of the first and second wire; the bottom sub-circuit is a fan 2 between two identity 1.

### 6.2.2 Shallow Embeddings and OOP

Shallow embeddings define a language directly through encoding its semantics using procedural abstraction. In the case of Scans, a shallowly embedded implementation should conform to the following types:

```haskell
type Circuit = ... 
identity :: Int -> Circuit 
fan :: Int -> Circuit 
beside :: Circuit -> Circuit -> Circuit 
above :: Circuit -> Circuit -> Circuit 
stretch :: [Int] -> Circuit -> Circuit 
```

The type Circuit, representing the semantic domain, is to be filled in with a concrete type according to the semantics. Suppose that the semantics of Scans is to calculate the width of a circuit. The definitions would be:

```haskell
type Circuit = Int 
identity n = n 
fan n = n 
beside c1 c2 = c1 + c2 
```
above c1 c2 = c1
stretch ns c = sum ns

Note that, for this interpretation, the Haskell domain is simply Int. This means that we will get the width right after the construction of a circuit. For example, running code that represents the circuit shown in Figure 6.1

```
Prelude> :{
  Prelude| (fan 2 'beside' fan 2) 'above'
  Prelude| stretch [2,2] (fan 2) 'above'
  Prelude| (identity 1 'beside' fan 2 'beside' identity 1)
  Prelude| :}
4
```

This domain is a degenerate case of procedural abstraction, where Int can be viewed as a no argument function. In Haskell, due to laziness, Int is a good representation. In a call-by-value language a no-argument function () -> Int would be more appropriate to deal correctly with potential control-flow language constructs. We will see an interpretation of a more complex domain in Section 6.3.3.

Towards OOP A simple, semantics preserving, rewriting of the above program is given below, where a record with a sole field captures the domain and is declared as a newtype:

```hs
newtype Circuit = Circuit {width :: Int}
id n = Circuit {width = n}
fan n = Circuit {width = n}
beside c1 c2 = Circuit {width = width c1 + width c2}
above c1 c2 = Circuit {width = width c1}
stretch ns c = Circuit {width = sum ns}
```

The implementation is still shallow because newtype does not add any operational behavior to the program, and hence the two programs are effectively the same. However, having fields makes the program look more like an OOP program.

Porting to Scala Indeed, we can easily translate the Haskell program into an OO language like Scala:

```scala
package width

trait Circuit {
  def width: Int
}

trait Identity extends Circuit {
  val n: Int
  def width = n
}

trait Fan extends Circuit {
  val n: Int
  def width = n
}

trait Beside extends Circuit {
  val c1, c2: Circuit
  def width = c1.width + c2.width
}

trait Above extends Circuit {
  val c1, c2: Circuit
  def width = c1.width
}

trait Stretch extends Circuit {
  val ns: List[Int]
  val c: Circuit
  def width = ns.sum
}
```
The record type maps to the trait `Circuit` and field declaration becomes a method declaration. Each case in the semantic function corresponds to a trait and its parameters become fields of that trait. And these traits extend `Circuit` and implement `width`.

Essentially, this implementation is how we would model Scans with an OO language in the first place, following the Interpreter pattern (which uses Composite pattern to organize classes). A minor difference is the use of traits, instead of classes. Using traits instead of classes enables some additional modularity via multiple (trait-)inheritance.

To use this Scala implementation in a manner similar to the Haskell implementation, we define some smart constructors:

```scala
def identity(x: Int) = new Identity {val n=x}
def fan(x: Int) = new Fan {val n=x}
def above(x: Circuit, y: Circuit) = new Above {val c1=x; val c2=y}
def beside(x: Circuit, y: Circuit) = new Beside {val c1=x; val c2=y}
def stretch(xs: List[Int], x: Circuit) = new Stretch {val ns=xs; val c=x}
```

Then we are able to construct and calculate the width of the circuit shown in Figure 6.1 again:

```
scala> { above(beside(fan(2), fan(2)),
| above(stretch(List(2,2), fan(2)),
| beside(identity(1), beside(fan(2), identity(1))))).width
| }
res0: Int = 4
```

### 6.3 Interpretations in Shallow Embeddings

An often stated limitation of shallow embeddings is that they allow only a single interpretation. Gibbons and Wu [23] work around this problem by using tuples. However, their encoding needs to modify the original code, and thus is non-modular. This section illustrates how various types of interpretations can be modularly defined in OOP.

#### 6.3.1 Multiple Interpretations

**Multiple Interpretations in Haskell** Suppose that we want to have an additional function that checks whether a circuit is constructed correctly. Gibbons and Wu’s solution is:

```scala
type Circuit = (Int,Int)
identity n = (n,0)
fan n = (n,1)
above c1 c2 = (width c1,depth c1 + depth c2)
beside c1 c2 = (width c1 + width c2, depth c1 'max' depth c2)
stretch ns c = (sum ns,depth c)

width = fst
depth = snd
```

A tuple is used to accommodate multiple interpretations and each interpretation is defined as a projection on the tuple. This solution is not modular because it relies on defining the two interpretations (`width` and `depth`) simultaneously, using a tuple. It is not possible to reuse the independently defined `width` function in Section 6.2.2. Whenever a new interpretation is needed (e.g. `depth`), the original code has to be revised: the arity of the
Multiple Interpretations in Scala  In contrast, Scala allows new interpretations to be introduced in a modular way, as shown in Figure 6.2. The encoding relies on three OOP abstraction mechanisms: inheritance, subtyping and type-refinement. Specifically, the new `Circuit` is a subtype of `width.Circuit` and declares a new method `width`. The hierarchy implements the new `Circuit` by inheriting the corresponding trait from the `width` package and implementing the `width` method. Also, fields of `Beside` are refined with the new `Circuit` type to avoid type mismatches in methods [68].

6.3.2 Dependent Interpretations

Dependent Interpretations in Haskell  Dependent interpretations are a generalization of multiple interpretations. A dependent interpretation does not only depend on itself but also on other interpretations. An instance of such interpretation is `wellSized`, which checks whether a circuit is constructed correctly. `wellSized` is dependent because combinators such as `above` have width constraints on its circuit components.

In Haskell, dependent interpretations are again defined with tuples in a non-modular way:

```haskell
  type Circuit = (Int,Bool)
  identity n = (n,True)
  fan n    = (n,True)
  above c1 c2 = (width c1,welSized c1 && welSized c2 && width c1==width c2)
  beside c1 c = (width c1 + width c2,wellSized c1 && wellSized c2)
  stretch ns = (sum ns,wellSized c && length ns==width c)

  width    = fst
  welSized = snd
```

Figure 6.2: Adding new interpretations

Tuple must be incremented and the new interpretation has to be appended to each case.
Dependent Interpretations in Scala  Fortunately, an OO approach does not have such restriction:

```scala
package wellsized
trait Circuit extends width.Circuit {
  def wellSized: Boolean
}
trait Identity extends width.Identity with Circuit {
  def wellSized = n > 0
}
trait Fan extends width.Fan with Circuit {
  def wellSized = n > 0
}
trait Beside extends width.Beside with Circuit {
  val c1, c2: Circuit
  def wellSized = c1.wellSized && c2.wellSized
}
trait Above extends Circuit with width.Above {
  val c1, c2: Circuit
  def wellSized = c1.wellSized && c2.wellSized &&
  c1.width==c2.width
}
trait Stretch extends Circuit with width.Stretch {
  val c: Circuit
  def wellSized = c.wellSized && ns.length==c.width
}
```

Note that width and wellSized are defined separately. Essentially, it is sufficient to define wellSized while knowing only the signature of width in Circuit.

6.3.3 Context-Sensitive Interpretations

Context-sensitive Interpretations in Haskell  Interpretations may rely on some mutable contexts. Consider an interpretation that simplifies the representation of a circuit. A circuit can be divided horizontally into layers. Each layer can be represented as a sequence of pairs \((i, j)\), denoting the connection from wire \(i\) to wire \(j\). For instance, circuit shown in Figure 6.1 has the following layout:

\[
[[((0, 1), (2, 3))], [(1, 3)], [(1, 2)]]
\]

The following Haskell code models the interpretation described above:

```haskell
type Layout = [[[Int, Int]]]
type Circuit = (Int,(Int -> Int) -> Layout)
identity n = (n,\f -> [])
fan n = (n,\f -> [[[f 0,f j] | j <- [1..n-1]]])
above c1 c2 = (width c1,\f -> tlayout c1 f ++ tlayout c2 f)
beside c1 c2 = (width c1 + width c2,
  \f -> lzw (+++) (tlayout c1 f) (tlayout c2 ((width c1+) . f)))
stretch ns c = (sum ns,\f -> tlayout c (pred . (vs!!) . f))
  where vs = scanl1 (+) ns
width = fst
tlayout = snd

lzw :: (a -> a -> a) -> [a] -> [a] -> [a]
lzw f [ ] ys = ys
lzw f xs [ ] = xs
lzw f (x : xs) (y : ys) = f x y : lzw f xs ys
```
tlayout is firstly a dependent interpretation, relying on itself as well as |width|. More importantly, it is a context-sensitive interpretation. A circuit’s layout would be changed when it is stretched or put on the right hand side of another circuit. To efficiently produce a layout, these changes are not immediately applied to the affected circuit. Rather, they are accumulated in a parameter and are applied all at once in the end. The domain |tlayout| is thereby not a direct value that represents the layout (|Layout|) but a function that takes a transformation on wires and then produces a layout (|(Int->Int)->Layout|). An auxiliary definition \(\text{lzw} \) ("long zip with") zips two lists by applying the function to the two elements of the same index and appending the remaining elements of the longer list to the resulting list.

**Context-sensitive Interpretations in Scala**  
Context-sensitive interpretations in our OO approach are unproblematic as well.

```scala
type Layout = List[List[Tuple2[Int, Int]]]

trait Circuit extends width.Circuit {
  def tlayout(f: Int => Int): Layout
}

trait Identity extends Circuit with width.Identity {
  def tlayout(f: Int => Int) = List()
}

trait Fan extends Circuit with width.Fan {
  def tlayout(f: Int => Int) = List(for (i <- List.range(1,n)) yield (f(0),f(i)))
}

trait Above extends Circuit with width.Above {
  val c1, c2: Circuit
  def tlayout(f: Int => Int) = c1.tlayout(f) ++ c2.tlayout(f)
}

trait Beside extends Circuit with width.Beside {
  val c1, c2: Circuit
  def tlayout(f: Int => Int) = lzw [c1.tlayout(f), c2.tlayout(f.andThen(c1.width + _))] (_ ++ _)
}

trait Stretch extends Circuit with width.Stretch {
  val c: Circuit
  def tlayout(f: Int => Int) = c.tlayout(f.andThen(partialSum(ns)(_) - 1))
}

  case (Nil,_) => ys
  case (_,Nil) => xs
  case (x::xs,y::ys) => f(x,y)::lzw(xs,ys)(f)
}

def partialSum(ns: List[Int]): List[Int] = ns.scanLeft(0)(_ + _).tail
```

The Scala version is both modular and intuitive, where mutable contexts are captured as method arguments.

### 6.3.4 Adding New Constructs

Not only new interpretations, new constructs may need when a DSL evolves. For the case of Scans, we may want to have a \(rstretch\) (right stretch) combinator which is similar to the \(stretch\) combinator but inserts wires from the opposite direction. Shallow embeddings make the addition of \(rstretch\) easy through defining a new function:
6.3 Interpretations in Shallow Embeddings

```haskell
rstretch :: [Int] -> Circuit -> Circuit
rstretch = ...
```

Such simplicity of adding new constructs retain on our OO approach, just through defining new traits that implement `Circuit`:

```scala
trait RStretch extends Circuit {
  val ns: List[Int]
  val c: Circuit
  ...
}
```

### 6.3.5 Discussion

Gibbons and Wu claim that in shallow embeddings new language constructs are easy to add, but new interpretations are hard. As our OOP approach shows, in OOP both language constructs and new interpretations are easy to add in shallow embeddings. In other words, the circuit DSL presented so far does not suffer from the Expression Problem. The key point is that procedural abstraction combined with OOP features (subtyping, inheritance and type-refinement) adds expressiveness over traditional procedural abstraction. Gibbons and Wu do discuss a number of advanced techniques that can solve some of the modularity problems. For example, using type classes, *finally tagless* [10] can deal with the example in Section 6.3.1. However tuples are still needed to deal with dependent interpretations. In contrast the approach proposed here is just straightforward OOP, and dependent interpretations are not a problem.
Chapter 7

Related Work and Conclusion

7.1 Related Work

7.1.1 The Visitor Pattern

Extensible Visitors Early work on the Visitor pattern [35, 66, 49] pointed out extensibility limitations in the Visitor pattern and proposed several solutions. Those early approaches use runtime checks and can suffer from runtime errors without careful use. Palsberg and Jay [49] proposed a generic class Walkabout as the root of visitors. By using Java’s runtime reflection, the Walkabout removes the need for accept method in AST types. This decouples AST type from the visitor interface, allowing new variants to be introduced as well. Unfortunately, the extensive use of runtime reflection causes severe performance penalties. Based on the Walkabout, Grothoff proposed the Runabout [25], attempting to achieve reasonable performance through sophisticated bytecode generation and caching. Forax’s Sprintabout [18] further improves the performance of Runabout by eliminating the manual creation of AST infrastructure. However, Walkabout and its successors are not type-safe. Torgersen [62] developed variations of the Visitor pattern to solve the Expression Problem [67]. The solutions are type-safe but rely on advanced features of generics such as wildcards or F-bounds. Also the programming patterns are relatively complex thus hard for programmers to learn. Inspired by other type-safe variations of Visitor pattern [48, 45, 28] using advanced Scala type system features, our work applies similar techniques but requires only simple generics available in Java. The visitX() methods in generalized object algebra interfaces are a novel contribution of our work, and greatly account for the simplicity and flexibility of EVF.

Structure-Shy Traversals with Visitors There has also been work on eliminating boilerplate code in the Visitor pattern. A typical way is to use default visitors [43]. A default visitor defines the traversal template for a specific visitor interface. By subclassing the default visitor, concrete visitors only need to override interesting cases. Walkabout [49] removes the need of a new traversal template for every visitor interface by providing a single traversal template that works for all visitors. The default traversal in Walkabout is achieved through invoking the overloaded visit method on children. EVF employs static reflection to automatically generate specialized traversal templates for each modular visitor interface. But the fundamental difference is that static type safety is preserved in
**EVF.** Visser [65] ported ideas from rewriting system Stratego [63] to the Visitor pattern. The resulting framework JJTraverler exposes a series of visitor combinators to achieve flexible traversal control and visitor combination. The proposed combinators can express various traversal strategies such as bottom-up, top-down, sequential or alternative composition of visitors. To make these combinators generic, runtime reflection is also used. The combinators are developed in the setting of imperative visitors in [65] and hence cannot be directly mapped to **EVF**. We would like to explore a library of visitor combinators in **EVF** as future work.

### 7.1.2 Object Algebras

Object Algebras [46] are a modular programming pattern. However, as discussed in detail in Section 2.5.1, Object Algebras have several expressiveness limitations. There has been a lot of effort pointing out and trying to solve these limitations [47, 24], in particular, improving support for dependencies [47, 54]. Improved support for dependent operations is achieved using pairs, encodings of delegation, and languages with advanced type systems, such as Scala. Rendel et al. [54] establish the relationship between Object Algebras and attribute grammars and also propose new approaches to deal with modularity issues. However, this is also done in Scala. Using a new generalization of Object Algebras, **EVF** removes several limitations of Object Algebras. In particular, dependencies are dealt with standard OO composition. The **Shy** framework [74] is another work based on Object Algebras. **Shy** provides various types of default traversals for data structures, but only bottom-up traversals are supported. **EVF** removes the limitation of bottom-up traversals only in **Shy**, giving the user flexibility in the traversal strategy.

### 7.1.3 Solutions to the EP in Functional Programming

In functional programming, the two main solutions to the EP are tagless final [10] and data types à la carte [61] (DTC). Finally, tagless approach uses a type class to abstract over all possible interpretations of a language. Concrete interpretations are given through creating a data type and making it an instance of that type class. DTC represents language constructs separately and composes them together using extensible sums. DTC and finally tagless work well for most interpretations except for those that are dependent. Dependent interpretations still have to be defined along with what they depend on. This prevents new interpretations that depend on existing interpretations from being introduced modularly. Moreover, not like OO languages which come with subtyping, one has to manually implement the subtyping machinery for variants in DTC.

### 7.1.4 Component-Based Development

**Component-Based Language Development** The idea of constructing languages by assembling components can date back to 1980s [34]. Most closely related is Mosses’s work on component-based semantics [42]. The idea is to provide a collection of highly reusable fundamental constructs (funcons) with predefined semantics [11]. By mapping the constructs of a language to these funcons, the operational semantics of the language can be
obtained for free. The semantics of these funcons are specified using modular structural operational semantics (MSOS) [41]. Interpreters can be generated based on the MSOS specifications. Similar funcons can also be developed in EVF with the semantics specified as EVF interpreters. But the difference is that these interpreters can be modularly type checked and separately compiled.

Software-Product Lines Software-Product Lines (SPLs) [12, 39, 32] allow similar systems (with different variations) to be generated from a set of common features. There are various tools that can be used to develop SPLs, including GenVoca [7], AHEAD [6], FeatureC++ [3] and FeatureHouse [2]. SPLs tools can also be used to modularize features in programming languages and are an alternative to language workbenches. In contrast to language workbenches, SPLs tools are targeted at general purpose software development. Similarly to most language workbenches, most SPLs tools use syntactic modularization mechanisms, which do not support separate compilation and/or modular type-checking.

7.1.5 Object-Oriented Pattern Matching

Pattern matching is a feature originally from functional programming languages. Much work has been done on porting pattern matching to OO languages, which can generally be classified into three categories: encodings, language extensions and new languages.

Encodings The simplest way of simulating pattern matching in OOP is through encodings. Encodings allow existing OO languages to support pattern matching without extra installations (e.g. compiler-plugins). Forax and Roussel [19] proposed a solution based on method overloading and reflections. Pattern matching functionality can be obtained through extending the PatternMatcher class and overriding the match method for each pattern. Then an appropriate overloaded match will be invoked according to the runtime type of the object being matched through reflection. Their approach supports pattern matching over several arguments. However, the encoding is not type safe and the exhaustiveness of pattern matching is not checked. Moreover, the performance penalty is high, which is much slower than the Visitor pattern even with sophisticated optimizations.

Pattern matching in EVF can also be viewed as an encoding. The pattern matching infrastructure of EVF is built on external visitors, inspired by EADDs [72] and derive4j [1]. derive4j is an annotation processor that generates class hierarchy and pattern matching infrastructure for an algebraic datatype. Unlike EVF, pattern matching in derive4j is not extensible: only regular (closed) pattern matching is supported. Moreover, derive4j attempts to statically enforce one case per pattern. This leads to an explosion in the number of classes needed in the embedded DSL. Differently, EVF generates only one extensible matcher interface for each algebra interface. A matcher exposes fluent APIs for emulating pattern matching. Similar to EADDs we force a trailing default case. But the operations do not automatically work in extensions. Programmers would get warned when instantiating algebras if they forget to do type-refinements for the dependencies on
matchers.

**Language Extensions**  Another line of work attempts to extend existing OO languages like Java for pattern matching. TOM [52] extends Java with built-in support for algebraic datatypes and pattern matching. TOM’s `match` construct supports deep pattern matching. But unlike EVF, neither the datatype definitions or operations in TOM are extensible. Later development of TOM [4] has better integration with Java and supports powerful rewriting strategies. Similar rewriting strategies have been studied in the setting of the classic Visitor pattern [65] and are partially supported by EVF.

JMatch [38] is another Java extension with the support of pattern matching and iteration through modal abstraction. JMatch methods support multiple modes. They can not only be used as normal methods in forward modes but also deduce argument values for a given result in backward modes, merging the `apply()` and `unapply()` methods of Scala in one definition. However, exhaustiveness of pattern matching is not checked in JMatch. Later work [31] extends JMatch with exhaustiveness checking and extra expressiveness with the help of SMT solvers. But the type system of JMatch becomes complicated.

**New OO Languages**  New OO languages try to support pattern matching in the first place. We have compared our work with EADDs and case classes in Scala throughout chapter 4. As Emir et al. [16] pointed out, case classes require less notational overhead than visitors and have been applied in several projects to improve modularity [58]. Hofer et al. [28] presented an approach to modular DSLs in Scala, aiming to find a better way to model non-compositional operations. Three techniques: internal visitors, external visitors, and Scala’s case classes are investigated. It turns out that code using case classes is simpler than using visitors. In contrast to EADDs and case classes, EVF can achieve type-safe extensibility without defaults, by using visitors. Furthermore, the generated visitor infrastructure and traversal templates in EVF simplify the use of visitors. Users do have a choice between pattern matching with defaults and visitors. The former needs no instantiation and fewer dependencies while the latter is reusable. Both of the two approaches retain type-safe extensibility in EVF.

Fortress language [59] is yet another OO language that supports first-class pattern matching. Similar to Scala, Fortress uses traits to define a datatype and objects to define its cases. Exhaustiveness and extensibility in Fortress cannot be achieved at the same time as well. Nevertheless, Fortress provides a `comprises` clause for explicitly specifying all cases of a trait. This allows concrete implementation of those cases to be defined in separate files while preserving exhaustiveness checking. Fortress preserves type arguments at runtime, which brings extra expressiveness for pattern matching.

**7.2 Conclusion**

In this thesis we showed how to use OO languages to modularize language components. Equipped with powerful semantic modularization techniques, OO languages are suitable for developing extensible internal and external DSLs.
In the first part of the thesis, we introduced **EVF**, an extensible and expressive Java Visitor framework, for facilitating external DSL development. **EVF**’s support of extensible external visitors allows complex dependencies between operations to be expressed modularly and provides users with flexible traversal strategies for defining expressive operations. To make **EVF** easy to use, we design a fluent interface style embedded DSL to simulate pattern matching and support concrete syntax via annotations. Users only need to annotate conventional object algebra interfaces, then boilerplate code will be generated by the annotation processor. The infrastructure generated by **EVF** include visitors, traversals, pattern matching and grammar. The TAPL case study illustrates the applicability of **EVF**.

The second part of this thesis revealed the close correspondence between OOP and shallow embeddings, and how OOP improves the modularity of shallow EDSLs. In particular, multiple interpretations can be defined modularly with OOP.

### 7.3 Future Work

This thesis focuses on the modularity and extensibility problem in semantic part of a DSL. Other aspects of a DSL implementation, such as performance and developer tool support, are ignored. However, these issues are important in practice. One line of future work is to integrate **EVF** with Graal VM [71] and the Truffle framework [70] for obtaining performant DSLs easily. Another avenue is to continue developing **EVF** towards a language workbench.
Appendices
### A.1 Untyped Lambda Calculus with the Visitor Pattern

```java
interface LamAlg<O> {
    O Var(String x);
    O Abs(String x, Exp e);
    O App(Exp e1, Exp e2);
    O Lit(int n);
    O Sub(Exp e1, Exp e2);
}

interface Exp {
    <O> O accept(LamAlg<O> v);
}

class Var implements Exp {
    String x;
    Var(String x) {
        this.x = x;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Var(x);
    }
}

class Abs implements Exp {
    String x;
    Exp e;
    Abs(String x, Exp e) {
        this.x = x;
        this.e = e;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Abs(x, e);
    }
}

class App implements Exp {
    Exp e1, e2;
    App(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.App(e1, e2);
    }
}

class Lit implements Exp {
    int n;
    Lit(int n) {
        this.n = n;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Lit(n);
    }
}

class Sub implements Exp {
    Exp e1, e2;
    Sub(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    public <O> O accept(LamAlg<O> v) {
        return v.Sub(e1, e2);
    }
}```
Appendix A. Appendix

class FreeVars implements LamAlg<Set<String>> {
    public Set<String> Var(String x) {
        return Collections.singleton(x);
    }
    public Set<String> Abs(String x, Exp e) {
        return e.accept(this).stream().filter(y -> !y.equals(x))
            .collect(Collectors.toSet());
    }
    public Set<String> App(Exp e1, Exp e2) {
        return Stream.concat(e1.accept(this).stream(), e2.accept(this).stream())
            .collect(Collectors.toSet());
    }
    public Set<String> Lit(int n) {
        return Collections.emptySet();
    }
    public Set<String> Sub(Exp e1, Exp e2) {
        return Stream.concat(e1.accept(this).stream(), e2.accept(this).stream())
            .collect(Collectors.toSet());
    }
}

class SubstVar implements LamAlg<Exp> {
    String x;
    Exp s;
    SubstVar(String x, Exp s) { this.x = x; this.s = s; }
    public Exp Abs(String y, Exp e) {
        if (y.equals(x)) return new Abs(x, e);
        if (s.accept(new FreeVars()).contains(x)) throw new RuntimeException();
        return new Abs(x, e.accept(this));
    }
    public Exp App(Exp e1, Exp e2) {
        return new App(e1.accept(this), e2.accept(this));
    }
    public Exp Var(String y) {
        return y.equals(x) ? s : new Var(x);
    }
    public Exp Lit(int n) {
        return new Lit(n);
    }
    public Exp Sub(Exp e1, Exp e2) {
        return new Sub(e1.accept(this), e2.accept(this));
    }
}

A.2 Untyped Lambda Calculus with Object Algebras

interface LamAlg<Exp> {
    Exp Var(String x);
    Exp Abs(String x, Exp e);
    Exp App(Exp e1, Exp e2);
    Exp Lit(int n);
    Exp Sub(Exp e1, Exp e2);
}
interface IFV {
    Set<String> FV();
}
class FreeVars<Exp> implements LamAlg<IFV> {
    public IFV Var(String x) {
        return () -> Collections.singleton(x);
    }
    public IFV Abs(String x, IFV e) {

return () -> e.FV().stream().filter(y -> !y.equals(x))
   .collect(Collectors.toSet());
}

public IFV App(IFV e1, IFV e2) {
  return () -> Stream.concat(e1.FV().stream(), e2.FV().stream())
   .collect(Collectors.toSet());
}

public IFV Lit(int n) {
  return () -> Collections.emptySet();
}

public IFV Sub(IFV e1, IFV e2) {
  return () -> Stream.concat(e1.FV().stream(), e2.FV().stream())
   .collect(Collectors.toSet());
}

interface ISubst<Exp> {
  Exp before();
  Exp after();
}

class SubstVar<Exp extends IFV> implements LamAlg<ISubst<Exp>> {
  String x;
  Exp s;
  LamAlg<Exp> alg;
  SubstVar(String x, Exp s, LamAlg<Exp> alg) {
    this.x = x;
    this.s = s;
    this.alg = alg;
  }

  public ISubst<Exp> Var(String y) {
    return new ISubst<Exp>() {
      public Exp before() {
        return alg.Var(y);
      }
      public Exp after() {
        return y.equals(x) ? s : alg.Var(y);
      }
    };
  }

  public ISubst<Exp> Abs(String y, ISubst<Exp> e) {
    return new ISubst<Exp>() {
      public Exp before() {
        return alg.Abs(y, e.before());
      }
      public Exp after() {
        if(y.equals(x)) return alg.Abs(y, e.before());
        if(s.FV().contains(y)) throw new RuntimeException();
        return alg.Abs(y, e.after());
      }
    };
  }

  public ISubst<Exp> App(ISubst<Exp> e1, ISubst<Exp> e2) {
    return new ISubst<Exp>() {
      public Exp before() {
        return alg.App(e1.before(), e2.before());
      }
      public Exp after() {
        return alg.App(e1.after(), e2.after());
      }
    };
  }

  public ISubst<Exp> Lit(int n) {
    return new ISubst<Exp>() {
      public Exp before() {
        return alg.Lit(n);
      }
      public Exp after() {
        return alg.Lit(n);
      }
    };
  }
}
public ISubst<Exp> Sub(ISubst<Exp> e1, ISubst<Exp> e2) {
    return new ISubst<Exp>() {
        public Exp before() {
            return alg.Sub(e1.before(), e2.before());
        }
        public Exp after() {
            return alg.Sub(e1.after(), e2.after());
        }
    };
}
Bibliography


[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addisson-Wesley, 1994.


