EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse*

Weixin Zhang[1] and Bruno C. d. S. Oliveira[2]

1 **The University of Hong Kong, Hong Kong, China**
wxzhang2@cs.hku.hk
2 **The University of Hong Kong, Hong Kong, China**
bruno@cs.hku.hk

Abstract

Object Algebras are a design pattern that enables extensibility, modularity, and reuse in mainstream object-oriented languages such as Java. The theoretical foundations of Object Algebras are rooted on Church encodings of datatypes, which are in turn closely related to folds in functional programming. Unfortunately, it is well-known that certain programs are difficult to write and may incur performance penalties when using Church-encodings/folds.

This paper presents **EVF**: an extensible and expressive Java VISITOR framework. The visitors supported by **EVF** generalize Object Algebras and enable writing programs using a *generally recursive style* rather than folds. The use of such generally recursive style enables users to more naturally write programs, which would otherwise require contrived workarounds using a fold-like structure. **EVF** visitors retain the type-safe extensibility of Object Algebras. The key advance in **EVF** is a novel technique to support modular *external* visitors. Modular external visitors are able to control traversals with direct access to the data structure being traversed, allowing *dependent* operations to be defined modularly without the need of advanced type system features. To make **EVF** practical, the framework employs annotations to automatically generate large amounts of boilerplate code related to visitors and traversals. 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. Using **EVF** we are able to create a modular *software product line* (SPL) of the TAPL interpreters, enabling sharing of large portions of code and features. The TAPL software product line contains several modular operations, which would be non-trivial to define with standard Object Algebras.

1998 ACM Subject Classification D.1.5 Object-oriented Programming, D.3.3 Language Constructs and Features, D.3.4 Processors

Keywords and phrases Visitor Pattern, Object Algebras, Modularity, Domain-Specific Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.29

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.10

1 Introduction

New Programming or Domain-Specific Language (DSL) implementations are needed all the time. However creating new languages is hard! There are two major factors that contribute to

* This work has been sponsored by the Hong Kong Research Grant Council projects number 27200514 and 17258816.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 29; pp. 29:1–29:32

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


such difficulty: 1) the amount of *implementation effort*; and 2) the need for *expert knowledge* in language design/implementation. A lot of implementation effort is involved in the creation and the maintenance of a language. A programming language consists of various components: syntactic and semantic analyzers, a compiler or interpreter, and tools that are used to support the development of programs in that language (e.g. IDE's or debuggers). Furthermore, the language has to be maintained, bugs have to be fixed, and new features have to be implemented. In addition to those engineering problems, software engineers lacking proper training miss the knowledge to do good language design. Because of these two factors, the costs for creating a new language are usually prohibitive, or it is hard to find engineers with the right skills for doing programming language implementation.

One way to address those challenges is to *reuse language components*. Programming languages share a lot of features. This is the case with Java or C, for example. Both languages have mechanisms to declare variables, support basic arithmetic operations as primitives, have loop constructs, or have similar scoping rules for variables. Moreover, nearly all new languages or DSLs will "copy" many features from existing languages, rather than having a completely new set of features. Therefore, there is conceptual reuse in programming languages. Unfortunately, it is hard to materialize the *conceptual reuse* into *software engineering reuse* in existing programming languages.

A simple way to achieve "reuse" is to copy and paste the code for an existing compiler and modify that. While this may be relatively effective (if the existing compiler is well-written), it duplicates code. Changes to the original compiler (bug fixes, new features, refactorings, etc.) will very likely be difficult to apply to the derived compiler. Because of that, the code of the two compilers often diverges, leading to duplication. So, reuse by copy&paste only works in the initial phase. At later stages, reuse becomes harder as careful synchronization of changes in both code bases is needed.

Many researchers have noticed the problems of copy&paste for reuse before. A popular approach to reuse of language components is offered by some language workbenches [18, 13, 15, 22]. Language workbenches aim at rapid prototyping of languages and related programming tools. The importance of modularity, reuse and composition of languages in language workbenches is well-recognized in the community. Erdweg et al. [15] mention "*Reuse and composition of languages, leading to language-oriented programming both at the object level and metalevel*" as one of the three key trends observed in the field of language workbenches. Indeed many language workbenches use *syntactic meta-programming* techniques to create language implementations and tools. One of the earliest uses of meta-programming techniques was the ASF+SDF approach to language composition [30]. In ASF+SDF it is possible to construct a library of language definition modules. Various other language workbenches (e.g. Spoofax [29] or Neverlang [54]) use similar techniques to modularize language definitions. However, while this simple syntactic modularization approach works, it lacks the desirable properties of separate compilation and modular type-checking.

An alternative approach to reuse of language components is offered by design patterns [19] that work on mainstream programming languages. Until recently it was thought that limitations in mainstream languages prevented or significantly complicated reusing language components in a modularly type-safe way. Indeed, well-known (minimalistic) challenges such as the Expression Problem (EP) [59], were created to illustrate such difficulties. However, recent research [53, 39, 52, 7, 41, 60] has shown that mainstream languages do allow for relatively practical solutions to the EP.

One such solution is the so-called Object Algebras [41]. Object Algebras provide a generalization of abstract factories and can solve challenging modularity problems, including

the EP. Object Algebras fully preserve separate compilation and modular type-checking. Nevertheless, solving the EP is just a step towards providing reuse of language components. Reusability in realistic language components requires addressing other modularity challenges, which the EP does not account for. Although significant progress [20, 27, 43, 50, 62] has been made towards scaling up Object Algebras to more realistic language components, there are still obstacles that need to be overcome.

A particular problem with Object Algebras is that they force a programming style similar to Church encodings [8] or functional programming folds [26]. While this structure works for many practical operations, certain operations are hard to express and/or incur on performance penalties. An example is *capture-avoiding substitution* [47], which poses two major challenges: 1) it is typically implemented with a top-down algorithm, which may not require traversing the full term, if shadowing exists; and 2) it *depends* on another operation that collects *free variables* from a term to avoid capture. Object Algebras are naturally suited for bottom-up algorithms that do a full traversal of the term. Simulating top-down algorithms with Object Algebras is possible but can be cumbersome and penalizing in terms of performance. Moreover, dependencies have to either be encoded via tuples with a heavy encoding [41] or require sophisticated type system features not available in Java [43, 50].

This paper presents **EVF**: an extensible and expressive Java Visitor framework. **EVF** is helpful to modularize *semantic components* of programming languages that operate on *Abstract Syntax Trees* (ASTs). Examples of such semantic components include: interpreters, compilers, pretty printers, various program analyses, and several optimizations and transformations over ASTs. **EVF** semantic components are just *standard Java programs* that are type-checked and separately compiled by the Java compiler. With **EVF**, library writers can develop such semantic language components modularly for various programming language features. Users can then simply choose the required features for the language and reuse the semantic components from the libraries, possibly with some new language constructs added for their specific purpose. In other words, **EVF** allows users to develop *Software-Product Lines* (SPLs) [10] of semantic language components.

The visitors in **EVF** generalize Object Algebras, and enable writing programs using a *generally recursive style* rather than folds. The use of such generally recursive style enables users to write programs more naturally. The support for extensibility improves on techniques used by *Object Algebras*, and on *modular visitors* [40]. It is known that Object Algebras are closely related to internal visitors [44]: a simple, but less expressive, variant of visitors related to Church encodings of datatypes [6]. The key advance in **EVF** is a novel technique to support modular *external* visitors that works in Java-like languages. In contrast to internal visitors, external visitors [6, 44] are based on Parigot encodings of datatypes [46]: a more expressive form of encodings that enables direct control of traversals, and *modularly expressing dependencies*.

To alleviate programmers from writing large amounts of boilerplate related to ASTs and AST traversals, **EVF** employs annotations to automatically generate such code. In essence, a user needs only to specify an Object Algebra interface, which describes the desired structure. **EVF** processes that interface and generates various useful interfaces and classes. Noteworthy are **EVF**'s generic queries and transformations, which generalize **Shy**-style traversals [62] and remove the limitation of bottom-up only traversals.

Overall, while there is a cost associated to learning the framework, **EVF** helps in reducing both the implementation effort and the required knowledge for programming language implementations through reuse. Essentially, through reuse, the more complex and intricate parts of several algorithms can be moved to properly encapsulated library code. Thus, we

believe the benefits of using **EVF** outweigh the costs of learning it. Section 3 shows a detailed example on how reuse can lower complexity in language implementation.

To further illustrate the applicability of **EVF** we conduct a case study refactoring many non-modular interpreters from the "Types and Programming Languages" (TAPL) book [47]. Using **EVF** we are able to create a modular SPL of the TAPL interpreters, enabling sharing large portions of code and features. Our programming language SPL contains several modular operations, which would be non-trivial to define with standard Object Algebras.

In summary, the contributions of this paper are:

- **A new approach to modular external visitors:** We present a novel technique to support modular *external* visitors that works in Java-like languages. The new technique allows modular visitor components to be expressed using a generally recursive style.
- **Simpler modular dependent operations:** Previous attempts to modular dependent operations either require a lot of boilerplate or sophisticated features not available in Java-like languages. Modular external visitors solve this problem with simple generics.
- **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.
- **Code generation for AST 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 non-trivial and non-modular OCaml code base into modular and reusable Java code. **EVF** and the case study are available at: `https://github.com/wxzh/EVF`

2 Modular External Visitors

This section provides the background and presents the key technical idea: a form of *external visitors* which is modular/extensible, offers control over traversals and only requires a simple form of generics for its implementation.

2.1 Background: Internal/External Visitors and Object Algebras

The origins of Object Algebras go back to the relationship between type-theoretic encodings of datatypes and the Visitor pattern. The original connection was established by Buchlovsky and Thielecke [6]. They pointed out that variants of the Visitor pattern correspond to different types of type-theoretic encodings. So-called *internal visitors* correspond to (typed) Church encodings of datatypes [5], whereas *external visitors* correspond to Parigot encodings of datatypes [46, 44].

Internal Visitors and Object Algebras. A simple example of internal visitors is shown in Figure 1. The example models a basic form of arithmetic expressions consisting of only two constructs: integer literals and addition. The interface `Alg<E>` models the visitor interface. The two methods (`Lit` and `Add`) model the so-called *visit methods*. Object Algebras use *exactly* the same interface as internal visitors [41]. In the context of Object Algebras, we would refer to the interface as an *Object Algebra interface*. The point at which internal visitors and Object Algebras differ is on how to create ASTs. Internal visitors use an interface `Exp` which contains an `accept` method. Then, for each language construct, there is a class

```
interface Alg<E> {
  E Lit(int n);
  E Add(E e1, E e2);
}
interface Exp {
  <E> E accept(Alg<E> v);
}
class Lit implements Exp {
  int n;
  public <E> E accept(Alg<E> v) {
    return v.Lit(n);
  }
}
class Add implements Exp {
  Exp e1, e2;
  public <E> E accept(Alg<E> v) {
    return v.Add(e1.accept(v), e2.accept(v))
    ;
  }
}
```

Figure 1 Code for internal visitors.

```
interface EVis<E> {
  E Lit(int n);
  E Add(EExp e1, EExp e2);
}
interface EExp {
  <E> E accept(EVis<E> v);
}
class ELit implements EExp {
  int n;
  public <E> E accept(EVis<E> v) {
    return v.Lit(n);
  }
}
class EAdd implements EExp {
  EExp e1, e2;
  public <E> E accept(EVis<E> v) {
    return v.Add(e1, e2);
  }
}
```

Figure 2 Code for external visitors.

that implements such interface. Object Algebras do not use such interface. Instead, they construct expressions directly through instances of the `Alg<E>` interface.

A concrete example of an operation on arithmetic expressions is evaluation. Evaluation is defined as a visitor (or Object Algebra), which implements `Alg<Integer>`:

```
class Eval implements Alg<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(Integer e1, Integer e2) { return e1 + e2; }
}
```

External Visitors. Figure 2 shows the equivalent code for modeling arithmetic expressions written with external visitors. The interface `EVis` plays the same role as `Alg`. However, differently from `Alg`, in `EVis` the `Add` method takes two expressions as arguments. There is also similar code for defining the type of expressions, and the two classes that implement expressions. Because of the different signature for the `Add` method in `EVis`, the definition of the `accept` method in `EAdd` is different as well. Instead of calling the `accept` method in the two subexpressions (`e1` and `e2`) and passing the result to `Add`, the new code passes the subexpressions directly to `Add`. In other words, external visitors offer control over the traversal of the term to the visitor implementation. For example, when defining evaluation, the `Add` method now calls the `accept` method and computes the result:

```
class EEval implements EVis<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(EExp e1, EExp e2) { return e1.accept(this) + e2.accept(this);
    }
}
```

2.2 Internal versus External Visitors

To better compare the advantages and disadvantages of internal and external visitors, lets consider an extension to expressions with subtraction and conditionals.

Extension using Internal Visitors. With internal visitors (or Object Algebras) it is simple to create an interface, which extends the original algebra interface for arithmetic expressions:

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

The extension includes two new constructs for the language: subtraction (`Sub`), and a simple form of conditional expressions (`If`). For simplicity, the condition evaluates to a number with 0 representing false, and any other number representing true. With such extended visitor interface, writing an extended evaluator is, at first sight, quite easy:

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) { return e1 - e2; }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3; // WRONG!!
  }
}
```

Problem 1: Lack of Control in Internal Visitors. The `Sub` case is trivial. However, the definition for `If` expressions is clearly wrong. Moreover, it is not possible to find a correct definition *without changing how the visitor is instantiated*. All methods in the visitor receive the results of evaluation as an argument: they cannot control when to (recursively) evaluate expressions. This works very well when the computation being expressed *traverses the full term* in a purely *bottom-up* manner. Unfortunately, for conditionals this is a problem, since only one branch needs to be evaluated. The implementation in `ExtEval`, however, evaluates both branches. This not only is problematic for performance reasons, but it is the wrong thing to do if the language being implemented supports, for example, some form of side-effects.

The lack of control problem is not fundamental, but it significantly complicates programming in practice and may introduce performance penalties. Previous work has shown how to correctly model far more complicated constructs and languages using Object Algebras [20, 27, 43, 50, 62]. However, this is done by changing the way Object Algebras are instantiated and using more complex techniques. Instead of choosing `Integer` as the instantiation for the type parameter of `Alg`, a different type, which suitably delays evaluation, is used. Several other problems also arise from the lack of control problem. For example, expressing dependent (non-compositional) operations (i.e. operations which are modularly defined in terms of other operations) is very inconvenient. To express such kinds of operations tuples can be used in Java, but this requires the definition of a lot of boilerplate code [41]. Another approach is to use *intersection types* [11, 48] with a special *merge operator* [51] to perform composition. This requires a type system more powerful than Java. Scala does support intersection types and it is possible to encode a weak form of a merge operator [43, 50], but the lack of support for a native merge operator in Scala complicates code and limits the scalability of the approach.

All in all, the lack of control problem in Object Algebras is a well-known, more than 80-year-old problem. When Church discovered Church encodings in the untyped lambda

calculus [8], he realized that certain operations were quite difficult to express. The most famous instance of that is the predecessor function on Church numerals. When Church tried to define the predecessor function on Church numerals, it first appeared impossible to define. Eventually, he realized that it is possible to encode the predecessor function using a pair, which performs computation bottom-up and rebuilds the original term. While such an algorithm does compute the predecessor function, it is much more complicated than the traditional predecessor function, and it takes linear time to compute, instead of being a constant time operation. Since Church's work, various other programming techniques have been based on Church encodings [26, 23, 42, 7], but the essential difficulties in expressing certain operations remained. Object Algebras are no different. Being essentially Church encodings, similar difficulties arise for certain operations, and similar workarounds apply, as Section 3 further illustrates.

Problem 2: Lack of Extensibility with External Visitors. The obvious attempt to solve the limitations of Object Algebras is to turn to external visitors, which do allow control over the traversal. However, if we try to do the same extension with external visitors we face a different problem: it is no longer possible to simply extend the original visitor interface.

In order to account for the extension with subtraction and conditionals, we have to change or copy&paste existing code for visitors. In other words, the visitor code is non-modular (unlike the code for Object Algebras). Different interfaces are necessary for the extension:

```
interface MVis<E> {
  E Lit(int x);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}
```

In external visitors, the visitor interface depends on the AST type and vice-versa. Since the new AST nodes for subtraction and conditionals require a visitor type that is aware of the new nodes, it is not possible to use the old interface `EExp`. Instead, a new interface `MExp` is needed with an `accept` method taking a richer type of visitors. Correspondingly, the visitor interface has to be changed. The `Add` method no longer takes expressions of type `EExp` as arguments. Instead, it now requires expressions of type `MExp`. When defining evaluation, the code for `EEval` cannot be reused either. Thus, the code for `Lit` and `Add` has to be essentially repeated in `MEval`:

```
class MEval implements MVis<Integer> {
  public Integer Lit(int n) { return n; }
  public Integer Add(MExp e1, MExp e2) { return e1.accept(this) + e2.accept(this);
    }
  public Integer Sub(MExp e1, MExp e2) { return e1.accept(this) - e2.accept(this);
    }
  public Integer If(MExp e1, MExp e2, MExp e3) {
    return !e1.accept(this).equals(0) ? e2.accept(this) : e3.accept(this);
  }
}
```

However, the implementation of `If` is now correct! Because external visitors delegate the control over traversals to the implementation of visitors, the expressions for the **`then`** and **`else`** branches only need to be evaluated when the suitable condition applies. Therefore, unlike internal visitors, no workarounds are necessary to implement the operation.

```
interface AVis<R,E> {
  E Lit(int x);
  E Add(R e1, R e2);
  E visitExp(R e);
}
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
interface CVis<E>
    extends AVis<CExp,E> {
  default E visitExp(CExp e) {
    return e.accept(this);
  }
}
```

```
class CLit implements CExp {
  int n;
  public CLit(int n) { this.n = n; }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Lit(n);
  }
}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}
```

Figure 3 Modular external visitors with abstracted recursive calls.

2.3 Key Idea: Abstracting Recursive Calls

To solve both problems we propose a new type of external visitors that abstracts the recursive calls. Figure 3 presents the code for the original arithmetic expressions encoded with the new visitor. Compared to `EVis`, this new visitor interface `AVis` has two changes. First, it uses an additional type parameter `R` to decouple itself from any concrete expression type. This first difference is known in the literature [43], and has been used in the past to provide generalized versions of Object Algebras. However, the second, and more important difference is the introduction of a new method `visitExp` that abstracts the recursive calls. Like the `accept` method in the VISITOR pattern, `visitExp` allows programmers to explicitly control recursive calls. In fact, calls to `visitExp` are essentially indirect calls to `accept`. Readers familiar with type-theoretic encodings of datatypes may find that the use of the `visitExp` method reminiscent of Mendler encodings of datatypes [34]. Indeed in Mendler-encodings of datatypes programmers can also control recursive calls with a function argument. However, as we shall discuss in Section 7 Modular External Visitors have significant differences to Mendler-style encodings.

The interface that provides the implementation for `visitExp` (which just calls `accept`) is `CVis`. Programmers will define their own visitors by implementing the other visit methods. When an actual visitor instance is needed, a class extending both the user-defined visitor and `CVis` is created. Thus, we end up with code which is essentially equivalent to non-modular external visitor code. Code for defining the AST hierarchy is very similar to non-modular external visitors. The `accept` method takes a `CVis` instance which extends `AVis` with `R` specified as `CExp`.

Evaluation with Control. `AEval` implements the evaluator for the expression language, where `R` is still a type parameter while `E` is instantiated to `Integer`:

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int n) { return n; }
  default Integer Add(R e1, R e2) { return visitExp(e1) + visitExp(e2); }
}
```

The evaluation on subexpressions of `Add` are now controlled via `visitExp`. However, `visitExp` should remain *abstract* for retaining the extensibility on `AEval`. `AEval` is hence modeled as an

interface with `Lit` and `Add` implemented using Java 8 *default methods*. An additional step to instantiate `AEval` as a class is needed for the purpose of creating objects. This can be done through defining a class that implements both `AEval` and `CVis`:

```
class CEval implements AEval<CExp>, CVis<Integer> {}
```

Then we are able to evaluate an expression using an instance of `CEval`:

```
CExp e = new CAdd(new CLit(1), new CLit(2));
e.accept(new CEval()); // 3
```

Modular Extension. As the dependency on the AST type has been removed from visitors, modular extensions on the visitor interface and its concrete implementations are enabled:

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
interface AEvalExt<R> extends AEval<R>, AVisExt<R,Integer> {
  default Integer Sub(R e1, R e2) { return visitExp(e1) - visitExp(e2); }
  default Integer If(R e1, R e2, R e3) {
    return !visitExp(e1).equals(0) ? visitExp(e2) : visitExp(e3);
  }
}
```

However, a remaining problem is that we still need a new AST infrastructure for the extension:

```
interface CExpExt { <E> E accept(AVisExt<CExpExt,E> v); }
interface CVisExt<E> extends AVisExt<CExpExt,E> {
  default E visitExp(CExpExt e) { return e.accept(this); }
}
... // 4 classes elided including Lit and Add
```

Discussion. The following table summarizes the strength and weakness of each approach:

Approach	Modular Visitor	Modular AST	Traversal Control
Object Algebras	Yes	Yes	No
Internal Visitors	Yes	No	No
External Visitors	No	No	Yes
Modular External Visitors	**Yes**	**No**	**Yes**

Object Algebras and internal visitors do not offer traversal control. Nevertheless, both visitor code and code for creating ASTs is modular in Object Algebras. The reason why the code for creating ASTs is modular in Object Algebras is that ASTs are created directly using an instance of the Object Algebra interface [41]. In contrast, all visitors (whether internal or external) require AST interfaces (such as `Exp`, `EExp`, `MExp`, `CExp`), and corresponding classes implementing those interfaces. However such AST class hierarchies are not reusable in extensions. External visitors provide traversal control at the price of losing modularity on the visitor code. Modular external visitors retain traversal control and bring modularity to the visitor code, but AST code is still not modular.

While AST code is still non-modular with modular external visitors, in practice, it is the visitor code that is important to modularize. The visitor code is what programmers

e	::=	x	variable	$FV(x)$	=	$\{x\}$
		$\lambda x.e$	abstraction	$FV(\lambda x.e)$	=	$FV(e) \setminus \{x\}$
		$e\ e$	application	$FV(e_1\ e_2)$	=	$FV(e_1) \cup FV(e_2)$
		i	literal	$FV(i)$	=	$\varnothing$
		$e - e$	subtraction	$FV(e_1 - e_2)$	=	$FV(e_1) \cup FV(e_2)$

(a) Syntax **(b)** Free variables

$[x \mapsto s]x$	=	s	
$[x \mapsto s]y$	=	y	if $y \neq x$
$[x \mapsto s](\lambda x.e)$	=	$\lambda x.e$	
$[x \mapsto s](\lambda y.e)$	=	$\lambda y.[x \mapsto s]e$	if $y \neq x \wedge y \notin FV(s)$
$[x \mapsto s](e_1\ e_2)$	=	$[x \mapsto s]e_1\ [x \mapsto s]e_2$	
$[x \mapsto s]i$	=	i	
$[x \mapsto s](e_1 - e_2)$	=	$[x \mapsto s]e_1 - [x \mapsto s]e_2$	

(c) Substitution

Figure 4 Formalization of the untyped lambda calculus.

will write to define various operations over ASTs. The AST code is mechanical and can be automatically generated, which is precisely one of the things that the **EVF** framework does. Programming with modular external visitors is, in some sense, similar to programming with algebraic datatypes in functional languages. That is, programmers can control the code for functions defined by pattern matching (similar to visitors) but not the code for constructors (similar to AST code).

3 EVF for Modularity and Reuse of PL Implementations

This section introduces the **EVF** framework by modeling the untyped lambda calculus. We are going to implement two operations, free variables and capture-avoiding substitution, where the latter depends on the former. Based on modular external visitors introduced in Section 2, **EVF** allows such dependency to be expressed in a simple way. **EVF** further complements modular external visitors by automatically generating boilerplate code related to ASTs and AST traversals. Compared with the implementations with traditional (non-modular) visitors and Object Algebras, the **EVF** implementation has advantages in terms of simplicity, modularity, and reusability. The section finishes with a discussion on how **EVF** reduces both the implementation effort and the need for specialized knowledge for language implementation.

3.1 Untyped Lambda Calculus: A Running Example

Figure 4 formalizes the untyped calculus: its syntax and two operations, free variables and capture-avoiding substitution, following Pierce's definition [47].

Syntax. The language has 5 syntactic forms: variables, lambda abstractions, lambda applications, integer literals and subtractions. The meta variable e ranges over expressions; x over names; i over integers. With the syntax, the operational semantics can be defined.

Free Variables. A variable in an expression is said to be *free* if it is not bound by any enclosing abstractions. The operation $FV(e)$ collects the set of free variables from an expression e. The definition relies on some set notations. Their meanings are: $\varnothing$ 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. 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, $y \neq x$, takes care of shadowing introduced by the abstraction. The second condition, $y \notin FV(s)$, avoids free variables in s being captured after substitution. For example, $[x \mapsto y](\lambda x.x)$ and $[x \mapsto y](\lambda y.x)$ have no effect because of the first and the second condition respectively. Thus, the two conditions together preserve the meaning of an expression after substitution.

3.2 A Summary of the Implementations and Results

We implemented the untyped lambda calculus using the Visitor pattern, Object Algebras and **EVF** respectively (full code can be found online). The table below summarizes the implementations from modularity, the source lines of code (SLOC) and the number of cases to implement for each operation:

Approach	Modular	Syntax	Free Variables		Substitution	
		SLOC	SLOC	# Cases	SLOC	# Cases
The Visitor Pattern	No	46	20	5	22	5
Object Algebras (w/ **Shy**)	Yes	7	12	2	55	5
EVF	Yes	7	12	2	13	2

From the table we can see that the **EVF** implementation is best in all these aspects. It is modular and uses least SLOC and number of cases to implement both operations. The comparison between the Visitor pattern and **EVF** shows the power of meta-programming. By generating AST and AST traversals automatically, **EVF** eliminates a large portion of SLOC. On the other hand, the comparison between Object Algebras and **EVF** illustrates the expressiveness of **EVF**. Object Algebras bypass the concrete representation of an AST structure, making the syntax definition simple. The definition of free variables in Object Algebras is as short as that in **EVF** with the help of the **Shy** framework [62], which generates traversal templates similar to **EVF**. However, substitution does not fit any **Shy** template, and worse it is very cumbersome to define with Object Algebras causing the expansion of SLOC. The remainder of this section explains the three implementations and the results in detail.

3.3 An Implementation with the Visitor Pattern

We first discuss an implementation with the (external) Visitor pattern presented in Figure 5.

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

```
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);
  }
}
... // 3 classes elided
```

```
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());
  }
  ... // 3 cases elided
}
class SubstVar implements LamAlg<Exp> {
  String x;
  Exp s;
  SubstVar(String x, Exp s) {
    this.x = x; this.s = s;
  }
  public Exp Var(String y) {
    return y.equals(x) ? s : new Var(x);
  }
  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));
  }
  ... // 3 cases elided
}
```

Figure 5 Untyped lambda calculus with the VISITOR pattern.

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

Substitution. Similarly, the class `SubstVar` models substitution. Substitution is a transformation over the expression structure. We hence instantiate the abstract type of `LamAlg` 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 is not restricted to be `this` and can indeed 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`.

Summary. 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 it requires substantial amounts of code, including AST classes and code for each of the 5 language constructs for both free variables and substitution.

```
@Algebra 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);
}
class FreeVars implements
    LamAlgQuery<Set<String>> {
  public Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
  public Set<String> Var(String x)
    {
    return Collections.singleton(x)
    ;
  }
  public Set<String> Abs(String x,
    Set<String> e) {
    return e.stream()
      .filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
}
interface IFV {
  Set<String> FV();
}
interface ISV<Exp> {
  Exp before();
  Exp after();
}
```

```
class SubstVar<Exp extends IFV>
    implements LamAlg<ISV<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 ISV<Exp> Var(String y) {
    return new ISV<Exp>() {
      public Exp before() {
        return alg.Var(y);
      }
      public Exp after() {
        return y.equals(x) ? s : alg.Var(y);
  }};}
  public ISV<Exp> Abs(String y, ISV<Exp> e) {
    return new ISV<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());
  }};}
  ... // 3 cases elided
}
```

Figure 6 Untyped lambda calculus with Object Algebras.

3.4 An Implementation with Object Algebras

Next, we discuss an implementation with Object Algebras shown in Figure 6.

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 both recursive arguments and return values are of the abstract type `Exp`. Note that `LamAlg` is annotated with `@Algebra` provided by the **Shy** framework. Through annotation processing, **Shy** will generate traversal templates for `LamAlg`.

Free Variables. Operations over the language are defined as Object Algebras, which are implementations of the `LamAlg` interface. The Object Algebra `FreeVars` 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. Also, by using **Shy**, the number of cases to implement is reduced to 2. The `LamAlgQuery` template provides a default implementation for each case by using a client-supplied monoid instance. Regarding `FreeVars`, it should return an empty set for a base case or unite the intermediate sets from subtrees for an intermediate case. By supplying a set monoid and overriding the variable and the abstraction case, we are able to give the complete definition for `FreeVars`.

Substitution. Modeling substitution using Object Algebras is tricky. There are two major difficulties: 1) expressing the dependency on free variables modularly; 2) substitution traverses the expression structure in a flexible way, and not in a purely bottom up manner. For the first difficulty, we define an interface `IFV` and set it as the upper bound of the `Exp`. This way, we are able to call `FV` on the expression `s`. For the second difficulty, a similar technique to that employed in defining the predecessor function on Church numerals is applied (see discussion in Section 2.2). Instead of just returning the expression after substitution, we also keep track of the original expression. The pair-like interface `ISV` 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 `ISV<Exp>`, we can call `before` or `after` for obtaining the expression before and after substitution.

Summary. Although capture-avoiding substitution is possible to model using Object Algebras, the implementation is rather inefficient and complicated. The dependency on free variables is pushed to the successor algebra that is applied after `SubstVar`, which requires additional boilerplate for composing that algebra with `FreeVars`. Unlike the implementation of free variables, which can benefit from the **Shy** framework to reduce the number of cases, the definition of substitution does not fit any of the traversal templates offered by the **Shy** framework. Thus 5 cases are needed for substitution.

3.5 An Implementation with EVF

The corresponding implementation of the untyped lambda calculus with **EVF** is given by Figure 7. **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 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 with new infrastructure generated.

From Object Algebras to Modular Visitors. **EVF** is used to complement code written with modular external visitors with code generation. Modular external visitor interfaces are the basis of the generated code. However, users of **EVF** do not need to write such modular external visitor interfaces directly. Instead, **EVF** allows clients to write the traditional Object Algebra interfaces, as done for example in lines 1-7 of Figure 7. Since it is possible to automatically generate a modular external visitor interface from an Object Algebra interface, this is done automatically by **EVF**. This is good for users because Object Algebra interfaces are simpler than modular external visitor interfaces. Figure 8 shows the corresponding modular external visitor interface generated for `LamAlg`. Note that `GLamAlg` is parameterized by two types `Exp` and `OExp`, where `Exp` captures 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`. We leave the discussion on the technical details to Section 4.

Code Generation for Structural Traversals. Neither `FreeVars` nor `SubstVar` extend `GLamAlg` directly. Instead, they extend the generated traversal templates `LamAlgQuery` and `LamAlgTransform` respectively. Similarly to **Shy**, **EVF** supports various traversal patterns that can be used to remove boilerplate code. The implementation of `FreeVars` using **EVF** is close to that using Object Algebras. One difference is that subexpressions are of abstract

```
1  @Visitor interface LamAlg<Exp> {
2    Exp Var(String x);
3    Exp Abs(String x, Exp e);
4    Exp App(Exp e1, Exp e2);
5    Exp Lit(int n);
6    Exp Sub(Exp e1, Exp e2);
7  }
8  interface FreeVars<Exp> extends LamAlgQuery<Exp,Set<String>> {
9    default Monoid<Set<String>> m() {
10     return new SetMonoid<>();
11   }
12   default Set<String> Var(String x) {
13     return Collections.singleton(x);
14   }
15   default Set<String> Abs(String x, Exp e) {
16     return visitExp(e).stream().filter(y -> !y.equals(x))
17       .collect(Collectors.toSet());
18   }
19 }
20 interface SubstVar<Exp> extends LamAlgTransform<Exp> {
21   String x();
22   Exp s();
23   Set<String> FV(Exp e);
24   default Exp Var(String y) {
25     return y.equals(x()) ? s() : alg().Var(y);
26   }
27   default Exp Abs(String y, Exp e) {
28     if (y.equals(x())) return alg().Abs(y, e);
29     if (FV(s()).contains(y)) throw new RuntimeException();
30     return alg().Abs(y, visitExp(e));
31   }
32 }
```

Figure 7 Complete code for the untyped lambda calculus with **EVF**.

AST type `Exp` and we call `visitExp` explicitly to trigger the traversal on subexpressions, e.g. in line 16. The ability to control the traversal makes a great difference in defining `SubstVar`. **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`. As a result, the implementation of `SubstVar` is not only simpler and more efficient than the one with Object Algebras, but it also requires only the explicit definition of 2 cases (instead of 5) due to **EVF**'s ability to reuse more flexible traversal templates. In Section 4.3 we will give formal specifications of the traversal templates and introduce more forms of traversal patterns.

Modular Dependent Visitors. The support for *external* visitors allows modular dependent operations to be defined with simple generics. For example, to express the dependency on free variables in the definition of substitution, we declare an abstract method `FV` in line 23 of Figure 7, which takes an expression and returns a set of free variables. Then we are able to collect the free variable set from `s` by calling `FV` in line 29. The reader may have noticed that `FV` and `FreeVars`'s `visitExp` share the same signature. In fact, `FV` is implemented by calling `visitExp` on an instance of `FreeVars`. But the coupling with peer visitors such as `FreeVars` are deferred to the instantiation stage of the dependent visitor, as we will see next. This

```
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);
}
```

Figure 8 Generated modular external visitor interface for the untyped lambda calculus.

```
1  class FreeVarsImpl implements FreeVars<CExp>, LamAlgVisitor<Set<String>> {}
2  class SubstVarImpl implements SubstVar<CExp>, LamAlgVisitor<CExp> {
3    String x;
4    CExp s;
5    public SubstVarImpl(String x, CExp s) { this.x = x; this.s = s; }
6    public String x() { return x; }
7    public CExp s() { return s; }
8    public Set<String> FV(CExp e) { return new FreeVarsImpl().visitExp(e); }
9    public LamAlg<CExp> alg() { return new LamAlgFactory(); }
10 }
11 public class LC {
12   public static void main(String[] args) {
13     LamAlgFactory alg = new LamAlgFactory();
14     CExp exp = alg.App(alg.Abs("y", alg.Var("y")), alg.Var("x")); // (\y.y) x
15     new FreeVarsImpl().visitExp(exp); // {"x"}
16     new SubstVarImpl("x", alg.Lit(1)).visitExp(exp); // (\y.y) 1
17   }
18 }
```

Figure 9 Instantiation and client code for the untyped lambda calculus.

simple reuse mechanism improves the modularity of visitors significantly, and can be used together with OO inheritance for modularity and extensibility. This is in contrast with the Object Algebras approach, which requires significant complexity to deal with dependencies.

Instantiation and Client Code. Abstract recursive calls and modular dependencies prevent visitors from being modeled as concrete classes. An additional step for instantiation is necessary for object creation. We use *interfaces* and *default methods* to define visitors and to make them extensible by exploiting Java 8 multiple interface inheritance. **EVF** generates `LamAlgVisitor`, an interface that extends `GLamAlg` with `visitExp` implemented. Line 1 and lines 2-10 of Figure 9 illustrate how to instantiate `FreeVars` and `SubstVar` using the generated `LamAlgVisitor`. The dependencies declared in `SubstVar` must be fulfilled. For example, in line 8, we call the `visitExp` method on an `FreeVarsImpl` instance to realize the `FV` method.

Concrete AST Representation. Different from conventional Object Algebras, the construction and interpretation of an AST are separated in **EVF**. An AST infrastructure like that in Figure 5 is automatically generated by **EVF**. The generated factory class, `LamAlgFactory`, is exposed to the clients for constructing ASTs. Once created, an AST will reside in memory and is able to accept different visitors to traverse itself. For example, we construct an AST of form $(\lambda y.y)\ x$ in line 14. By invoking the `visitExp` method defined on visitor instances, we traverse the same AST using `FreeVars` and `SubstVar` in line 15 and 16 respectively.

```
@Visitor ExtLamAlg<Exp> extends LamAlg<Exp> {
  Exp Bool(boolean b);
  Exp If(Exp e1, Exp e2, Exp e3);
}
interface ExtFreeVars<Exp> extends ExtLamAlgQuery<Exp,Set<String>>, FreeVars<Exp>
    {}
interface ExtSubstVar<Exp> extends ExtLamAlgTransform<Exp>, SubstVar<Exp> {}
```

Figure 10 Untyped lambda calculus with extensions.

3.6 Discussion

Suppose we wish to implement a larger language based on the untyped lambda calculus. Instead of defining everything from scratch, we can easily build this language through reusing existing **EVF** components, as illustrated by Figure 10. The annotated Object Algebra interface `ExtLamAlg` extends `LamAlg` with constructs for boolean values and if-expressions. To support free variables and substitution for this extended language, we can simply compose existing components defined for `LamAlg` (`FreeVars` and `SubstVar`) with newly generated templates for `ExtLamAlg` (`ExtLamAlgQuery` and `ExtLamAlgTransform`). We can even combine more features via multiple interface inheritance. Of course, similar instantiation code shown in Figure 9 is needed for the client code.

We discuss the strength and weakness of PL implementations using **EVF** here:

1. **Modularity:** Like Object Algebras, **EVF** components are modular, extensible and type-safe. This means that it is possible to create *libraries* of language components. For example, the implementations of the untyped lambda calculus can be put in a library, and be reused in implementations of larger programming languages that include the untyped lambda calculus. This is simply not possible (in a type-safe way) with an implementation based on traditional (non-modular) visitors. In other words, modularity enables the creation of SPLs of language components.
2. **Reduction of Implementation Effort:** A direct consequence of modularity is that implementation effort can be reduced through reuse. In **EVF** there are two different mechanisms which support reuse:
   - **Reuse from Extensibility:** A larger language can extend the existing operations and define only cases for the new language constructs. As the above example shows, for defining a new language that extends the untyped lambda calculus, only the cases for the extended constructs would be defined by the programmer.
   - **Reuse from Traversal Templates:** Many operations, including free variables and substitution are *structure-shy*. That is, in most cases the definition is a congruent recursive traversal of the children. Only a few cases (variables and binders) are actually defining interesting behavior. Thus, traversal templates significantly reduce the number of cases that needs to be written by language implementers. Indeed, if an extension to the untyped lambda calculus does not have new binders or types of variables like the above example, programmers do not need to define any new cases for free variables and substitution: they get an automatic implementation from the traversal templates.
3. **Reduction of Knowledge about PL Implementations:** Reuse enables moving complex aspects of PL implementations to library code. For example, it is well-known that capture-avoiding substitution is a rather subtle operation to define. If PL implementers can simply reuse implementations of such operations, they do not need to understand the tricky details of the operation. With **EVF** any language extensions that do not involve new types of binders or variables, do not require users to understand how capture-avoiding substitution works.

- **Syntax of Object Algebra Interfaces**

L	::=	$\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}$	Object Algebra interfaces
C	::=	$X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}$	constructors
I	::=	$A\texttt{<}\overline{X}\texttt{>}$	interface types
T	::=	$X \mid \texttt{int} \mid \texttt{boolean} \mid \ldots$	argument types

- **Translation Scheme**

$$[\![\texttt{@Visitor interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}]\!] = \texttt{interface}\ [\![I_0]\!]\ \texttt{extends}\ [\![\overline{I}]\!]\,\texttt{\{}\ [\![\overline{C}]\!]\ \mathsf{visitX}_{in}(I_0)\texttt{\}}$$
$$[\![A\texttt{<}\overline{X}\texttt{>}]\!] = \texttt{G}A\texttt{<}\overline{X}\texttt{,[O}X \mid X \in \mathsf{allX}_{in}(\mathsf{AT}(I))\texttt{]>}$$
$$[\![X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}]\!] = \texttt{O}X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}$$

- **Auxiliary Definitions**

$$\mathsf{returntype}(X\ c\texttt{(}\ \overline{T}\ \overline{x}\texttt{);}) = X$$
$$\mathsf{allX}_{in}(\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}) = \{\mathsf{returntype}(C) \mid C \in \overline{C}\} \cup \bigcup_{I \in \overline{I}} \mathsf{allX}_{in}(\mathsf{AT}(I))$$
$$\mathsf{newX}_{in}(\texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\,\texttt{\{}\ \overline{C}\texttt{\}}) = \mathsf{allX}_{in}(\mathsf{AT}(I_0)) \setminus \bigcup_{I \in \overline{I}} \mathsf{allX}_{in}(\mathsf{AT}(I))$$
$$\mathsf{visitX}_{in}(I) = [\texttt{O}X\ \texttt{visit}X\texttt{(}X\ \texttt{x);} \mid X \in \mathsf{newX}_{in}(\mathsf{AT}(I))]$$

Figure 11 Translation from Object Algebra interfaces to modular visitor interfaces.

There are also two main limitations of the **EVF** framework:

1. **Learning Effort:** The definitions of **EVF** visitors may not be very intuitive at first glance. It takes some effort from users to learn the modular visitor encoding, various traversal templates and how to instantiate visitors.
2. **Boilerplate Instantiation:** Although most boilerplate code is eliminated by **EVF**, there is still some left. Visitors have to be instantiated manually before they can be actually used, which may require significant amounts of code (see Figure 9 for example).

4 Code Generation in EVF

To facilitate development using modular external visitors, **EVF** automatically generates a lot of boilerplate code related to ASTs and AST traversals. This section gives the details about the generated code in a formal way.

4.1 Modular External Visitor Interfaces

It is cumbersome for users to directly write down the modular external visitor interfaces, especially when multiple sorts are needed. This motivates us to let **EVF** automatically translate a conventional Object Algebra interface into its corresponding modular external visitor interface. A generated modular external visitor interface has been shown in Figure 8. Figure 11 formalizes the translation.

Syntax of Object Algebra Interface. We first give the grammar of standard 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 Object Algebra interface table (AT) that maps an Object Algebra interface type I to its declaration L.

Translation Scheme. Translation rules are defined using semantic brackets ($[\![\cdot]\!]$). The bracket notation $[\mathsf{f}(A) \mid A \in \overline{A}]$ 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 $\{\mathsf{f}(A) \mid A \in \overline{A}\}$ is similar to the bracket notation except that it collects a set of elements while preserving their order.

The fundamental step of the translation is to separate *input types* from the type parameter list. We classify a type parameter as an *input 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 `visitX` method. The translation scheme consists of three main steps. First, we find out all input types and augment the type parameter list with their corresponding output types. Second, the return types of the constructors are replaced by output types. Last, the `visitX` methods are generated for new input types.

Auxiliary Definitions. The translation scheme relies on auxiliary definitions: returntype gets the return type of a constructor (considered as an input type); allX_{in} collects all input types from the interface hierarchy; newX_{in} collects input types that are not introduced by super interfaces; finally, visitX_{in} generates one `visitX` method for each input type.

4.2 AST Infrastructure

Each modular visitor interface should have the corresponding AST infrastructure for instantiation and client code. However, such AST infrastructure is non-modular and tedious to write, as we have seen in Section 2. This is because whenever extending a modular visitor, we have to define a new AST hierarchy representing both newly introduced constructs as well as *all* existing constructs. Fortunately, **EVF** automatically generates such infrastructure for us. For example, the following code shows the generated AST infrastructure for the untyped lambda calculus:

```
public interface Exp { <OExp> OExp accept(GLamAlg<Exp,OExp> v); }
public interface LamAlgVisitor<OExp> extends GLamAlg<Exp,OExp> {
  default OExp visitExp(Exp e) { return e.accept(this); }
}
public class LamAlgFactory implements LamAlg<Exp> {
  public Exp Var(String x) {
    return new Exp() {
      public <OExp> OExp accept(GLamAlg<Exp,OExp> v) {
        return v.Var(x);
  }};}
  public Exp Abs(String x, Exp e) {
    return new Exp() {
      public <OExp> OExp accept(GLamAlg<Exp,OExp> v) {
        return v.Abs(x, e);
  }};}
  ...
}
```

The code is slightly different from the code shown in Figure 3. Instead of generating one class per construct, **EVF** generates a concrete factory `LamAlgFactory` that implements the Object Algebra interface (abstract factory). `LamAlgFactory` exposes one factory method for each construct, which not only simplifies the creation of ASTs (without using `new` all the time) but also can be used for instantiating modular transformations. For example, line 9 and line 13-14 in Figure 9 illustrate the use of `LamAlgFactory`.

4.3 Boilerplate Traversals

AST traversals often contain a lot of boilerplate code. To address that problem the **Shy** framework [62] provides a number of boilerplate traversals automatically for Object Algebras.

EVF also supports boilerplate traversals just as **Shy** does, but it generalizes them to modular external visitors. Notably, and unlike **Shy**, boilerplate traversals in **EVF** are not restricted to be bottom-up. We have seen how such traversals help in eliminating boilerplate code in Section 3. In this section, we formalize two core traversal templates and additionally introduce a novel type of traversal pattern. Other **Shy** templates like contextual transformations are omitted for space reasons, but they are essentially variations of these core templates.

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 is handy for defining operations with a lot of cases sharing the same behavior. Consider the untyped calculus again. We may want to inspect the form of an expression, for example whether it is a literal. It would be tedious to define such an operation because we have to define a lot of repetitive cases - all cases except for `Lit` return a `false`. With the `LamAlgDefault` template, however, we only need to supply a default value (`false`) once via implementing the `m` method instead of giving each of those repetitive cases an implementation manually:

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

Now we give the template of queries with default values formally. Given an Object Algebra interface A, let $\mathbb{X}$ denote the input types of A where $\mathbb{X} = \mathsf{allX}_{in}(\mathsf{AT}(A))$. The template is:

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

interface A0Default<X̄0,O> extends GA0<X̄0,O,…,O>, ĀDefault<X̄,O> {
  Zero<O> m();
  default O c(T̄ x̄) { return m().empty(); }
}
```

(the $|\mathbb{X}_0|$ occurrences of `O` in `GA0<X̄0,O,…,O>`)

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

Queries by Aggregation. Another form of query traverses the whole AST and aggregates a value. Recall the definition of `FreeVars` shown in Figure 7. It uses the template `LamAlgQuery`. The template for queries by aggregation is given below:

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

interface A0Query<X̄0,O> extends GA0<X̄0,O,…,O>, ĀQuery<X̄,O> {
  Monoid<O> m();
  default O c(T̄ x̄) {
    return { m().empty();                                  if ∄T ∈ T̄ ∧ T ∈ 𝕏0,
           { Stream.of([visitT(x) | T ∈ T̄ ∧ T ∈ 𝕏0])       otherwise.
           {        .reduce(m().empty(),m()::join);
  }
}
```

(the $|\mathbb{X}_0|$ occurrences of `O` in `GA0<X̄0,O,…,O>`)

The `Monoid` interface can not only provide the default value through the `empty` method inherited from `Zero`, but also exposes a `join` method for combining intermediate results.

`Query` gives different implementations to a constructor according to whether it is a primitive (i.e. no argument of any input types) or a combinator. If the constructor is a primitive, the result is `m().empty()`; otherwise corresponding `visitX` methods get called on recursive arguments and their results are combined using `m().join()`. For example, in the definition of `FreeVars`, the generic `SetMonoid` class is used for fulfilling the `m` dependency where `empty` returns an empty set and `join` is the union of two sets:

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

Transformations. Transformations are operations that transform an AST to another AST. Transformations use a factory to construct another AST that is further transformed or consumed. Recall the definition of `SubstVar` shown in Figure 7. It uses the transformation template `LamAlgTransform` for eliminating boilerplate code. The general template for transformations is given below:

```
interface A0Transform<X̄0> extends GA0<X̄0,𝕏0>, ĀTransform<X̄,𝕏> {
  A0<X̄0> alg();
  default X c(T̄ x̄) { return alg().c(visitT(T, x)‾); }
}
```

In `Transform` the output types are the same as input types, reflecting the essence of a transformation. An auxiliary definition visit_T is needed, which transforms an argument only when it is of any input types:

$$\mathsf{visit}_T(T, x) = \begin{cases} \texttt{visit}T(x) & \text{if } T \in \mathbb{X}_0, \\ x & \text{otherwise.} \end{cases}$$

5 Case Study

To reveal the utility of **EVF**, we implemented a large number of interpreters from TAPL [47]. TAPL is a good benchmark for modularity mainly because it contains a dozen of languages, where subsequently defined languages are extensions of the previously defined ones. The original implementation in OCaml[1] is, however, non-modular. Using **EVF** we are able to create a modular SPL of the TAPL interpreters, enabling sharing large portions of code and features. Our programming language SPL contains several modular operations, which would be non-trivial to define with standard Object Algebras.

5.1 Overview

Terms and types are the main data structures for modeling languages, on which 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 (lambdas, records, references, exceptions, etc.) and combines them with some of existing features to form various languages. However,

[1] `https://www.cis.upenn.edu/~bcpierce/tapl`

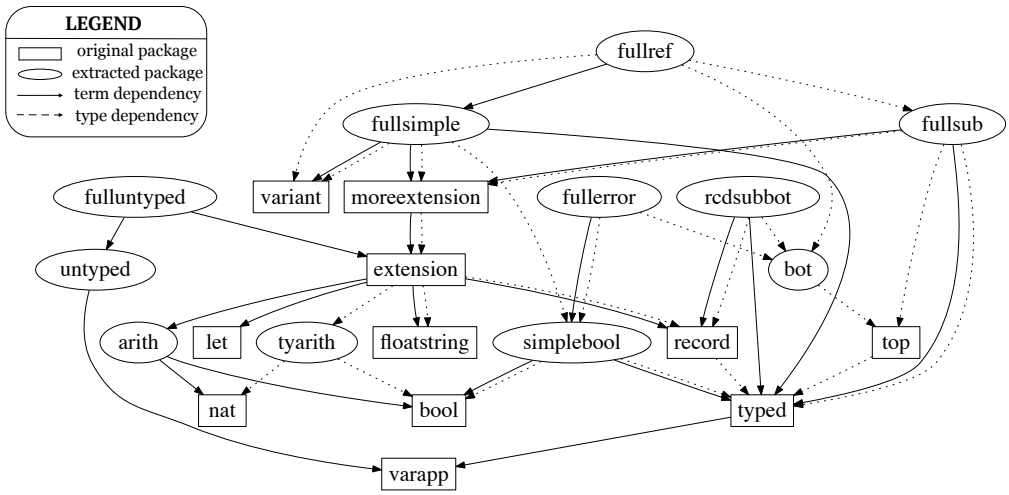


Figure 12 Package dependency graph.

due to the use of algebraic datatypes in OCaml, "combining" features is actually done through copy&paste, causing modularity issues. **EVF**, on the other hand, is equipped with modular composition mechanisms and can compose features without code duplication.

Figure 12 gives a bird's-eye view of the **EVF** implementation of TAPL. To enhance modularity, we extract conceptually independent features into separate packages for reuse. In Figure 12, original packages are represented using boxes and extracted packages are represented using ellipses. The interactions among languages are explicitly revealed by the arrows. For example, *bool* is an extracted language representing booleans and conditionals, on which *arith* and *simplebool* are built.

Composable Language Implementations. According to the criteria set by Erdweg et al. [14], **EVF** has a good support for language composition. 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, *arith* unifies *nat* and *bool* with an extension (`TmIsZero`) that supports testing whether a term is zero or not:

```
@Visitor 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 their respective `TermAlg`. From Figure 12 we can see that *arith*, as an extension, is further composed by *extension*. This kind of composability retains on operations as well.

Multiple Sorts. The case study also illustrates how multi-sorted languages can be defined using **EVF**. The demand for multiple sorts arises when a term needs a type in its definition. For instance, *typed* implements the typed lambda calculus, whose `TermAlg` is multi-sorted:

```
@Visitor interface TermAlg<Term,Ty> extends varapp.TermAlg<Term> {
```

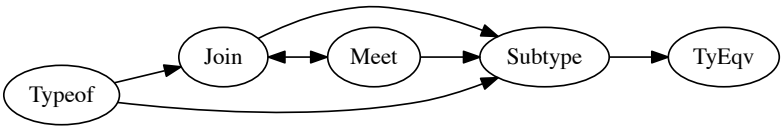
```
  Term TmAbs(String x, Ty ty, Term t);
}
```

The abstraction (`TmAbs`) requires its argument of a specific type. Here we use another type parameter `Ty` to loosely capture the dependency on types and model types in a separate Object Algebra interface:

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

The reason to separate types and terms is that they belong to different syntactic categories in the typed lambda calculus, on which two completely different sets of operations are defined. It would make no sense to have a small-step evaluator on types or defining subtyping relations on terms. This separation makes visitors fine-grained, allowing independent extensibility on both types and terms.

Dependent Operations. A key difference between TAPL and other case studies conducted on modularity is that operations in TAPL may have complex dependencies. An instance is the typechecking function, which has complex dependencies in the presence of subtyping:



The typechecker `Typeof` directly depends on `Join` and `Subtype` for calculating the least supertype of the two branches of an if-expression and performing a subtype check between the calculated type against the expected type respectively. `Join` and `Subtype` have their own dependencies on `Meet` and `TyEqv`. `Meet` in turn depends on `Join`, making the dependency circular. Such complex dependencies pose difficulties in modularizing `Typeof`. Fortunately, **EVF** makes the implementation of `Typeof` straightforward using similar dependency mechanism presented in Section 3.

5.2 Components

De Bruijn Indices. In TAPL, de Bruijn indices are used in languages based on the lambda calculus for modeling binder-related constructs. As opposed to the nominal representation used in Section 3, a variable is represented by a natural number, denoting the distance from the closest binder to its corresponding binder. For example, the nominal term $\lambda x.\lambda y.(x\ y)$ corresponds to the de Bruijn term $\lambda.\lambda.(1\ 0)$. From the implementation point of view, de Bruijn indices have many advantages, making it simpler to define substitution and α-equivalence. However, terms represented in de Bruijn indices become less readable and not so intuitive to manipulate. Hence, operations on de Bruijn indices are a good candidate to be part of the library so that end users can enjoy benefits of de Bruijn indices without being bothered by their technical details. We encapsulate these operations including shifting and substitution as **EVF** components and put into the extracted *varapp* package. To reuse de Bruijn indices and their associated operations elsewhere, an **EVF** user can easily reuse these components in their own languages via some glue code similar to Figure 10. On the other hand, an OCaml user would have to copy&paste the code snippet and modify it accordingly.

Constant Function Elimination. Optimizations are another suitable source of candidates to be modeled as components. The reason is that an optimization typically focuses on a small set of language constructs with a fixed algorithm. By implementing optimizations as **EVF** components, their complexity is hidden and they can be easily adapted elsewhere.

We added constant function elimination [32] to the TAPL case study for demonstration purposes. An abstraction $\lambda x.e_1$ is a *constant function* if x is not used in e_1. Then, an application $(\lambda x.e_1)\ e_2$ can be safely replaced by e_1 while retaining the semantics. Our goal is to eliminate all such constant functions in a term. This optimization is nontrivial to define as it has several dependencies.

First, we need to extract the body from an abstraction:

```
interface GetBodyFromTmAbs<Term> extends TermAlgDefault<Term,Optional<Term>> {
  default Zero<Optional<Term>> m() { return () -> Optional.empty(); }
  default Optional<Term> TmAbs(String x, Term t) { return Optional.of(t); }
}
```

By using the `TermAlgDefault` template, only the abstraction case needs to be explicitly defined. Next, we check whether the variable introduced by the abstraction is used in the body:

```
interface IsVarUsed<Term> extends TermAlgQueryWithCtx<Integer,Boolean,Term> {
  default Monoid<Boolean> m() { return new OrMonoid(); }
  default Function<Integer, Boolean> TmVar(int x, int n) { return c -> x == c; }
  default Function<Integer,Boolean> TmAbs(String x, Term t) {
    return c -> visitTerm(t).apply(c+1);
  }
}
```

The traversal template `TermAlgQueryWithCtx` is a variant of queries by aggregation, which additionally takes a context in recursive calls. Finally, we are able to define the optimization:

```
interface ConstFunElim<Term> extends TermAlgTransform<Term> {
  Term termShift(int d, Term t);
  Optional<Term> getBodyFromTmAbs(Term t);
  Boolean isVarUsed(int i, Term t);
  default Term TmApp(Term e1, Term e2) {
    Term e = visitTerm(e1);
    return getBodyFromTmAbs(e)
      .map(t -> isVarUsed(0, t) ? alg().TmApp(e, visitTerm(e2)) : termShift(-1, t))
      .orElse(alg().TmApp(e, visitTerm(e2)));
  }
}
```

`ConstFunElim` traverses the AST top down. When a `TmApp` is found, it will first optimize `e1` to be `e` and then extract the body `t` from `e` using `getBodyFromTmAbs`. Next we check whether the variable is used in the body via `isVarUsed`. If not, the whole expression will be replaced by `t` with its de Bruijn indices decreased by 1 using `termShift`. Otherwise, the optimization continues on `e2` and wraps the optimized `e1` and `e2` back to `TmApp`. Constant function elimination as well as various other operations (including the small-step evaluators) would be tricky to model using Object Algebras because they are in essence top-down operations. However, with modular external visitors such operations are easy to model and traversal templates can be used to eliminate boilerplate on them.

5.3 Evaluation

To evaluate **EVF**'s implementation of the case study, we compare to the original OCaml implementation. Table 1 compares SLOC (excluding blank lines and comments) of the **EVF**

Table 1 SLOC statistics **EVF** vs OCaml: A package perspective.

Extracted Package	EVF	Original Package	EVF	OCaml	% Reduced
bool	98	arith	33	102	68%
extension	34	bot	61	184	67%
floatstring	104	fullerror	105	366	72%
let	47	fullref	247	880	72%
moreextension	106	fullsimple	83	651	88%
nat	103	fullsub	116	628	82%
record	198	fulluntyped	47	300	85%
top	86	rcdsubbot	39	255	85%
typed	138	simplebool	38	211	77%
utils	172	tyarith	26	135	78%
varapp	65	untyped	46	128	61%
variant	161	**Total**	**2153**	**3840**	**44%**

implementation[2] with the non-modular OCaml version. The left-hand side counts SLOC of the extracted packages and the right-hand side compares SLOC of the original packages. Although an OOP language like Java is considerably more verbose than a functional language like OCaml, **EVF**'s implementation reduces 44% of SLOC counting all packages, thanks to modularity and code generation techniques. The reduction of SLOC for each original package is on average 76%. For feature-rich languages like *fullsimple*, the reduction is even more dramatic and can be up to 88%. The reason is that all these original packages reuse features from 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 12 we can see that features like the 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 is not that straightforward: **EVF**'s implementation has dependencies whereas the OCaml implementation is stand-alone. Table 2 does the comparison from the component perspective which sums the SLOC of two core components, AST definitions and small-step evaluators, for all packages. The results show that both SLOC are reduced significantly, which explains why the total SLOC of **EVF** is reduced.

As discussed in Section 3, the drawback of **EVF** components is an additional step for instantiation. The SLOC needed for instantiating an operation is proportional to the number of dependencies it has. To measure the instantiation overhead, we count the SLOC of instantiation per original package. The statistics show that the SLOC grows together with the language. Concretely, the SLOC for the simplest (*arith*), the medium (*simplebool*) and the largest (*fullref*) languages are 26, 63 and 109. The reason is that feature-rich languages support more operations and/or their supported operations have more dependencies.

6 Performance Measurements

This section gives the preliminary performance measurements on **EVF**. The novel `visitX` methods introduced by **EVF** add one more level of dispatching to the standard Visitor

[2] We count only the files *core.ml* and *syntax.ml*, excluding the parser, the REPL and etc.

Table 2 SLOC statics **EVF** vs OCaml: A component perspective.

Component	EVF	OCaml	% Reduced
AST Definition	85	231	64%
Small-step Evaluator	263	481	46%

Table 3 Performance.

Approach	Time (ms)
Imperative Visitor	133
Functional Visitor	163
Runabout	278
EVF	262

pattern, which causes some execution overhead. To have a rough idea about the impact of the `visitX` methods on performance, we run a microbenchmark adapted from [45]. We compare ourselves with respect to the two variants of the VISITOR pattern [6]: *imperative visitors* and *functional visitors*. An imperative visitor uses side effects to do the computation whereas a functional visitor computes a result via return values. We also compare ourselves to *Runabout* [21], 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 can be found online. The benchmark programs 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 3 summarizes the run time of each approach. The results show that imperative visitors are fastest among the four approaches. The functional visitor implementation ran slower than the imperative visitor approach due to the heavy use of recursion. One more layer of indirection brings additional performance penalty to **EVF**, which takes about double of the time with respect to the imperative visitor but still outperforms the Runabout. Of course, more rigorous and extensive benchmarks need to be performed to validate the results.

7 Related Work

Extensible Visitors. Early work on the VISITOR pattern [31, 58, 45] pointed out extensibility limitations of 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 [45] proposed a generic class *Walkabout* as the root of visitors. By using Java's runtime reflection, the Walkabout removes the need for `accept` methods in AST types. This decouples the AST type from the visitor interface, allowing new variants to be introduced as well. Unfortunately, the extensive use of introspection causes severe performance penalties. Based on the Walkabout, Grothoff proposed *Runabout* [21], attempting to achieve reasonable performance through sophisticated bytecode generation and caching. Forax's *Sprintabout* [17] further improves the performance of Runabout by eliminating the manual creation of AST infrastructure. However, Walkabout and its successors are not type-safe. Torgersen [53] developed variations of the VISITOR pattern to solve the Expression Problem [59]. 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 [44, 40, 24] using advanced Scala

type system features, our work applies similar techniques but requires only simple generics available in Java. The `visitX` methods in modular external visitor 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* [38]. 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 [45] 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 annotation processing 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 [57] ported ideas from the rewriting system Stratego [56] 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* and hence can not be directly mapped to **EVF**. We would like to explore a library of visitor combinators in **EVF** as future work.

Object Algebras and Church Encodings. Various programming techniques have been inspired by Church encodings in the past. Hinze [23] firstly Church-encoded datatypes using type classes in Haskell. Based on Hinze's work, Oliveira et al. [42] presented solutions to the EP using type classes. Carette et al. [7] and Hofer et al. [25] further illustrated the applicability of those techniques for defining interpreters and embedded DSLs. Another well-known solution to the EP is "Data types à la carte" (DTC) [52]. DTC represents a data type as a functor, where a type parameter is used for capturing recursive occurrences of that data type, enabling extensibility. A type-level fixpoint is defined for tying the knot. As discussed in detail in Section 2, Church encodings suffer from lack of traversal control. A variant of Church encodings called Mendler encoding [34] offers recursion control. Delaware et al. [12] combines Mendler encodings and DTC to develop modular meta-theory. Technically speaking, Modular external visitors differ from Mendler-style encodings in that they require recursive types. The use of recursive types is unproblematic in Java and it is the key for dealing with dependencies and achieving more efficient traversals. Mendler encodings, on the other hand, do not rely on recursive types, but cannot deal with dependencies and (just as regular Church encodings) suffer from efficiency problems. In object-oriented programming, *Object Algebras* [41] are also a modular design pattern based on Church encodings. Object Algebras solve the recursion control problem by instantiating Object Algebra interface using thunks [41]. Improved support for dependencies for Object Algebras have been proposed [43, 50]. Unfortunately, this cannot be ported to Java as more sophisticated features are required. Other problems such as no concrete AST representation hinder the practical use of Object Algebras [20]. **EVF** visitors solve these problems with only simple generics, thus eliminating the need for various techniques used with Object Algebras.

Component-Based Language Development. The idea of constructing languages by assembling components dates back to the 1980s [30]. Most closely related is Mosses's work on component-based semantics [36]. The idea is to provide a collection of highly reusable *fundamental constructs* (funcons) with predefined semantics [9]. 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) [35]. Later work on Implcitly MSOS (I-MSOS) [37] deals with the context propagation problem, further improving the modularity and reusability of semantics specification. From MSOS/I-MSOS specifications, interpreters can be derived [49]. Similar funcons can also be developed as **EVF** components.

Language Workbenches. Language workbenches are aimed at lowering the amount of effort to develop new languages. Examples of modern, mature language workbenches include: Xtext [13], MPS [18], Spoofax [29]. At the moment some language workbenches and other tools provide support for code reuse through syntactic modularization techniques, based on meta-programming and code generation. For example, DynSem [55] is a DSL integrated into Spoofax for generating interpreters from I-MSOS like specifications. Such techniques allow language 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. Recent work on MontiCore [22] generates both the visitor and the AST infrastructure from the grammar specification. MontiCore allows two dimensions of extensibility. The extensibility on data variants is achieved through making the extended AST types subtypes of the initial AST types, and overriding the `accept` methods inherited from the initial AST types appropriately. MontiCore automatically overrides the `accept` methods by checking the runtime type of the visitor instance and casting it to the most specific one. Moreover, since the `accept` methods are overloaded in extended AST types, the compiler gives no warning when an initial visitor is applied to an extended AST, leading to unexpected behavior. The technique is quite similar to Krishnamurthi et. al's [31] early solution to extensible visitors. Like their solution, Monticore's approach does not fully support modular type-checking, due to the use of casts. **EVF** provides a different approach to the composition of *semantic* language components that fully supports type-safe extensibility, as well as separate compilation. Unlike MontiCore, **EVF** generates different AST infrastructures for different visitor interfaces and requires no casts. Hence, the compiler will capture the mismatch between the visitors and the AST. However, **EVF** does not support modularization of *syntactic* language components (such as grammars and/or parsers) for the moment. An interesting venue for future work would be to integrate the **EVF** techniques into a language workbench, such as MontiCore.

Software Product-Lines. Software Product-Lines (SPLs) [10, 33, 28] 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 [4], AHEAD [3], FeatureC++ [2] and FeatureHouse [1]. 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.

8 Conclusion

We have presented **EVF**: an extensible and expressive Java Visitor framework. **EVF**'s support for modular 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 develop an annotation processor to generate boilerplate code. Users only need to annotate the Object Algebra interfaces. Then all the infrastructure will be automatically generated, including ASTs and AST traversals. The TAPL case study demonstrates the applicability and benefits of **EVF** in reducing both implementation effort and the need for specialized PL implementation knowledge. Currently, **EVF** users have to instantiate visitors manually. One line of future work is to investigate automatic instantiation of visitors. Similar instantiation problem has been identified by Wang and Oliveira [60] and solved by Wang et al. [61]. It may be possible to automatically instantiate visitors in **EVF** through a combination of family polymorphism [16] and the technique from [61]. Another avenue of future work is to use **EVF** in larger applications, such as compilers or program analysis tools.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments.

References

1 Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

2 Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: on the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, 2005.

3 Don Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

4 Don Batory and Bart J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

5 Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

6 Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155:309–329, 2006.

7 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

8 Alonzo Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

9 Martin Churchill, Peter D. Mosses, and Paolo Torrini. Reusable components of semantic specifications. In *Proceedings of the 13th International Conference on Modularity*, 2014.

10 Paul Clements and Linda Northrop. *Software product lines.* Addison-Wesley,, 2002.

11 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

12 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

13 Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

14 Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, 2012.

15 Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, 2013.

16 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, 2001.

17 Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of java extensions: the double-dispatch use-case. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

18 Martin Fowler. Language workbenches: The killer-app for domain specific languages, 2005. http://martinfowler.com/articles/languageWorkbench.html.

19 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addisson-Wesley, 1994.

20 Maria Gouseti, Chiel Peters, and Tijs van der Storm. Extensible language implementation with object algebras. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, 2014.

21 Christian Grothoff. Walkabout revisited: The runabout. In *European Conference on Object-Oriented Programming*, 2003.

22 Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In *European Conference on Modelling Foundations and Applications*, 2016.

23 Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5), 2006.

24 Christian Hofer and Klaus Ostermann. Modular domain-specific language components in scala. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*, 2010.

25 Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, 2008.

26 Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

27 Pablo Inostroza and Tijs van der Storm. Modular interpreters for the masses. In *Proceedings of the 2015 International Conference on Generative Programming: Concepts and Experiences*, 2015.

28 Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, 2011.

29 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

30 Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1993.

31 Shriram Krishnamurthi, Matthias Felleisen, and Daniel P Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, 1998.

32 Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

33 Roberto E Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*, 2005.

34 Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of pure and Applied logic*, 51(1-2):159–172, 1991.

35 Peter D Mosses. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:195–228, 2004.

36 Peter D Mosses. Component-based semantics. In *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, 2009.

37 Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, August 2009.

38 Martin E Nordberg III. Variations on the visitor pattern. *Ann Arbor*, 1996.

39 Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages*, 2005.

40 Bruno C. d. S. Oliveira. Modular visitor components. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

41 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

42 Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. *Trends in Functional Programming*, 7:199–216, 2006.

43 Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. Feature-oriented programming with object algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013.

44 Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 2008 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2008.

45 Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, 1998.

46 Michel Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

47 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

48 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

49 Casper Bach Poulsen and Peter D Mosses. Generating specialized interpreters for modular structural operational semantics. In *International Symposium on Logic-Based Program Synthesis and Transformation*, 2013.

50 Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. From object algebras to attribute grammars. In *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2014.

51 John C Reynolds. The coherence of languages with intersection types. In *International Symposium on Theoretical Aspects of Computer Software*, 1991.

52 Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

53 Mads Torgersen. The expression problem revisited – four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

54 Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

55 Vlad Vergu, Pierre Neron, and Eelco Visser. Dynsem: A dsl for dynamic semantics specification. In *26th International Conference on Rewriting Techniques and Applications*, 2015.

56 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, 2001.

57 Joost Visser. Visitor combination and traversal control. In *Proceedings of the 2001 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2001.

58 John Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, 1999.

59 Philip Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.

60 Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.

61 Yanlin Wang, Haoyuan Zhang, Bruno C d S Oliveira, and Marco Servetto. Classless java. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2016.

62 Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap your boilerplate with object algebras. In *Proceedings of the 2015 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, 2015.