Castor: Programming with Extensible Generative Visitors

Weixin Zhang[a,*], Bruno C. d. S. Oliveira[a]

[a]*The University of Hong Kong, Hong Kong, China*

Abstract

Much recent work on type-safe extensibility for Object-Oriented languages has focused on design patterns that require modest type system features. Examples of such design patterns include *Object Algebras*, *Extensible Visitors*, *Finally Tagless interpreters*, or *Polymorphic Embeddings*. Those techniques, which often use a functional style, can solve basic forms of the Expression Problem. However, they have important limitations.

This paper presents Castor: a Scala framework for programming with extensible, generative visitors. Castor has several advantages over previous approaches. Firstly, Castor comes with support for (type-safe) pattern matching to complement its visitors with a concise notation to express operations. Secondly, Castor supports type-safe interpreters (à la Finally Tagless), but with additional support for pattern matching and a generally recursive style. Thirdly, Castor enables many operations to be defined using an imperative style, which is significantly more performant than a functional style (especially in the JVM platform). Finally, functional techniques usually only support tree structures well, but graph structures are poorly supported. Castor supports type-safe extensible programming on graph structures. Key to Castor's usability is the use of annotations to automatically generate large amounts of boilerplate code to simplify programming with extensible visitors. To illustrate the applicability of Castor we present several applications and two case studies. The first case study compares the ability of Castor for modularizing the interpreters from the "Types and Programming Languages" book with previous modularization work. The second case study on UML activity diagrams illustrates the imperative aspects of Castor, as well as its support for hierarchical datatypes.

Keywords: modularity, visitor pattern, pattern matching, metaprogramming, OOP

1. Introduction

For many years researchers have been looking at improving modularity mechanisms in programming languages. A particular problem that is the focus of much recent work in modularity is the so-called Expression Problem [1]. In the Expression Problem, the key challenge is how to achieve *type-safe extensibility*. That is, how to: evolve software

*Corresponding author

Email addresses: zhangweixinxd@gmail.com (Weixin Zhang), bruno@cs.hku.hk (Bruno C. d. S. Oliveira)

in two dimensions (adding new variants and operations) without rewriting existing code; and without using type-unsafe features (such as casts or reflection). Over the years, many solutions were proposed. Some work proposes new programming languages or programming language features designed specifically with modularity in mind. These include *virtual classes* [2], *multi-methods* [3], and *family polymorphism* [4]. Other work has focused on more general language features – such as *generics* [5], *higher-kinded types* [6], *virtual types* [7], *traits* [8] and *mixins* [5] – which can also help with various modularity problems.

Much of the more recent work on type-safe extensibility for Object-Oriented languages focus is on design patterns that require modest type system features. Examples of such design patterns include *Object Algebras* [9], *Modular Visitors* [10], *Finally Tagless interpreters* [11] or *Polymorphic Embeddings* [12]. All of those techniques can solve basic forms of the Expression Problem, and are closely related.

The foundation for a lot of that work comes from functional programming and type-theoretic encodings of datatypes [13, 14]. In particular, the work by Hinze [15] was the precursor for those techniques. In his work Hinze employed so-called Church [13] and Scott [14] encodings of datatypes to model generic programming libraries. Later Oliveira et al. [16, 17] showed that variants of those techniques have wider applications and solve the Expression Problem [1]. These ideas were picked up by Carrete et al. [11] to enable tagless interpreters, while also benefiting from the extensibility properties of the techniques. Carrete et al.'s work popularized those applications of the techniques as the nowadays so-called *Finally Tagless* style. Soon after Hofer et al. [12] proposed *Polymorphic Embeddings* in Scala, highly inspired by the Finally Tagless style in languages like Haskell and OCaml.

In parallel with the work on *Finally Tagless* and *Polymorphic Embeddings* the connections of those techniques to the Visitor pattern in OOP were further explored [18], building on observations between the relationship between type-theoretic encodings of datatypes and visitors by Buchlovsky and Thielecke [19]. That work showed that Church and Scott encodings of datatypes correspond to two variants of the Visitor pattern called, respectively, *Internal* and *External* visitors. Later on Oliveira and Cook [9] showed a simplified version of *Internal Visitors* called *Object Algebras*, which could solve the Expression Problem even in languages like Java.

While *Internal Visitors*, *Object Algebras*, *Finally Tagless* or *Polymorphic Embeddings* can all be traced back to Church encodings, there has been much less work on techniques that are based on Scott encodings. Scott encodings are more powerful, as they allow a (generally) recursive programming style. In contrast, Church encodings rely on a programming style that is akin to programming with folds in functional programming [20]. In general, Scott encodings require more sophisticated type system features, which is one reason why they have seen less adoption. In particular recursive types are necessary, which also brings up extra complications due to the interaction of recursive types and subtyping. Nevertheless, recent work by Zhang and Oliveira [21] on the Java EVF framework picked up on modular *External Visitors* and shows *External Visitors* can be made practical even with modest language features and code generation. The applicability of EVF is demonstrated by refactoring interpreters from the "Types and Programming Languages" (TAPL) book [22]. The interpreters are modularized, and various specific interpreters are recovered from modular, reusable components. This effort is non-trivial because TAPL interpreters are written in a small-step operational semantics style, which does not fit well with folds. The fundamental problem is that the recursion pattern for small-step operational semantics is quite different from a fold. Furthermore, many operations employed by implementations of TAPL interpreters depend on other operations. Such *dependencies* are hard to model in a modular setting, but the use of EVF's *External Visitors* can account for them. However, there are still critical limitations on existing type-safe extensibility approaches, including EVF. One drawback is the lack of support for pattern matching, which makes writing various operations quite cumbersome. Another drawback is that even for the techniques that have been adapted to Object-Oriented Programming (OOP), the focus is still on a functional programming style. Writing operations in an imperative style is difficult, and supporting graph structures (which are common in OOP) is nearly impossible.

This paper presents CASTOR: an extensible and expressive Scala visitor framework. Unlike previous work, CASTOR aims to support not only a functional style but also an imperative programming style with visitors. CASTOR visitors bring several advantages over existing approaches:

Concise Notation. Programming with the VISITOR pattern is typically associated with a lot of boilerplate code. *Extensible Visitors* make the situation even worse due to the heavy use of sophisticated type system features. Although previous work on EVF alleviated the burden of programmers by generating boilerplate code related to visitors and traversals, it is restricted by Java's syntax and annotation processor. CASTOR improves on EVF by employing Scala's concise syntax and Scalameta[1] to simplify client code. Unlike Java's annotation processor, we can directly transform the client code with Scalameta, hiding the boilerplate and sophisticated type system features from users.

Pattern Matching Support. CASTOR comes with support for (type-safe) pattern matching to complement its visitors with a concise notation to express operations. We identify several desirable properties for pattern matching in an OOP context and show how existing approaches are lacking some of these properties (Section 2). We argue that the traditional semantics of pattern matching, which is based on the *order* of patterns and adopted by many approaches, conflicts with the openness of data structures. Therefore we suggest that a more restricted, top-level pattern matching model, where the order of patterns is irrelevant. To compensate for the absence of ordered patterns we propose a complementary mechanism for case analysis with defaults, which can be used when nested or multiple case analysis is needed.

GADT-Style Definitions. CASTOR supports type-safe interpreters (à la *Finally Tagless*), but with additional support for pattern matching and a generally recursive style. While *Finally Tagless* interpreters are nowadays widely used by programmers in multiple languages (including Haskell and Scala), they must be written in fold-like style. Supporting operations that require nested patterns, or simply depend on other operations is quite cumbersome (although workarounds exist [23]), especially if modularity is to be preserved. In contrast, CASTOR can support those features naturally.

[1] `http://scalameta.org`

Hierarchical Datatypes. Functional datatypes are typically flat where variants have no relationships with each other. Object-oriented style datatypes, on the other hand, can be hierarchical [24] where datatype constructors can be refined by more specific constructors. Hierarchical datatypes facilitate reuse since the subtyping relationship allows the semantics defined for supertypes to be reused in subtypes. CASTOR exploits OOP features and employs subtyping to model hierarchical datatypes.

Imperative Traversals. CASTOR enables many operations to be defined using an imperative style, which is significantly more performant than a functional style (especially in the JVM platform). Both functional and imperative visitors [19] written with CASTOR are fully extensible and can later support more variants modularly. Imperative visitors enable imperative style traversals that instead of returning a new Abstract Syntax Tree (AST), modify an existing AST in-place.

Graph Structures. Finally functional techniques usually only support tree structures well, but graph structures are poorly supported. CASTOR supports type-safe extensible programming on graph structures. Compared to trees, graphs are a more general data structure that have many important applications. In the domain of compilers, abstract semantic graphs can be used for representing shared subexpressions, which facilitate optimizations like common subexpression elimination.

In summary, this paper makes the following contributions:

- **Extensible pattern matching with modular external visitors:** We evaluate existing approaches to pattern matching in an OOP context (Section 2). We show how to incoorporate extensible (or open) pattern matching support on modular external visitors, which allows CASTOR to define non-trivial pattern matching operations.

- **Support for hierarchical datatypes:** Besides flat datatypes that are typically modeled in functional languages, we show how OOP style hierarchical datatypes is supported in CASTOR (Section 3).

- **Support for GADTs:** We show how to use CASTOR's support for GADTs in building well-typed interpreters (Section 4), which would be quite difficult to model in a *Finally Tagless* style.

- **Imperative style modular external visitors:** We show how to define imperative style modular external visitors in CASTOR (Section 5).

- **Support for graph structures:** We show how to do type-safe extensible programming on graph structures, which generalize the typical tree structures in functional programming (Section 5).

- **The CASTOR framework:** We present a novel encoding for modular pattern matching based on extensible visitors (Section 2.7). The encoding is automated using metaprogramming and the transformation is formalized (Section 6).

- **Case studies:** We conduct two case studies to illustrate the effectiveness of Castor. The first case study on TAPL interpreters (Section 7) demonstrates functional aspects of Castor, while the second one on UML activity diagrams (Section 8) demonstrates the object-oriented aspects of Castor.

This paper is a significantly extended version of a conference paper [25]. We revise the presentation the paper and more importantly extend Castor with novel features. Firstly, we add a detailed comparison with our previous work on EVF (Section 2.8). Secondly, we improve the way of declaring variants of open datatypes, which enables hierarchical variants (Section 3), GADTs (Section 4), graphs and imperative style visitors (Section 5). Thirdly, we revise the formalization on the new encoding (Section 6). Finally, we conduct an additional case study on UML activity diagrams (Section 8) for assessing these added features.

Source code for Castor and case studies is available at:

`https://github.com/wxzh/Castor`

2. Open Pattern Matching

Pattern matching is a pervasive and useful feature in functional languages (e.g. ML [26] and Haskell [27]) for processing data structures conveniently. Data structures are firstly modeled using algebraic datatypes and then processed through pattern matching. On the other hand, OOP uses class hierarchies instead of algebraic datatypes to model data structures. Still, the same need for processing data structures also exists in OOP. However, there are important differences between data structures modeled with algebraic datatypes and class hierarchies. Algebraic datatypes are typically *closed*, having a fixed set of variants. In contrast, class hierarchies are *open*, allowing the addition of new variants. A closed set of variants facilitates exhaustiveness checking of patterns but sacrifices the ability to add new variants. OO class hierarchies do support the addition of new variants, but without mechanisms similar to pattern matching, some programs are unwieldy and cumbersome to write. In this section, we first characterize four desirable properties of pattern matching in the context of OOP. We then review some of the existing pattern matching approaches in OOP and discuss why they fall in short of the desirable properties. This section ends with an overview of Castor and a comparison of the presented approaches.

2.1. Desirable Properties of Open Pattern Matching

We identify the following properties for pattern matching that are desirable in an OOP context:

- **Conciseness.** Patterns should be described concisely with potential support for wildcards, deep patterns, and guards.

- **Exhaustiveness.** Patterns should be exhaustive to avoid runtime matching failure. The exhaustiveness of patterns should be statically verified by the compiler and the missing cases should be reported if patterns are incomplete.

- **Extensibility.** Datatypes should be extensible in the sense that new data variants can be added while existing operations can be reused without modification or recompilation.
- **Composability.** Patterns should be composable so that complex patterns can be built from smaller pieces. When composing overlapped patterns, programmers should be warned about possible redundancies.

Using these properties as criteria, we next evaluate pattern matching approaches in OOP. We show that many widely used approaches lack some of these properties. We argue that a problem is that many approaches try to closely follow the traditional semantics of pattern matching, which assumes a closed set of variants. Under a closed set of variants, it is natural to use the *order* of patterns to prioritize some patterns over the others. However, when the set of variants is not predefined a priori then relying on some ordering of patterns is problematic, especially if separate compilation and modular type-checking are to be preserved. Nonetheless, many OO approaches, which try to support both an extensible set of variants and pattern matching, still try to use the order of patterns to define the semantics. Unfortunately, this makes it hard to support other desirable properties such as exhaustiveness or composability.

2.2. Running Example: ARITH

To facilitate our discussion, a running example from TAPL [22]—an untyped, arithmetic language called ARITH—is used throughout this paper. The syntax and semantics of ARITH are formalized in Figure 1. Our goal is to model the syntax and semantics of ARITH in a concise and modular manner.

ARITH has the following syntactic forms: zero, successor, predecessor, true, false, conditional and zero test. The definition *nv* identifies 0 and successive application of `succ` to 0 as numeric values. The operational semantics of ARITH is given in *small-step* style, with a set of reduction rules specifying how a term can be rewritten in one step. Repeatedly applying these rules will eventually evaluate a term to a value. There might be multiple rules defined on a single syntactic form. For instance, rules PREDZERO, PREDSUCC and PRED are all defined on a predecessor term. How `pred` *t* is going to be evaluated in the next step is determined by the shape of the inner term *t*. If *t* is a successor application to a numeric value, then PREDSUCC will be applied, etc.

ARITH is a good example for assessing the four properties because: 1) The small-step style semantics is best expressed with a concise *nested case analysis* on terms; 2) ARITH is, in fact, a unification of two sublanguages, NAT (zero, successor and predecessor) and BOOL (true, false, and conditional) plus an extension (zero test). Ideally, NAT and BOOL should be *separately defined* and *modularly reused*.

2.3. The VISITOR *Pattern*

The VISITOR design pattern [18] is frequently used to implement interpreters or compilers because of its ability to add new interpretations or compiler phases without modifying the class hierarchy. Let us implement the ARITH language using the VISITOR pattern step by step. The implementation is written in Scala without using any Scala-specific features and can be easily mapped to other OOP languages like C++ or Java.

$$
\begin{array}{lcl}
t & ::= & 0 \mid \texttt{succ}\ t \mid \texttt{pred}\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{iszero}\ t \\
nv & ::= & 0 \mid \texttt{succ}\ nv
\end{array}
$$

$$
\frac{t_1 \rightarrow t_1'}{\texttt{succ}\ t_1 \rightarrow \texttt{succ}\ t_1'}
\qquad
\frac{}{\texttt{pred}\ 0 \rightarrow 0}\ \textsc{PredZero}
$$

$$
\frac{}{\texttt{pred}\ (\texttt{succ}\ nv_1) \rightarrow nv_1}\ \textsc{PredSucc}
\qquad
\frac{t_1 \rightarrow t_1'}{\texttt{pred}\ t_1 \rightarrow \texttt{pred}\ t_1'}\ \textsc{Pred}
$$

$$
\frac{}{\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \rightarrow t_2}
\qquad
\frac{}{\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \rightarrow t_3}
$$

$$
\frac{t_1 \rightarrow t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \rightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}
\qquad
\frac{}{\texttt{iszero}\ 0 \rightarrow \texttt{true}}
$$

$$
\frac{}{\texttt{iszero}\ (\texttt{succ}\ nv_1) \rightarrow \texttt{false}}
\qquad
\frac{t_1 \rightarrow t_1'}{\texttt{iszero}\ t_1 \rightarrow \texttt{iszero}\ t_1'}
$$

Figure 1: The syntax and semantics of Arith.

Abstract Syntax. The abstract syntax of Arith is modeled by the following class hierarchy:

```
abstract class Tm {
  def accept[A](v: TmVisit[A]): A
}
class TmZero() extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmZero(this)
}
class TmSucc(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmSucc(this)
}
class TmPred(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmPred(this)
}
class TmTrue() extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmTrue(this)
}
class TmFalse extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmFalse(this)
}
class TmIf(val t1: Tm, val t2: Tm, val t3: Tm) extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmIf(this)
}
class TmIsZero(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmIsZero(this)
}
```

The abstract class `Tm` represents the datatype of terms, and syntactic constructs of terms are subclasses of `Tm`. A generic `accept` method is defined throughout the class hierarchy, which is implemented by invoking the corresponding lowercase *visit method* exposed by `TmVisit`.

Visitor Interface. `TmVisit` is the *visitor interface* that declares all the visit methods required by `accept` implementations. Its definition is given below:

```
trait TmVisit[A] {
  def tmZero(x: TmZero): A
  def tmSucc(x: TmSucc): A
  def tmPred(x: TmPred): A
  def tmTrue(x: TmTrue): A
  def tmFalse(x: TmFalse): A
  def tmIf(x: TmIf): A
  def tmIsZero(x: TmIsZero): A
}
```

`TmVisit` is parameterized by `A` for abstracting over the return type of visit methods. Each visit method takes an instance of its corresponding class and returns a value of `A`.

Concrete Visitors. Operations over `Tm` are *concrete visitors* that implement the visitor interface `TmVisit`. The numeric value checker is defined like this:

```
class Nv extends TmVisit[Boolean] {
  def tmZero(x: TmZero) = true
  def tmSucc(x: TmSucc)= x.t.accept(this)
  def tmPred(x: TmPred) = false
  def tmTrue(x: TmTrue) = false
  def tmFalse(x: TmFalse) = false
  def tmIf(x: TmIf) = false
  def tmIsZero(x: TmIsZero) = false
}
```

`Nv` implements `TmVisit` by instantiating the type parameter `A` as `Boolean` and giving an implementation to each visit method. Here, the interesting cases are `tmZero` and `tmSucc`. For the former, a `true` is returned; for the latter, we call `_.t.accept(this)` for recursively applying `Nv` to check the inner term. The remaining cases are not numeric values thus return `false`.

With `Nv` defined, we can now implement the small-step evaluation visitor:

```
class Eval1 extends TmVisit[Tm] { eval1 => // Dependency
  val nv = new Nv                          // Dependency
  def tmZero(x: TmZero) = throw NoRuleApplies
  def tmSucc(x: TmSucc) = new TmSucc(x.t.accept(this))
  def tmPred(x: TmPred) = x.t.accept(new TmVisit[Tm] {
    def tmZero(y: TmZero) = y                                 // PredZero
    def tmSucc (y: TmSucc) =
      if (y.t.accept(nv)) y.t                                 // PredSucc
      else new TmPred(y.t.accept(eval1))                      // Pred
    def tmPred(y: TmPred) = new TmPred(y.accept(eval1))       // Pred
    def tmTrue(y: TmTrue) = new TmPred(y.accept(eval1))       // Pred
    def tmFalse(y: TmFalse) = new TmPred(y.accept(eval1))     // Pred
    def tmIf(y: TmIf) = new TmPred(y.accept(eval1))           // Pred
    def tmIsZero(y: TmIsZero) = new TmPred(y.accept(eval1)) // Pred
  })
  def tmTrue(x: TmTrue) = throw NoRuleApplies
  def tmFalse(x: TmFalse)  = throw NoRuleApplies
  def tmIf(x: TmIf) = x.t1.accept(new TmVisit[Tm] {
    def tmTrue(y: TmTrue) = x.t2
```

```
    def tmFalse(y: TmFalse) = x.t3
    def tmZero(y: TmZero) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmSucc(y: TmSucc) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmPred(y: TmPred) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmIf(y: TmIf) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmIsZero(y: TmIsZero) = new TmIf(y.accept(eval1),x.t2,x.t3)
  })
  def tmIsZero(x: TmIsZero) = x.t.accept(new TmVisit[Tm] {
    def tmZero(y: TmZero) = new TmTrue
    def tmSucc (y: TmSucc) =
      if (y.t.accept(nv)) new TmFalse
      else new TmIsZero(y.accept(eval1))
    def tmPred(y: TmPred) = new TmIsZero(y.accept(eval1))
    def tmTrue(y: TmTrue) = new TmIsZero(y.accept(eval1))
    def tmFalse(y: TmFalse) = new TmIsZero(y.accept(eval1))
    def tmIf(y: TmIf) = new TmIsZero(y.accept(eval1))
    def tmIsZero(y: TmIsZero) = new TmIsZero(y.accept(eval1))
  })
}
```

The small-step evaluator rewrites a term to another thus `A` is instantiated as `Tm`. Since primitive cases are already values, we simply throw a `NoRuleApplies` exception for `tmZero`, `tmTrue` and `tmFalse`. Defining the case for `tmSucc` is easy too: we construct a new successor with its inner term rewritten by `eval1`. In contrast, defining `tmPred`, `tmIf` and `tmIsZero` is trickier because they all have multiple rules. Take `tmPred` for example. As a visitor recognizes only one level representation of a term, it is insufficient to encode rules that require nested case analysis. To further reveal the shape of the inner term, anonymous visitors are created. Rules like PredSucc can then be specified inside the `tmSucc` method of the inner visitor. Moreover, the inner visitor of `tmPred` depends on both `Nv` and `Eval1`. These dependencies are respectively expressed by the field `nv` and the synonym `eval1` for the outer `this`. Then we can pass `nv` or `eval1` as an argument to the `accept` method for using the dependency. Notice that the Pred rule is repeatedly implemented 6 times. A similar situation also happens inside `tmIf` and `tmIsZero`, making the overall implementation of `Eval1` quite lengthy.

Client Code. We can write some tests for our implementation of Arith:

```
// iszero (if false then true else pred (succ 0))
val tm = new TmIsZero(
  new TmIf(new TmFalse,new TmTrue,new TmPred(new TmSucc(new TmZero))))
val eval1 = new Eval1
val tm1 = tm.accept(eval1)  // iszero (pred (succ 0))
val tm2 = tm1.accept(eval1) // iszero 0
val tm3 = tm2.accept(eval1) // 0
```

where we construct a term using all syntatic forms of the Arith language and evaluate it step by step using `eval1`. The evaluation result of each step is shown in the comments on the right hand side.

Discussion of the Approach. The conventional Visitor pattern has been criticized for its *verbosity* and *inextensibility* [28, 29], which are manifested in the implementation of Arith. Programming with the Visitor pattern is associated with a lot of infrastructure code, including the visitor interface, the class hierarchy, etc. Writing such infrastructure manually is tedious and error-prone, especially when there are many classes involved. Such verbosity restricts the usage VISITOR pattern, as Martin [30] wrote:

> "*Often, something that can be solved with a* VISITOR *can also be solved by something simpler.*"

Moreover, the VISITOR pattern suffers from the Expression Problem [1]: it is *easy* to add new operations by defining new visitors (as illustrated by `nv` and `eval1`) but *hard* to add new variants. The reason is that `Tm` and `TmVisit` are tightly coupled. When trying to add new subclasses to the `Tm` hierarchy, it is not possible to implement their `accept` methods because there exist no corresponding visit methods in `TmVisit`. A non-solution is to modify `TmVisit` with new visit methods. As a consequence, all existing concrete implementations of `TmVisit` have to be modified in order to account for those variants. This violates the "*no modification on existing code*" principle of the Expression Problem. Modification is even impossible if the source code is unavailable. As a result, NAT and BOOL cannot be separated from ARITH. Thus, the whole implementation is neither extensible nor composable. Nevertheless, the exhaustiveness on visit methods is guaranteed since a class cannot contain any abstract methods.

2.4. Sealed Case Classes

The VISITOR pattern is often used as a poor man's approach to pattern matching in OO languages. Fortunately, Scala [31] is a language that unifies functional and OO paradigms and supports pattern matching natively via case classes/extractors [32]. Case classes can be either open or sealed. Sealed case classes are close to algebraic datatypes in functional languages, which have a fixed set of variants.

Representing the `Tm` hierarchy using sealed case classes looks like this:

```
sealed trait Tm
case object TmZero extends Tm
case class TmSucc(t: Tm) extends Tm
case class TmPred(t: Tm) extends Tm
case object TmTrue extends Tm
case object TmFalse extends Tm
case class TmIf(t1: Tm,t2: Tm,t3: Tm) extends Tm
case class TmIsZero(t: Tm) extends Tm
```

The differences are that `Tm` is a `sealed trait` and variants of `Tm` are additionally marked as `case`. Also, no-argument variants are Scala's singleton `object`s and fields of case classes are by default `val`.

The `case` keyword triggers the Scala compiler to automatically inject methods into a class, including a constructor method (`apply`) and an extractor method (`unapply`). The injected constructor method simplifies creating objects from case classes. For example, a successor application to zero can be constructed via `TmSucc(TmZero)`. Conversely, the injected extractor enables tearing down an object via pattern matching.

The numeric value checker can be defined by pattern matching on the term:

```
def nv(t: Tm): Boolean = t match {
  case TmZero => true
  case TmSucc(t1) => nv(t1)
  case _ => false
}
```

The term `t` is matched sequentially against a series of patterns (`case` clauses). For example, `TmSucc(TmZero)` will be handled by the second `case` clause of `nv`, which recursively invokes `nv` on its subterm `t1` (which is `TmZero`). Then, `TmTrue` will be matched by the first `case` clause with a `true` returned eventually. A *wildcard pattern* (_) is used in the last `case` clause for handling boring cases altogether.

The strength of pattern matching shines in encoding the small-step semantics:

```
def eval1(t: Tm): Tm = t match {
  case TmSucc(t1) => TmSucc(eval1(t1))
  case TmPred(TmZero) => TmZero          // PredZero
  case TmPred(TmSucc(t1)) if nv(t1) => t1 // PredSucc
  case TmPred(t1) => TmPred(eval1(t1))    // Pred
  case TmIf(TmTrue,t2,_) => t2
  case TmIf(TmFalse,_,t3) => t3
  case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
  case TmIsZero(TmZero) => TmTrue
  case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
  case TmIsZero(t1) => TmIsZero(eval1(t1))
  case _ => throw NoRuleApplies
}
```

With the help of pattern matching, the overall definition is a direct mapping from the formalization shown in Figure 1. There is a one-to-one correspondence between the rules and the case clauses. For example, PredSucc is concisely described by a *deep* pattern (`TmPred(TmSucc(t1))`) with a *guard* (`if nv(t1)`) and Pred is captured only once by `TmPred(t1)`.

Client Code. The client code is also more natural and compact than that in visitors:

```
// iszero (if false then true else pred (succ 0))
val tm = TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
val tm1 = eval1(tm)  // iszero (pred (succ 0))
val tm2 = eval1(tm1) // iszero 0
val tm3 = eval1(tm2) // 0
```

where `new` clauses are no longer needed.

Discussion of the Approach. The Arith implementation using sealed case classes is very concise. Moreover, sealed case classes facilitate exhaustiveness checking on patterns since all variants are statically known. If we forgot to write the wildcard pattern in `nv`, the Scala compiler would warn us that a `case` clause for `TmPred` is missing. An exception is `eval1`, whose exhaustiveness is not checked by the compiler due to the use of guards. The reason is that a guard might call some function whose execution result is only known at runtime, making the reachability of that pattern difficult to decide statically. The price to pay for exhaustiveness is the inability to add new variants of `Tm` in separate files. Thus, like the visitor version, the implementation is neither extensible nor composable.

2.5. Open Case Classes

While the implementation using sealed case classes is concise, it is not modular because Arith is still defined as a whole. To separate out Nat and Bool, we turn to open case classes by trading exhaustiveness checking for the ability to add new variants in separate files. To make up for the loss of exhaustiveness, Zenger and Odersky's idea of *Extensible Algebraic Datatypes with Defaults* (EADDs) [33] can be applied. The key idea is to always use a default in each operation to handle variants that are not explicitly mentioned. The existence of a default makes operations extensible, as variants added later will be automatically subsumed by that default. If the extended variants have behavior different from the default, we can define a new operation that deals with the extended variants and delegates to the old operation.

We first remove the `sealed` constraint on `Tm` and specify the default behavior of `eval1` inside a trait `Term`:

```
trait Term {
  trait Tm
  def eval1(t: Tm): Tm = throw NoRuleApplies
}
```

Then, Nat can be defined as an extended trait for `Term`:

```
trait Nat extends Term {
  case object TmZero extends Tm
  case class TmSucc(t: Tm) extends Tm
  case class TmPred(t: Tm) extends Tm
  def nv(t: Tm): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  override def eval1(t: Tm): Tm = t match {
    case TmSucc(t1) => TmSucc(eval1(t1))
    case TmPred(TmZero) => TmZero           // PredZero
    case TmPred(TmSucc(t1)) if nv(t1) => t1 // PredSucc
    case TmPred(t1) => TmPred(eval1(t1))    // Pred
    case _ => super.eval1(t)
  }
}
```

`Nat` introduces `TmZero`, `TmSucc` and `TmPred` as variants of `Tm`. `nv` is defined in the old way. `eval1` is overridden with case clauses for `TmSucc` and `TmPred`, and `TmZero` is dealt by `Term`'s `eval1` via a `super` call.

Similarly, Bool is defined as another trait that extends `Tm` with its own variants and `eval1`:

```
trait Bool extends Tm {
  case object TmTrue extends Tm
  case object TmFalse extends Tm
  case class TmIf(t1: Tm,t2: Tm,t3: Tm) extends Tm
  override def eval1(t: Tm): Tm = t match {
    case TmIf(TmTrue,t2,_) => t2
    case TmIf(TmFalse,_,t3) => t3
    case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
    case _ => super.eval1(t)
  }
}
```

Finally, Arith can be defined as a unification of Nat and Bool implementations:

```
trait Arith extends Nat with Bool {
  case class TmIsZero(t: Tm) extends Tm
  override def eval1(t: Tm) = t match {
    case TmIsZero(TmZero) => TmTrue
```

```
    case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
    case TmIsZero(t1) => TmIsZero(eval1(t1))
    case TmZero => super[Nat].eval1(t)
    case _: TmSucc => super[Nat].eval1(t)
    case _: TmPred => super[Nat].eval1(t)
    case _ => super[Bool].eval1(t)
  }
}
```

Scala's mixin composition allows `Arith` to extend both `Nat` and `Bool`. The definition `nv` inherited from `Nat` works well in `Arith`, as it happens to have a very good default that automatically fits for the new cases. For instance, calling `nv(TmFalse)` returns `false` as expected. However, overridding `eval1` becomes problematic. We cannot simply complement the cases for `TmIsZero` and handle all the inherited cases at once. Instead we have to separate the inherited cases using *typecases* and delegate appropriately to either `Nat` or `Bool` via `super` calls.

Discussion of the Approach. Combining open case classes with EADDs brings extensibility. This idea works well for *linear* extensions (such as Nat and Bool) but not so well for *non-linear* extensions like Arith. As shown by `eval1` in `Arith`, composing non-linear extensions is tedious and error-prone. Without any assistance from the Scala compiler during this process, it is rather easy to make mistakes like forgetting to delegate a case or delegating a case to a wrong parent. Moreover, the exhaustiveness checking on case clauses is lost. Although in the spirit of EADDs case clauses should always end with a wildcard that ensures exhaustiveness, it is not enforced by the Scala compiler.

2.6. Partial Functions

To ease the composition of `Nat` and `Bool`, one may turn to Scala's `PartialFunction`. `PartialFunction` provides an `orElse` method for composing partial functions. `orElse` tries the composed partial functions sequentially until no `MatchError` is raised.

The open case class version of Arith can be adapted to a partial function version with a few changes. First, `eval1` in `Term` should be declared as a partial function:

```
def eval1: PartialFunction[Tm,Tm]
```

Second, wildcards cannot be used in implementing `eval1` anymore because they will shadow other partial functions to be composed. For example, `eval1` in `Bool` is rewritten as:

```
override def eval1 {
  case TmIf(TmTrue,t2,_) => t2
  case TmIf(TmFalse,_,t3) => t3
  case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
  case TmTrue => throw NoRuleApplies
  case TmFalse => throw NoRuleApplies
}
```

`PartialFunction[Tm,Tm]` is constructed using the anonymous function syntax with the argument `Tm` being directly pattern matched. The wildcard pattern is replaced by two named patterns `TmTrue` and `TmFalse` with identical right hand side, losing some convenience. Nevertheless, partial functions make the composition work more smoothly, avoiding the problems caused by the open case classes approach:

```scala
override def eval1 = super[Nat].eval1 orElse super[Bool].eval1 orElse {
  case TmIsZero(TmZero) => TmTrue
  case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
  case TmIsZero(t1) => TmIsZero(eval1(t1))
}
```

`eval1` is overridden by chaining `eval1` from `Nat` and `Bool` as well as a new partial function for the zero test using the `orElse` combinator.

Discussion of the Approach. Although combining open case classes with partial functions makes the composition smoother, it is still not fully satisfactory. The `orElse` combinator is left-biased, thus the *composition order determines the composed semantics*. That is, `f orElse g` is not equivalent to `g orElse f`, if `f` and `g` are two overlapped partial functions (i.e. both `f` and `g` define *same* `case` patterns). When composing such overlapped partial functions, `orElse` gives no warning. Also, the semantics of the overlapped patterns are all from either `f` or `g`, depending on which comes first. It is not possible to have a mixed semantics for overlapped patterns (e.g. picking `case A` from `f` and `case B` from `g` when both `f` and `g` define `case A` and `case B`), which restricts the reusability of partial functions. Lastly, partial functions is based on exception handling, which has a negative impact on performance.

2.7. Extensible Visitors

Essentially what makes pattern matching hard to be extended or composed is the *order-sensitive* semantics of pattern matching and wildcard patterns that cover both known and unknown variants. We think it is useful to distinguish between top-level (shallow) patterns and nested (deep) patterns. Top-level patterns should be order-insensitive and partitioned into multiple definitions so that they can be easily composed. We can achieve this by combining open case classes with extensible visitors [34, 10, 35, 21].

The Arith implementation is organized in a way similar to the open case classes approach. Let us start with `Term`:

```scala
trait Term {
  type TmV <: TmVisit
  trait Tm { def accept(v: TmV): v.OTm }
  trait TmVisit { _: TmV =>
    type OTm
    def apply(t: Tm) = t.accept(this)
  }
  trait TmDefault extends TmVisit { _: TmV =>
    def tm: Tm => OTm
  }
  trait Eval1 extends TmDefault { _: TmV =>
    type OTm = Tm
    def tm = _ => throw NoRuleApplies
  }
  val eval1: Eval1
}
```

Instead of using `TmVisit` in declaring the `accept` method, we use an *abstract type member* `TmV` and constrain it to be a *subtype* of `TmVisit`. This enables invocations on the methods declared inside `TmVisit`, but at the same time, decouples `Tm` from

`TmVisit`. The upper bound of the return type of the visit methods is also captured by an abstract type rather than a type parameter for avoiding reinstantiation in inherited visitors. Accordingly, the return type of `accept` is now a path dependent type `v.OTm`. A syntactic sugar method `apply` is defined inside `TmVisit` for enabling `v(x)` as a shorthand of `x.accept(v)`, where `x` and `v` are instances of `Tm` and `TmVisit`, respectively. To pass `this` as an argument of `accept` in implementing `apply`, we state that `TmVisit` is of type `TmV` using a *self-type annotation*. `TmDefault` is the default visitor interface [36], which extends `TmVisit` with a generic `tm` method for specifying the default behavior. To mimic wildcards, we use default visitors. But unlike wildcards, default visitors only deal with *known* variants. The method `tm` declared in `TmDefault` is where to specify the default behaviour for all terms. `Eval1` is a default visitor thus it extends `TmDefault`, specifies the output type `OTm` as `Tm` and implements `tm`. Each concrete visitor has a companion `val` declaration for allowing themselves to be used in other visitors.

The encoding makes more sense with the implementation of `Nat` given:

```
trait Nat extends Term {
  type TmV <: TmVisit
  case object TmZero extends Tm {
    def accept(v: TmV): v.OTm = v.tmZero
  }
  case class TmSucc(t: Tm) extends Tm {
    def accept(v: TmV): v.OTm = v.tmSucc(t)
  }
  case class TmPred(t: Tm) extends Tm {
    def accept(v: TmV): v.OTm = v.tmPred(t)
  }
  trait TmVisit extends super.TmVisit { _: TmV =>
    def tmZero: OTm
    def tmSucc: TmSucc => OTm
    def tmPred: TmPred => OTm
  }
  trait TmDefault extends TmVisit with super.TmDefault { _: TmV =>
    def tmZero = tm(TmZero)
    def tmSucc = tm
    def tmPred = tm
  }
  def nv(t: Tm): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  trait Eval1 extends TmDefault with super.Eval1 { _: TmV =>
    override def tmSucc = x => TmSucc(this(x.t))
    override def tmPred = {
      case TmPred(TmZero) => TmZero
      case TmPred(TmSucc(t)) if nv(t) => t
      case TmPred(t) => TmPred(this(t))
    }
  }
}
```

`Tm` is extended with several case classes/objects. Correspondingly `TmVisit` is extended with new visit methods and `TmV` is *covariantly refined* as the subtype of the extended

`TmVisit`. Visit methods are declared using Scala's functions instead of ordinary methods for two reasons. First, the argument type (`TmSucc`) has already been revealed by the method name (`tmSucc`) and can be inferred by the Scala compiler without losing information. Second, first-class functions facilitate pattern matching on the argument. These two advantages result in a concise definition of `Eval1`, where the type of `x` is omitted and a value of `TmPred => Tm` is constructed by pattern matching. Unlike conventional visitors, nested case analysis is much simplified via (nested) pattern matching rather than auxiliary visitors. For example, when a predecessor term is processed by `Eval1`, it will be recognized and dispatched to the `tmPred` method. Then the `TmPred` object is matched by the `case` clauses. As these are `case` clauses, deep patterns and guards can be used. To restore the convenience of wildcards for top-level patterns, `TmDefault` is used, which implements visit methods by delegating to `tm`. Notice that `Eval1` is defined as a trait instead of a class for enabling mixin composition. By extending both `TmDefault` and `super.Eval1`, `Eval1` only needs to override interesting cases.

The numeric value checker is defined as a method rather than a visitor. This is because, as we have discussed, *nv* is a good candidate for applying EADDs. Of course, `nv` can be defined as a default visitor like `Eval1`. But whenever `Nat` is extended with new terms, the definition of *nv* has to be refined, although this can be done by composing `Nv` with the newly generated `TmDefault` in one line.

Bool is defined in a similar manner:

```
trait Bool extends Term {
  type TmV <: TmVisit
  trait TmVisit extends super.TmVisit { _: TmV =>
    def tmTrue: OTm
    def tmFalse: OTm
    def tmIf: TmIf => OTm
  }
  trait TmDefault extends TmVisit with super.TmDefault { _: TmV =>
    def tmTrue = tm(TmTrue)
    def tmFalse = tm(TmFalse)
    def tmIf = tm
  }
  case object TmTrue extends Tm {
    override def accept(v: TmV) = v.tmTrue
  }
  case object TmFalse extends Tm {
    override def accept(v: TmV) = v.tmFalse
  }
  case class TmIf(t1: Tm, t2: Tm, t3: Tm) extends Tm {
    override def accept(v: TmV) = v.tmIf(this)
  }
  trait Eval1 extends TmDefault with super.Eval1 { _: TmV =>
    override def tmIf = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1), t2, t3)
    }
  }
}
```

With case clauses partitioned into visit methods according to their top-level pattern, unifying `Nat` and `Bool` becomes easy via Scala's mixin composition:

```
trait Arith extends Nat with Bool {
  type TmV <: TmVisit
  case class TmIsZero(t: Tm) extends Tm {
    override def accept(v: TmV) = v.tmIsZero(this)
  }
  trait TmVisit extends super[Nat].TmVisit
              with super[Bool].TmVisit { _: TmV =>
    def tmIsZero: TmIsZero => OTm
  }
  trait TmDefault extends TmVisit with super[Nat].TmDefault
                with super[Bool].TmDefault { _: TmV =>
    def tmIsZero = tm
  }
  trait Eval1 extends TmVisit with super[Nat].Eval1
            with super[Bool].Eval1 { _: TmV =>
    def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
  }
}
```

Defining `Eval1` for `Arith` only needs to inherit `Eval1` definitions from `Nat` and `Bool` and complement the `tmIsZero` method. Since `tmIsZero` is an interesting case, `Eval1` extends `TmVisit` rather than `TmDefault`.

Instantiation. Components defined in this way cannot be directly used in client code. An additional step to instantiate traits into objects is required. Instantiating `Arith`, for example, is done like this:

```
object Arith extends Arith {
  type TmV = TmVisit
  object eval1 extends Eval1
}
```

The companion object `Arith` binds the abstract type `TmV` to its corresponding the visitor interface `TmVisit`. The `eval1` declaration is met by a singleton object that extends `Eval1`. If `Eval1` does not implement all the visit methods, the object creation fails, with the missing methods reported.

Client Code. Now we can use `Arith` in client code through `import Arith._`. By importing `Arith`, the constructors and visitors defined inside `Arith` are right in scope. With the syntactic sugar defined for visitors, a term can be constructed and evaluated identically to the case class version.

Discussion of the Approach. With the powerful extensible visitor encoding, the Arith implementation is made both extensible and composable. However, extensible visitors are even more verbose than conventional ones. The use of traits in implementing visitors brings composability but, at the same time, requires extra instantiation code. Another downside of using traits is that the exhaustiveness checking on visit methods is deferred to the instantiation stage. Moreover, the encoding relies on advanced features of Scala, making it less accessible to novice Scala programmers.

2.8. EVF

Programming with visitors can be greatly simplified with the associated infrastructure automatically generated. This idea has been adopted in our previous work on EVF [21], which employs Java annotation processors for generating extensible visitor infrastructure.

EVF uses Object Algebra interfaces [9] to describe the abstract syntax:

```
@Visitor interface TmAlg<Tm> {
  Tm TmZero();
  Tm TmSucc(Tm t);
  Tm TmPred(Tm t);
}
```

where the type parameter `Tm` represents the datatype and capitalized methods that return `Tm` represent variants of `Tm`. Annotated as `@Visitor`, `TmAlg` will be recognized and processed by EVF. Then the infrastructure for `TmAlg` will be generated, including a class hierarchy, a visitor interface and various default visitors. Based on the generated visitor infrastructure, we are able to define `Nv`:

```
interface Nv<Tm> extends TmAlgDefault<Tm,Boolean> {
  @Override default Zero<Boolean> m() {
    return () -> false;
  }
  default Boolean TmZero() {
    return true;
  }
  default Boolean TmSucc(Tm t) {
    return visitTm(t);
  }
}
```

`Nv` is defined as an interface with visit methods implemented using default methods for retaining composability. The Java extensible visitor encoding adopted by EVF is, however, not as powerful as the Scala one shown in Section 2.7, which does not support modular ASTs. Whenever an annotated Object Algebra interface gets extended, a new hierarchy has to be generated, including the classes for the inherited variants. Thus, we cannot refer to a concrete datatype directly in visitors since this will make them inextensible. Instead, datatypes are kept abstract in visitors. To traverse an abstract datatype like `Tm`, `visitTm` is called. `visitTm` is a method exposed by the generated visitor interface, similar to `apply` shown in Section 2.7. `TmAlgDefault` is the default visitor similar to `TmDefault`, where the default behaviour is specified inside `m()`.

Defining `Eval1` is tricker:

```
interface Eval1<Tm> extends TmAlgDefault<Tm,Tm>, tm.Eval1<Tm> {
  TmAlgMatcher<Tm,Tm> matcher(); // Dependency declaration
  TmAlg<Tm> f();                 // Dependency declaration
  Nv<Tm> nv();                   // Dependency declaration
  @default Tm TmPred(Tm t) {
    return matcher()
        .TmZero(() -> t)
        .TmSucc(t1 -> nv().visitTm(t1) ? t1 : TmPred(visitTm(t)))
        .otherwise(() -> f().TmPred(visitTm(t)))
        .visitTm(t);
  }
  default Tm TmSucc(Tm t) {
```

```
      return f().TmSucc(visitTm(t));
    }
  }
```

There are three dependencies declared using abstract methods. Firstly, since Java does not support native pattern matching, the `matcher` dependency is convenient for constructing anonymous visitors. `matcher` returns an instance of the generated `TmAlgMatcher` interface, which provides fluent setters for defining visit methods via Java 8's lambdas. The `otherwise` setter mimics the wildcard pattern. Secondly, the reconstruction of a term is done via an abstract factory `f` of type `TmAlg<Tm>`. Lastly, the abstract method `nv` expresses the dependency on the visitor `Nv`.

Bool is implemented similarly in another package `bool`, whose definition is omitted. The implementation of Arith is more interesting, which is shown below:

```
@Visitor interface TmAlg<Tm> extends nat.TmAlg<Tm>, bool.TmAlg<Tm> {
  Tm TmIsZero(Tm t);
}
interface Eval1<Tm> extends GTmAlg<Tm,Tm>,bool.Eval1<Tm>,nat.Eval1<Tm> {
  TmAlgMatcher<Tm,Tm> matcher(); // Dependency refinement
  TmAlg<Tm> f();                 // Dependency refinement
  default Tm TmIsZero(Tm t) {
    return matcher()
      .TmZero(() -> f().TmTrue())
      .TmSucc(t1 -> nv(t1) ? f().TmFalse() : f().TmIsZero(visitTm(t)))
      .otherwise(() -> f().TmIsZero(visitTm(t)))
      .visitTm(t);
  }
}
interface Nv<Tm> extends TmAlgDefault<Tm,Boolean>, nat.Nv<Tm> {}
```

Nat and Bool implementations are merged using Java 8' multiple interface inheritance. Despite complementing `TmIsZero`, return types of dependencies are covariantly refined for allowing `TmIsZero` calls. Since `Nv` is implemented as a visitor, it needs to be refined as well.

Instantiation. Instantiating interfaces into classes for creating objects is also required:

```
static class NvImpl implements Nv<CTm>, TmAlgVisitor<Boolean> {}
static class Eval1Impl implements Eval1<CTm>, TmAlgVisitor<CTm> {
  public TmAlg<CTm> f() { return f; }
  public TmAlgMatcher<CTm,CTm> matcher() {
    return new TmAlgMatcherImpl<>();
  }
  public Nv<CTm> nv() { return nv; }
}
static TmAlgFactory f = new TmAlgFactory();
static NvImpl nv = new NvImpl();
static Eval1Impl eval1 = new Eval1Impl();
```

The interfaces are instantiated into classes with a suffix `Impl`. `Eval1Impl`, for example, implements `Eval1` by: 1) instantiating `Tm` as the generated datatype `CTm`; 2) inheriting the generated `TmAlgVisitor` for a `visitTm` implementation; 3) fullfilling the dependencies using `TmAlgFactory`, `TmAlgMatcherImpl` and `NvImpl` respectively.

Client Code. The term is constructed via the factory object `f` and can be evaluated like this:

```
CTm tm = f.TmIsZero(
  f.TmIf(f.TmFalse(),f.TmTrue(),f.TmPred(f.TmSucc(f.TmZero()))));
eval1.visitTm(eval1.visitTm(eval1.visitTm(tm)))
```

Discussion of the Approach. EVF simiplifies programming with visitors through code generation. It further addresses the extensibility issue by adopting extensible visitors. Restricted by Java, nested case analysis in EVF is done by means of anonymous visitors, which is not as expressive and concise as pattern matching in Scala. To enable composability, EVF visitors are defined using Java 8's interfaces with default methods—in the same spirit of using traits in Scala. Consequently, the exhaustiveness checking on the top-level visit methods is lost in visitor definition site and is delayed to the visitor instantiation site. Nevertheless, the exhaustiveness on the visit methods of the anonymous visitors is guaranteed because the `otherwise` setter must be called when constructing an anonymous visitor.

2.9. Castor

Highly inspired by EVF, Castor is a Scala framework designed for programming with generative, extensible visitors. Castor improves on EVF in two aspects. First, Castor adopts a more powerful Scala extensible visitor encoding presented in Section 2.7 that additionally enables pattern matching, GADTs, hierarchical datatypes, graphs, etc. Second, Castor employs Scalameta for annotation processing, which allows not only generating new code based on the annotated code but also modifying the annotated code itself. These extra abilities together result in more concise and expressive visitor code than that in EVF. We next give a modular implementation of Arith using Castor, which has a one-to-one correspondence with the code shown in Section 2.7.

Let us start with the root component `Term`:

```
@family trait Term {
  @adt trait Tm
  @default(Tm) trait Eval1 {
    type OTm = Tm
    def tm = _ => throw NoRuleApplies
  }
}
```

Several Castor's annotations are employed: `@family` denotes a Castor's component; `@adt` denotes a datatype; `@default(Tm)` denotes a default visitor on `Tm`. Compared to the `Term` definition shown in Section 2.7, the definition here is much simplified. The `accept` declaration, the type member `TmV`, the visitor interface `TmVisit` and the default visitor `TmDefault` are all generated by analyzing the `@adt` definition of `Tm`. Similarly, Castor adds the extends clause, the self type annotation and the corresponding `val` declaration for `Eval1` by the annotation `@default(Tm)`.

Defining `Nat` is also much simplified:

```
@family trait Nat extends Term {
  @adt trait Tm extends super.Tm {
    case object TmZero
    case class TmSucc(t: Tm)
    case class TmPred(t: Tm)
  }
  def nv(t: Tm): Boolean = t match {
```

```
      case TmZero => true
      case TmSucc(t1) => nv(t1)
      case _ => false
    }
    @default(Tm) trait Eval1 extends super.Eval1 {
      override def tmSucc = x => TmSucc(this(x.t))
      override def tmPred = {
        case TmPred(TmZero) => TmZero
        case TmPred(TmSucc(t)) if nv(t) => t
        case TmPred(t) => TmPred(this(t))
      }
    }
  }
```

Variants of `Tm` are declared inside `Tm`. Castor will pull them outside of `Tm` and automatically complement the `extends` clause and the `accept` method definition. Since new variants of `Tm` are introduced, Castor will add the extended `TmVisit`, `TmDefault` and refined `TmV` to `Nat`.

Similarly, Bool can be defined as follows:

```
@family trait Bool extends Term {
  @adt trait Tm extends super.Tm {
    case object TmTrue
    case object TmFalse
    case class TmIf(t1: Tm, t2: Tm, t3: Tm)
  }
  @default(Tm) trait Eval1 extends super.Eval1 {
    override def tmIf = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1),t2,t3)
    }
  }
}
```

The code below finishes the Arith implementation:

```
@family trait Arith extends Nat with Bool {
  @adt trait Tm extends super[Nat].Tm with super[Bool].Tm {
    case class TmIsZero(t: Tm)
  }
  @visit(Tm) trait Eval1 extends super[Nat].Eval1
                          with super[Bool].Eval1 {
    def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
  }
}
```

Since the `TmIsZero` is an interesting case for `Eval1`, `@visit` annotation is used. Thus, `Eval1` extends `TmVisit` in the generated code.

Client Code. A `@family` trait can be directly imported in client code since Castor automatically generates a companion object for it. As a result, `Arith` can be used in the exact way as shown in Section 2.7.

Table 1: Pattern matching support comparison: ●= good, ◐= neutral, ○= bad.

	Conciseness	Exhaustiveness	Extensibility	Composability
Conventional visitors	○	●	○	○
Sealed case classes	●	●	○	○
Open case classes	●	○	●	○
Partial functions	●	○	●	◐
Extensible visitors	○	○	●	●
EVF	◐	◐	●	●
Castor	●	◐*	●	●

* Castor only gets half score on exhaustiveness because *for nested case analysis Scala cannot enforce a default*. In a language-based approach nested case analysis should always require a default, thus fully supporting exhaustiveness.

Discussion of the Approach. We discuss how Castor addresses the four properties:

- **Conciseness.** By employing Scala's concise syntax and metaprogramming, Castor greatly simplifies the definition and usage of visitors. In particular, the need for auxiliary visitors in performing deep case analysis is now replaced by pattern matching via `case` clauses. The concept of visitors is even made transparent to the end-user, making the framework more user-friendly.

- **Exhaustiveness.** The exhaustiveness of patterns in Castor consists of two parts. The exhaustiveness of visit methods is checked by the Scala compiler when generating companion objects. For nested patterns using `case` clauses, a default must be provided. However, this default is neither statically enforced by Scala nor Castor. Note, however, that with specialized language support it is possible to enforce that nested patterns always provide a default. This is precisely what EADDs [33] do.

- **Extensibility.** As illustrated by `Nat`, `Bool` and `Arith`, we can extend the datatype with new variants and operations, modularly. Such extensibility is enabled by the underlying extensible visitor encoding.

- **Composability.** Castor obtains composability via Scala's mixin composition, as illustrated by `Arith`. Unlike partial functions, which silently compose overlapped patterns, composing overlapped patterns in Castor will trigger compilation errors because they are conflicting methods from different traits. The error message will indicate the source of conflicts and we are free to select an implementation in resolving the conflict. The composition order does not matter as well.

Table 1 summarizes the evaluation on pattern matching approaches in terms of conciseness, exhaustiveness, extensibility, and composability. Castor is compared favorably in terms of the four properties among the approaches.

3. Hierarchical Datatypes

Traditional functional style datatypes are *flat*: variants have no relationships among each other. In contrast, object-oriented style datatypes (i.e. data structures modeled as class hierarchies) can be *hierarchical*: a variant can extend intermediate datatypes and/or an existing variant. In other words, while OO style class hierarchies can be arbitrarily deep, typical functional datatypes would correspond to a hierarchy where the depth is always one.

Hierarchical datatypes facilitate reuse. The subtyping relationship allows the semantics defined for supertypes to be reused in subtypes. Castor supports both styles of datatypes. In this section, we illustrate Castor's support for hierarchical datatypes by revising the Arith language. Another form of hierarchical datatypes will be shown in Section 5, where a new variant is introduced by refining an existing variant. Moreover, the case study on UML Activity Diagrams Section 8 further illustrates the use of hierarchical datatypes.

3.1. Flat Datatypes versus Hierarchical Datatypes

Terms of the Arith language shown in Section 2 are represented as a flat datatype, where all the variants extend the root datatype `Tm`. In fact, terms can be organized in a hierarchical manner according to their types and arities. Figure 2 visualizes the hierarchical representation of terms and the following code materializes it using Castor:

```
@adt trait Tm {
  trait TmNullary
  trait TmUnary { val t: Tm }
  trait TmTernary { val t1, t2, t3: Tm }
  trait TmNat extends TmNullary
  trait TmBool extends TmNullary
  trait TmNat2Nat extends TmUnary
  trait TmNat2Bool extends TmUnary
  case object TmZero extends TmNat
  case class TmSucc(t: Tm) extends TmNat2Nat
  case class TmPred(t: Tm) extends TmNat2Nat
  case object TmTrue extends TmBool
  case object TmFalse extends TmBool
  case class TmIf(t1: Tm, t2: Tm, t3: Tm) extends TmTernary
  case class TmIsZero(t: Tm) extends TmNat2Bool
}
```

Several intermediate datatypes are introduced, making the hierarchy multi-layered. Case classes/objects do not directly extend `Tm` but an intermediate datatype. Specifically, `TmNullary` and `TmUnary` and `TmTernary` classify terms according to their arities. `TmNat`, `TmBool`, `TmNat2Nat`, `TmNat2Bool` further classify terms according to their types. For example, `TmNat2Nat` is the common supertype for `TmSucc` and `TmPred`.

3.2. Explicit Delegations

Now we illustrate the advantages of hierarchical datatypes. Suppose we would like to define a printer for Arith that prints out a term using an S-expression like format. For example, `TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))` will be printed as `"(iszero (if false true (pred (succ 0))))"`. With terms being classified according to their arities, the printer can be modularized:

```
@visit(Tm) trait Print {
  type OTm = String
  def tmZero = "0"
```

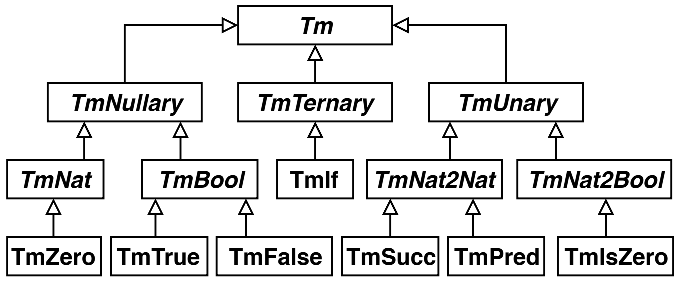

Figure 2: Hierarchical representation of Arith terms.

```
  def tmUnary(x: TmUnary, op: String) = "(" + op + " " + this(x.t) + ")"
  def tmSucc = tmUnary(_,"succ")
  def tmPred = tmUnary(_,"pred")
  def tmTrue = "true"
  def tmFalse = "false"
  def tmIf = x =>
    "(if " + this(x.t1) + " " + this(x.t2) + " " + this(x.t3) + ")"
  def tmIsZero = tmUnary(_,"iszero")
}
```

Since all unary terms (i.e. `TmSucc`, `TmPred` and `TmIsZero`) are printed in the same way except for the operator name, we define an auxilary method `tmUnary`. `tmUnary` prints out the operator and the inner term of `TmUnary` surrounded by parenthesis. Then, `tmSucc`, `tmPred` and `tmIsZero` are implemented just by calling `tmUnary` with their respective object and operator name.

3.3. Default Visitors

The previous example has shown how to enhance the modularity through explicit delegations. When subtypes share the same behavior with supertypes, the explicit delegations can be eliminated with the help of the generated default visitor. Currently, the Arith language presented allows ill-typed terms such as `TmPred(TmTrue)` to be constructed. To rule out these ill-typed terms, typechecking is needed. Some of the terms share typing rules: `TmTrue` and `TmFalse`; `TmSucc` and `TmPred`. With Castor's default visitor, we can avoid duplication of typing rules:

```
@adt trait Ty {
  case object TyNat
  case object TyBool
}
@default(Tm) trait Typeof {
  type OTm = Option[Ty]
  override def tmBool = _ => Some(TyBool)
  override def tmNat = _ => Some(TyNat)
  override def tmNat2Nat = x => this(x.t) match {
    case Some(TyNat) => Some(TyNat)
    case _ => None
  }
  override def tmNat2Bool = x => this(x.t) match {
    case Some(TyNat) => Some(TyBool)
```

```scala
      case _ => None
    }
    override def tmIf = x => (this(x.t1),this(x.t2),this(x.t3)) match {
      case (Some(TyBool),ty1,ty2) if ty1 == ty2 => this(x.t2)
      case _ => None
    }
    def tm = _ => None
  }
```

Like `Tm`, `Ty` is a datatype for representing types, where `TyNat` and `TyBool` are two concrete types. A visitor `Typeof` is defined for typechecking terms. The output type of `Typeof` is `Option[Ty]`, indicating that if a term is well-typed, some type will be returned; otherwise a `None` will be returned. Except for `TmIf`, typing rules are defined on intermediate datatypes. For example, `tmNat2Nat` is overridden, which checks whether its inner term is of type `TyNat` and returns `TyNat` if so. `tmSucc` and `tmPred` are implicitly implemented by the inherited default visitor, whose definition is given below:

```scala
trait TmDefault extends TmVisit { _: TmV =>
  def tm: Tm => OTm
  def tmNullary = (x: TmNullary) => tm(x)
  def tmUnary = (x: TmUnary) => tm(x)
  def tmTernary = (x: TmTernary) => tm(x)
  def tmNat = (x: TmNat) => tmNullary(x)
  def tmBool = (x: TmBool) => tmNullary(x)
  def tmNat2Nat = (x: TmNat2Nat) => tmUnary(x)
  def tmNat2Bool = (x: TmNat2Bool) => tmUnary(x)
  def tmZero = tmNat(TmZero)
  def tmSucc = tmNat2Nat(_)
  def tmPred = tmNat2Nat(_)
  def tmTrue = tmBool(TmTrue)
  def tmFalse = tmBool(TmFalse)
  def tmIf = tmTernary(_)
  def tmIsZero = tmNat2Bool(_)
}
```

We can see that the default visitor extends the visitor interface with visit methods for intermediate datatypes and each visit method is implemented by delegating to its direct parent's visit method.

4. GADTs and Well-Typed EDSLs

In this section, we show the support of *generalized algebraic data types* (GADTs) [37] in Castor. GADTs allow not only datatypes to be parameterized but also well-formedness constraints to be expressed in constructors. GADTs are widely used for building well-typed domain-specific languages (EDSLs), which exploit the type system of the host language to typecheck the terms of the EDSL. Popular approaches to EDSLs like Finally Tagless [11] can provide an encoding of GADTs and provide modularity as well. However, the encoding employed by Finally Tagless is based on Church encodings. Unfortunately, this makes it hard to model several operations that require nested patterns or operations with dependencies. The interested reader is referred to Section 2 and 3 of the EVF paper [21] for a detailed discussion on the issue of Church encodings. We show that just as Finally Tagless encodings, modularity is supported; and like GADTs nested pattern matching and dependencies are easy to do as well.

4.1. GADTs and Well-Typed Terms

We have shown how to rule out ill-typed terms using a type-checking algorithm in Section 3.3. A better solution, however, is to prevent such terms from being constructed in the first place. This is possible through representing ARITH terms using a GADT-style:

```
@family trait GArith {
  @adt trait Tm[A] {
    case object TmZero extends Tm[Int]
    case class TmSucc(t: Tm[Int]) extends Tm[Int]
    case class TmPred(t: Tm[Int]) extends Tm[Int]
    case object TmTrue extends Tm[Boolean]
    case object TmFalse extends Tm[Boolean]
    case class TmIf[A](t1: Tm[Boolean], t2: Tm[A], t3: Tm[A])
      extends Tm[A]
    case class TmIsZero(t: Tm[Int]) extends Tm[Boolean]
  }
}
```

`Tm` is now parameterized by a type parameter `A`. When declaring variants of `Tm`, the `extends` clause cannot be omitted since the type parameter `A` must be explicitly instantiated. Notice that `A` can be instantiated differently as `Int` or `Boolean` for expressing well-formedness constraints. For example, `TmIsZero` requires its subterm `t` of type `Tm[Int]`. Consequently, one cannot supply a term of type `Tm[Boolean]` constructed from `TmTrue`, `TmFalse` or `TmIsZero` to `TmIsZero`. Therefore, ill-formed terms are statically rejected by the Scala type system:

```
TmIsZero(TmZero) // Accepted!
TmIsZero(TmTrue) // Rejected!
```

4.2. Well-Typed Big-Step Evaluator

As opposed to the small-step semantics, big-step semantics immediately evaluates a valid term to a value. In the case of ARITH, a term can either be evaluated to an integer or a boolean value. Without GADTs, implementing a big-step evaluator for ARITH is tedious:

```
@family @adts(Tm) @ops(Eval1) trait EvalArith extends Arith {
  @adt trait Value {
    case class IntValue(v: Int)
    case class BoolValue(v: Boolean)
  }
  @visit(Tm) trait Eval {
    type OTm = Value
    def tmZero = IntValue(0)
    def tmSucc = x => this(x.t) match {
      case IntValue(n) => IntValue(n+1)
      case _ => throw NoRuleApplies
    }
    def tmPred = x => this(x.t) match {
      case IntValue(n) => IntValue(n-1)
      case _ => throw NoRuleApplies
    }
    def tmTrue = BoolValue(true)
    def tmFalse = BoolValue(false)
    def tmIf = x => this(x.t1) match {
```

```
      case BoolValue(true) => this(x.t2)
      case BoolValue(false) => this(x.t3)
      case _ => throw NoRuleApplies
    }
    def tmIsZero = x => this(x.t) match {
      case IntValue(0) => BoolValue(true)
      case IntValue(_) => BoolValue(false)
      case _ => throw NoRuleApplies
    }
  }
}
```

EvalArith illustrated the operation extensibility of Castor.

inherited datatypes and operations are provided by @adts and @ops for code generation. The implementation suffers from the tag problem [11]. To accommodate different evaluation result types, an open datatype Value is defined for accommodating integers, booleans and many other evaluation result types that might be added in the future. The two variants IntValue and BoolValue are introduced for wrapping integers and boolean values, respectively. Pattern matching is used for unwrapping the evaluation results from inner terms. A defensive wildcard is needed for dealing with ill-typed terms. We can see that the tagging overhead is high.

Fortunately, we can avoid the tag problem with the help of Castor's GADTs. The extensible visitor encoding for GADTs is slightly different from the one presented in Section 2.7, which additionally take the type information carried by terms into account. For instance, the visitor interface generated for Tm[A] is listed below:

```
trait TmVisit { _: TmV =>
  type OTm[A]
  def apply[A](x: Tm[A]) = x.accept(this)
  def tmZero: OTm[Int]
  def tmSucc: TmSucc => OTm[Int]
  def tmPred: TmPred => OTm[Int]
  def tmTrue: OTm[Boolean]
  def tmFalse: OTm[Boolean]
  def tmIf[A]: TmIf[A] => OTm[A]
  def tmIsZero: TmIsZero => OTm[Boolean]
}
```

Each visit method now returns a value of a higher-kinded type OTm[A], where A is instantiated consistently with how it is instantiated in the extends clause. For example, tmZero is of type OTm[Int] while tmTrue is of type OTm[Boolean]. Then, a well-typed big-step evaluator can be made *tagless*:

```
@family @adts(Tm) trait EvalGArith extends GArith {
  @visit(Tm) trait Eval {
    type OTm[A] = A
    def tmZero = 0
    def tmSucc = x => this(x.t) + 1
    def tmPred = x => this(x.t) - 1
    def tmTrue = true
    def tmFalse = false
    def tmIf[A] = x => if (this(x.t1)) this(x.t2) else this(x.t3)
    def tmIsZero = x => this(x.t) == 0
  }
}
```

With the output type specified as `A`, the visit method returns a value of the type carried by the term. For example, visit methods `tmZero` and `tmTrue` return `Int` and `Boolean` respectively. Moreover, this `Eval` implementation remains retroactive when terms of a new type (such as `Tm[Float]`) are introduced.

Here are some terms that evaluate to results of different types.

```
import EvalGArith._
eval(TmSucc(TmZero))   // 1
eval(TmIsZero(TmZero)) // true
```

4.3. Well-Typed Small-Step Evaluator

Well-typed big-step evaluators can be defined with Finally Tagless in an equally simple manner. What distinguishes Castor from Finally Tagless is the ability to define interpreters using a small-step semantics in an easy way. The need for deep patterns and the dependency on an operation for checking if a value is numeric causes immediate trouble for Finally Tagless. Although workarounds may be possible for some of the issues, they are cumbersome and require significant amounts of boilerplate code [23]. In contrast, writing small-step semantics in GADT-stype with Castor is unproblematic:

```
@family @adts(Tm) trait Eval1Arith extends GArith {
  def nv[A](t: Tm[A]): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  @default(Tm) trait Eval1 {
    type OTm[A] = Tm[A]
    def tm[A] = x => throw NoRuleApplies
    override def tmIf[A] = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1),t2,t3)
    }
    override def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
    ... // Other cases are the same as before
  }
}
```

The instantiation of the output type guarantees that the small-step evaluator is *type-preserving*. That is, the type carried by a term remains the same after one step of evaluation. For example, calling `eval1` on `TmZero` will never return `TmTrue` no matter how `Eval1` is implemented. The actual definition of `Eval1` is almost the same as before except that `nv`, `tm` and `tmIf` become generic. Still, the ability to do nested pattern matching and to call `nv` in `Eval1` is preserved.

4.4. An Extension: Higher-Order Abstract Syntax for Name Binding

A recurring problem in designing EDSLs is how to deal with binders. For example, in lambda calculus, operations involved with names like α-equivalence and capture-avoiding substitution are non-trivial to define. Higher-order abstract syntax (HOAS) [38] avoids these problems through reusing the binding mechanisms provided by the host language. The following code shows how to extend Arith with simply-typed lambda calculus modularly:

```
@family trait HOAS extends EvalGArith {
  @adt trait Tm[A] extends super.Tm[A] {
    case class TmVar[A](v: A) extends Tm[A]
    case class TmAbs[A,B](f: Tm[A] => Tm[B]) extends Tm[A => B]
    case class TmApp[A,B](t1: Tm[A => B], t2: Tm[A]) extends Tm[B]
  }
  @visit(Tm) trait Eval extends super.Eval {
    def tmVar[A] = _.v
    def tmAbs[A,B] = x => y => this(x.f(TmVar(y)))
    def tmApp[A,B] = x => this(x.t1)(this(x.t2))
  }
}
```

Three new forms of terms are introduced: lifters (`TmVar`), lambda abstractions (`TmAbs`) and applications (`TmApp`). Of particular interest is `TmAbs`, which constructs a term of type `Tm[A => B]` from a Scala lambda function `Tm[A] => Tm[B]` and thus is *higher-order*.

Correspondingly, `Eval` is extended with three new visit method implementations. `tmVar` simply extracts the value out of the lifter. `tmAbs` is trickier since it returns a value of type `A => B`. A lambda function is hence created, which takes `y` of type `A` and lifts it into `Tm[A]` using `TmVar`, then applies `x.f` to the lifted term for computing a `Tm[B]` and finally does a recursive call to evaluate `Tm[B]` into `B`. `tmApp` recursively evaluates the `t1` and `t2`, which returns the value `A => B` and `A` respectively. Then it applies `A => B` to `A` for getting a value of `B`.

Here is an example that illustrates the use of `HOAS`:

```
import HOAS._
eval(TmApp(TmAbs((t: Tm[Int]) => TmSucc(TmSucc(t))), TmZero)) // 2
```

We first create an abstraction term that applies successor twice to the argument `t` and then apply it to constant zero. Note that the type of `t` is explicitly specified because Scala's type system is not powerful enough to infer the type of `TmAbs` without annotations.

5. Graphs and Imperative Visitors

Examples presented so far are all *functional* visitors (i.e. computation is done via returning values) on immutable trees. In fact, Castor also supports *imperative* visitors (i.e. computation is done via side effects) and the data structure can be a mutable graph. Imperative computation is, in some cases, more efficient than the functional counterpart regarding time and memory. Compared to trees, graphs are a more general data structure that have many important applications. For instance, in the domain of compilers, abstract semantic graphs can be used for representing shared subexpressions, facilitating optimizations like common subexpression elimination. In this section, we show how to model graphs and imperative visitors with Castor.

5.1. The Difficulties in Modeling Graphs

Modeling graphs modularly is non-trivial in approaches such as Object Algebras. Consider modeling a Finite State Machine (FSM) language. Figure 3 shows a UML class

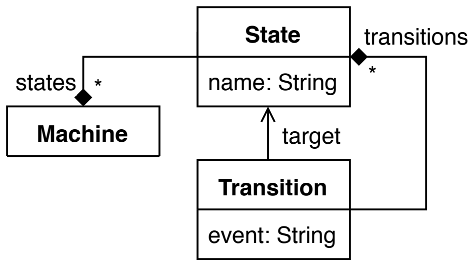

Figure 3: Class diagram of FSM.

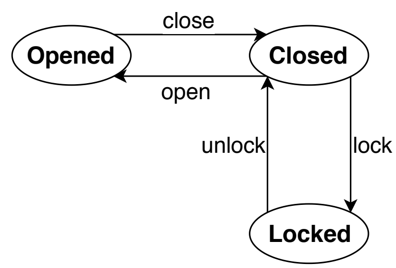

Figure 4: A state machine for controlling a door.

diagram for the FSM language. A `Machine` consists of some `States`. Each `State` has a name and a number of `Transitions`. A `Transition` is triggered by an `event`, taking one `State` to another. Concretely, Figure 4 shows a simple state machine for controlling a door, which has three states (opened, closed and locked) and four transitions (close, open, unlock and lock). From Figure 4 we can see that this state machine forms a *graph*, where we can go back and forth from one state to another along with the transitions.

A Failed Attempt with Object Algebras. Let us try to model the FSM language with Object Algebras [9]. Describing the FSM language using an Object Algebra interface is unproblematic:

```
trait FSM[M,S,T] {
  def machine(states: List[S]): M
  def state(name: String, trans: List[T]): S
  def trans(event: String, target: S): T
}
```

where `M`, `S`, `T` and their variants are captured as type parameters and factory methods respectively. However, constructing a graph using this representation is hard because Object Algebras support only immutable tree structures that are built *bottom up*. Here is a failed attempt on modeling the door state machine:

```
// Forward reference error!
def door[M,S,T](f: FSM[M,S,T]) = {
  val close: T = f.trans("close",closed)
  val open: T = f.trans("open",opened)
  val lock: T = f.trans("lock",locked)
  val unlock: T = f.trans("unlock",closed)
  val opened: S = f.state("opened", List(close))
  val closed: S = f.state("opened", List(open,lock))
  val locked: S = f.state("opened", List(unlock))
  f.machine(List(opened,closed,locked))
}
```

A *forward reference* error will always occur no matter how we arrange these statements. The reason is that there is no proper way to decouple the cyclic references between states and transitions.

5.2. *FSM in* Castor

Fortunately, modeling the FSM language using Castor is not a problem:

```
@family trait FSM {
  @adt trait M {
```

```
    val states = ListBuffer[S]()
    class Machine
  }
  @adt trait S {
    val trans = ListBuffer[T]()
    var name: String
    class State(var name: String)
  }
  @adt trait T {
    class Trans(val event: String, var target: S)
  }
  @visit(M,S,T) trait Print {
    type OM = String
    type OS = OM
    type OT = OM
    def machine = _.states.map{this(_)}.mkString("\n")
    def state = s => s.trans.map{this(_)}.mkString(s.name+":\n","\n","")
    def trans = t => t.event + " -> " + t.target.name
  }
  @visit(M,S,T) trait Step {
    type OM = String => Unit
    type OS = OM
    type OT = OM
    var res: S = null
    def machine = m => event => m.states.foreach{this(_)(event)}
    def state = s => event => s.trans.foreach{this(_)(event)}
    def trans = t => event => if (event == t.event) res = t.target
  }
}
```

The actual class hierarchies of the FSM language are slightly different from what Figure 3 shows. Each class in the UML diagram is defined inside an `@adt` trait for allowing potential variant extensions. Fields are either declared as `var` or `val` for enabling/disabling mutability.

Combined Visitors. There are two visitors defined for the FSM language, namely `Print` and `Step`. Both of them are *combined* visitors that apply to transitions, states, and machines. Such a combined implementation is much more compact than defining three mutually dependent visitors with distinct names. Annotated as `@visit(M,S,T)`, `Print` instantiates the output types `OM`, `OS`, `OT` consistently as `String` and implements three visit methods `machine`, `state` and `trans` altogether. Concretely, methods `machine` and `state` map `Print` to the substructures and concatenate the results with a newline. For `trans`, we should not call `this` on the `target` state otherwise it will not terminate. Instead, we print out the `name` field on the `target` state only.

Imperative Visitors. The `Step` visitor captures the small-step execution semantics of FSM. Given an event, it goes through the structure for finding out the transition triggered by that event and returning the state that transition points to. Note that `Step` is, at the same time, an *imperative* visitor. `Step` instantiates the output types as `String => Unit` and updates the field `res` to the found target transition. If `res` is still `null` after traversal, then no such transition exists.

Now we are able to model the state machine that controls doors like this:

```
import FSM._

val door = new Machine
val opened = new State("Opened")
val closed = new State("Closed")
val locked = new State("Locked")
val open = new Trans("open",opened)
val close = new Trans("close",closed)
val lock = new Trans("lock",locked)
val unlock = new Trans("unlock",closed)

door.states += (opened,closed,locked)
opened.trans += close
closed.trans += (open,lock)
locked.trans += unlock
```

The graph is constructed in a conventional OOP style. Unlike Object Algebras, the structure is built *top down*. To decouple cyclic references, the declaration and initialization of the variables are separated. This is possible in Castor because variants are concrete classes provided with setters whereas in Object Algebras they are abstract types without concrete representations.

Calling `print(door)` produces the following output:

```
Opened:
close -> Closed
Closed:
open -> Opened
lock -> Locked
Locked:
unlock -> Closed
```

Some tests on `Step` are:

```
step(door)("open")
println(step.res.name) // "Opened"
step.res = null        // Reset to null
step(door)("close")
println(step.res.name) // "Closed"
```

Imperative visitors should be used more carefully. In the case of `Step`, its field `res` needs to be reset to `null` afterwards. Otherwise, the result may be wrong next time we call `step`.

5.3. Language Composition and Memoized Traversals

Consider unifying FSM and Arith. The unification happens when a new kind of transition called guarded transitions is introduced. A guarded transition additionally contains a boolean term and is triggered not only by the event but also by the evaluation result of that term. Combining FSM with the GADT version of Arith is given below:

```
@family @adts(Tm,F,S) @ops(Eval)
trait GuardedFSM extends FSM with EvalArith {
  @adt trait T extends super[FSM].T {
    class GuardedTrans(event: String, target: State, val tm: Tm[Boolean])
      extends Trans(event, target)
  }
  @visit(M,S,T) trait Print extends super[FSM].Print {
    def guardedTrans = t => trans(t) + " when " + t.tm.toString
```

```
  }
  @visit(F,S,T) trait Step extends super[FSM].Step {
    def guardedTrans = t => event => if (eval(t.tm)) trans(t)(event)
  }

  @visit(S,T) trait Reachable {
    type OS = Unit
    type OT = Unit
    val reached = collection.mutable.Set[S]()
    def state = s =>
      if (!reached.contains(s)) {
        reached += s
        s.trans.foreach(this(_))
      }
    def trans = t => this(t.target)
    def guardedTrans = t => if (eval(t.tm)) this(t.target)
  }
}
```

Class `GuardedTrans` extends `Trans` with an additional field `tm` of type `Tm[Boolean]`. To handle `GuardedTrans`, `Print` and `Step` are extended with an implementation of `guardedTrans` method. Having `GuardedTrans` as a subtype of `Trans`, we are able to partially reuse the semantics of `Trans` for `GuardedTrans` via passing `t` to the inherited `trans` method.

Memoized Traversals. Naively traversing a graph might be inefficient because the same object may be traversed multiple times. In the worst case, the traversal may not even terminate if not dealt with carefully. A better approach is to memoize the results of traversed objects and fetch the cached result when an object is traversed again. `Reachable` is a combined imperative visitor that finds out all reachable states for the given state. The reachable states are collected in a `reached` field, which is initialized as an empty mutable set. `Reachable` employs memoized depth-first search, which first checks whether the state has already been traversed. If not, the state is added to `reached` and the recursion goes to the states its transitions lead to. Similarly, memoization can be applied to functional visitors by changing `reached` to a mutable map.

We can build a guarded door controller by changing the `import` statement and how `lock` is initialized:

```
val lock = new GuardedTrans("lock",locked,TmFalse)
```

Now, an opened door can no longer be locked because the guard evaluates to `false`:

```
reachable(open)
println(reachable.reached.size) // 2
```

By setting the expression to `TmTrue`, the door can be locked again:

```
lock.tm = TmTrue
reachable.clear // Reset to empty
reachable(open)
println(reachable.reached.size) // 3
```

6. Formalized Code Generation

Castor employs Scalameta [39], a modern Scala meta-programming library, for generating the boilerplate required by the extensible visitor encoding. In this section,

$$
\begin{array}{lcl}
\mathit{Fam} & ::= & \texttt{@family @adts(}\overline{D}\texttt{) @ops(}\overline{V}\texttt{) trait } F \texttt{ extends } \overline{F}\texttt{\{}\overline{\mathit{Adt}}\ \overline{\mathit{Vis}}\texttt{\}} \\
\mathit{Adt} & ::= & \texttt{@adt trait } D\texttt{[}\overline{X}\texttt{] extends } \overline{\texttt{super[}F\texttt{].}D\texttt{[}\overline{X}\texttt{]}}\texttt{\{}\overline{\mathit{Ctr}}\texttt{\}} \\
\mathit{Ctr} & ::= & \texttt{class } C\texttt{[}\overline{X}\texttt{] extends (}C\texttt{[}\overline{T}\texttt{] with)? } \overline{D\texttt{[}\overline{T}\texttt{]}} \\
 & | & \texttt{object } C \texttt{ extends (}C\texttt{[}\overline{T}\texttt{] with)? } \overline{D\texttt{[}\overline{T}\texttt{]}} \\
 & | & \texttt{trait } D\texttt{[}\overline{X}\texttt{] extends } \overline{D\texttt{[}\overline{T}\texttt{]}} \\
\mathit{Vis} & ::= & \texttt{@(default | visit)(}\overline{D}\texttt{) trait } V \texttt{ extends } \overline{\texttt{super[}F\texttt{].}V} \\
T & ::= & X \mid D\texttt{[}\overline{T}\texttt{]} \mid \texttt{Int} \mid T \texttt{=>} T
\end{array}
$$

Figure 5: Syntax.

we formally describe the valid Scala programs accepted by CASTOR and the translation scheme.

6.1. Syntax

Figure 5 describes valid Scala programs accepted by CASTOR. Uppercase metavariables range over capitalized names. $\overline{A}$ is written as a shorthand for a potentially empty sequence $A_1 \bullet \ldots \bullet A_n$, where $\bullet$ denotes `with`, comma or semicolon depending on the context. $(\ldots)?$ denotes that $\ldots$ is optional. For brevity, we ignore the syntax that is irrelevant in translation, such as the `case` modifier, constructors, fields, and methods. These parts are kept unchanged during translation.

6.2. Translation

Figure 6 formalizes the translation. We use semantic brackets ($[\![\cdot]\!]$) in defining the translation rules and angle brackets (<>) for processing sequences. The translation is given by pattern matching on the concrete syntax and is quite straightforward. One can see that processing the ARITH implementation in CASTOR (c.f. Section 2.9) through Figure 6 will get back the extensible visitor implementation (c.f. Section 2.7).

Here we briefly discuss some interesting cases. A `trait` is recognized as a *base case* if it extends nothing. Base cases have extra declarations such as `accept` declaration for datatypes or `val` declaration for visitors. Variants declared using `class`, `trait` or `object` are treated differently. `object`s and `class`es have their corresponding visit methods in the visitor interface while visit methods for `trait`s only exist in the default visitor. The `extends` clause for `@adt` is used in inferring the `extends` clause for concrete visitors.

6.3. Implementation

The actual implementation closely follows the formalization. After parsing, the Scala source program is represented as an AST. We then do pattern matching on the parsed AST for checking its validity. If the annotated program is not valid (e.g. annotating `@adt` not on a trait), the translation fails with errors reported. We next extract the necessary information from the valid AST for code generation. Finally, the transformed AST is typechecked by the Scala compiler. During the process, Scala's quasiquotes are used, which allow us to analyze and reconstruct the AST conveniently via the concrete syntax.

$$
\begin{array}{l}
[\![\texttt{@family @adts}(\overline{D})\ \texttt{@ops}(\overline{V})\ \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\,\texttt{\{}\,\overline{Adt}\ \overline{Vis}\texttt{\}}\,]\!] = \\
\quad \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\,\texttt{\{}\,[\![\overline{Adt}]\!]\ [\![\overline{Vis}]\!]\texttt{\}} \\
\quad \texttt{object}\ F\ \texttt{extends}\ F\,\texttt{\{} \\
\qquad \langle \texttt{type}\ D\texttt{V}\ \texttt{=}\ D\texttt{Visit} \mid D \in \overline{D} \cup \overline{Adt} \rangle \\
\qquad \langle \texttt{object}\ v\ \texttt{extends}\ V \mid V \in \overline{V} \cup \overline{Vis} \rangle \\
\quad \texttt{\}} \\
[\![\texttt{@adt trait}\ D\texttt{[}\overline{X}\texttt{]}\,\texttt{\{}\,\overline{Ctr}\texttt{\}}\,]\!] = \\
\quad \texttt{type}\ D\texttt{V}\ \texttt{<:}D\texttt{Visit} \\
\quad \texttt{trait}\ D\texttt{[}\overline{X}\texttt{]\{def accept(v:}D\texttt{V): v.O}D\texttt{[}\overline{X}\texttt{]\}} \\
\quad [\![\overline{Ctr}]\!] \\
\quad \texttt{trait}\ D\texttt{Visit\{ \_:}D\texttt{V =>} \\
\qquad \texttt{type O}D\texttt{[}\overline{X}\texttt{]} \\
\qquad \texttt{def apply[}\overline{X}\texttt{](x:}D\texttt{[}\overline{X}\texttt{]) = x.accept(this)} \\
\qquad [\![\overline{Ctr}]\!]_{visit} \\
\quad \texttt{\}} \\
\quad \texttt{trait}\ D\texttt{Default extends}\ D\texttt{Visit\{ \_:}D\texttt{V =>} \\
\qquad \texttt{def}\ d\texttt{[}\overline{X}\texttt{]:}D\texttt{[}\overline{X}\texttt{] => O}D\texttt{[}\overline{X}\texttt{]} \\
\qquad [\![\overline{Ctr}]\!]_{default} \\
\quad \texttt{\}} \\
[\![\texttt{@adt trait}\ D\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}D}\,\texttt{\{}\,\overline{Ctr}\texttt{\}}\,]\!] = \\
\quad \texttt{type}\ D\texttt{V}\ \texttt{<:}D\texttt{Visit} \\
\quad [\![\overline{Ctr}]\!] \\
\quad \texttt{trait}\ D\texttt{Visit extends}\ \overline{\texttt{super[}F\texttt{].}D\texttt{Visit}}\texttt{\{ \_:}D\texttt{V =>}[\![\overline{Ctr}]\!]_{visit}\texttt{\}} \\
\quad \texttt{trait}\ D\texttt{Default extends}\ D\texttt{Visit with}\ \overline{\texttt{super[}F\texttt{].}D\texttt{Default}}\texttt{\{\_:}D\texttt{V =>}\ [\![\overline{Ctr}]\!]_{default}\texttt{\}} \\
[\![\texttt{class}\ C\texttt{[}\overline{X}\texttt{]}\ \ldots]\!] = \texttt{class}\ C\texttt{[}\overline{X}\texttt{]}\ \ldots\texttt{\{override def accept(v:}D\texttt{V) = v.}c\texttt{(this)\}} \\
[\![\texttt{object}\ C\ \ldots]\!] = \texttt{object}\ C\ \ldots\texttt{\{override def accept(v:}D\texttt{V) = v.}c\texttt{\}} \\
[\![Ctr]\!] = Ctr \\
[\![\texttt{class}\ C\texttt{[}\overline{X}\texttt{] extends}\ (\ldots\texttt{with})?\ D\texttt{[}\overline{T}\texttt{]}\,]\!]_{visit} = \texttt{def}\ c\texttt{[}\overline{X}\texttt{]:}C\ \texttt{=> O}D\texttt{[}\overline{T}\texttt{]} \\
[\![\texttt{object}\ C\ \texttt{extends}\ (\ldots\ \texttt{with})?\ D\texttt{[}\overline{T}\texttt{]}\,]\!]_{visit} = \texttt{def}\ c\texttt{: O}D\texttt{[}\overline{T}\texttt{]} \\
[\![Ctr]\!]_{visit} = \varnothing \\
[\![\texttt{class}\ C_1\texttt{[}\overline{X}\texttt{] extends}\ C_2\texttt{[}\overline{T}\texttt{]}\ \ldots]\!]_{default} = \texttt{def}\ c_1\texttt{[}\overline{X}\texttt{]= x =>}\ c_2\texttt{(x)} \\
[\![\texttt{object}\ C_1\ \texttt{extends}\ C_2\texttt{[}\overline{T}\texttt{]}\ \ldots]\!]_{default} = \texttt{def}\ c_1\ \texttt{=}\ c_2\texttt{(}C_1\texttt{)} \\
[\![\texttt{trait}\ D_1\texttt{[}\overline{X}\texttt{] extends}\ D_2\texttt{[}\overline{T}\texttt{]}\ \ldots]\!]_{default} = \texttt{def}\ d_1\ \texttt{= (x:}D_1\texttt{[}\overline{X}\texttt{]) =>}\ d_2\texttt{(x)} \\
[\![\texttt{@(default | visit)(}\overline{D}\texttt{) trait}\ V]\!] = \\
\quad \texttt{trait}\ V\ \texttt{extends}\ \overline{D\texttt{(Default | Visit)}}\texttt{\{ \_:}\overline{D\texttt{V}}\texttt{=>}\ldots\texttt{\}} \\
\quad \texttt{val}\ v : V \\
[\![\texttt{@(default | visit)(}\overline{D}\texttt{) trait}\ V\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}V}]\!] = \\
\quad \texttt{trait}\ V\ \texttt{extends}\ \overline{D\texttt{(Default | Visit)}}\ \texttt{with}\ \overline{\texttt{super[}F\texttt{].}V}\texttt{\{ \_:}\overline{D\texttt{V}}\texttt{=>}\ldots\texttt{\}} \\
[\![\overline{X}]\!] = \langle [\![X]\!] \mid X \in \overline{X} \rangle
\end{array}
$$

Figure 6: Translation.

7. Case Study I: Types and Programming Languages

In this section, we present a case study on modularizing the interpreters in TAPL [22]. The ARITH language and its variations are directly from or greatly inspired by the TAPL case study. TAPL are a good benchmark for examining CASTOR's capabilities of open pattern matching and modular dependencies. The reason is that core data structures of TAPL interpreters, types and terms, are modeled using algebraic datatypes; operations over types and terms are defined via pattern matching. There are a few operations that require nested patterns: small-step semantics, type equality, and subtyping relations. They all come with a default. The data structures and associated operations should be modular as new language features are introduced and combined. However, without proper support for modular pattern matching, the original implementation duplicates code for features that could be shared. With CASTOR and techniques shown in Section 2.9, we are able to refactor the non-modular implementation into a modular manner. Our evaluation shows that the refactored version significantly reduces the SLOC compared to a non-modular implementation found online. However, at the moment, improved modularity does come at some performance penalty.

7.1. Overview

An existing Scala implementation of TAPL[2] strictly follows the original OCaml version, which uses sealed case classes and pattern matching. The first ten languages (*arith*, *untyped*, *fulluntyped*, *tyarith*, *simplebool*, *fullsimple*, *fullerror*, *bot*, *rcdsubbot* and *fullsub*) are our candidates for refactoring. Each language implementation consists of 4 files: *parser*, *syntax*, *core* and *demo*. These languages cover various features including arithmetic, lambda calculus, records, fixpoints, error handling, subtyping, etc. Features are shared among these ten languages. However, such featuring sharing is achieved via duplicating code, causing problems like:

- **Inconsistent definitions.** Lambdas are printed as `"lambda"` in all languages except for *untyped*, where lambdas are printed as `"\"`.
- **Feature leaks.** Features introduced in the latter part of the book (e.g., System F) leak to previous language implementations such as *fullsimple*.

Our refactoring focuses on *syntax* and *core* where datatypes and associated operations are defined. Figure 7 gives a simplified high-level overview of the refactored implementation. The candidate languages are represented as gray boxes whereas extracted features/sub-languages are represented as white boxes. From Figure 7 we can see that the interactions between languages (revealed by the arrows) are quite intense. Take ARITH for example, it is a sublanguage for *tyarith*, *fulluntyped*, *fullerror*, *fullsimple* and *fullsub*. Unfortunately, without proper modularization techniques, the original implementation repeats the definition of *arith* at least five times. In the refactored implementation written with CASTOR, however, *arith* is defined only once and modularly reused in other places.

[2] `https://github.com/ilya-klyuchnikov/tapl-scala`

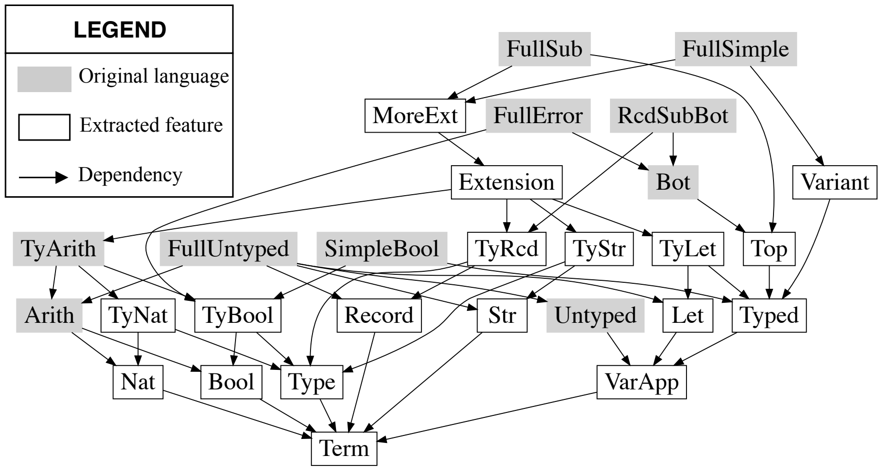

Figure 7: Simplified language/feature dependency graph.

7.2. Evaluation

We evaluate Castor by answering the following questions:

- **Q1.** Is Castor effective in reducing SLOC?
- **Q2.** How does Castor compare to EVF?
- **Q3.** How much performance penalty does Castor incur?

Q1. Table 2 reports the SLOC comparison results. With all the features/sublanguages extracted, implementing a candidate language with Castor is merely done by composing features/sublanguages. Therefore, the more features/sublanguages the candidate language uses, the more code Castor reduces. Compared to the non-modular Scala implementation, for a simple language like *arith*, the reduction rate[3] is 71%; for a feature-rich language like *fullsimple*, the reduction rate can be up to 96%. Overall, Castor reduces over *half* of the total SLOC with respect to the non-modular version.

Q2. Table 2 also compares Castor with EVF [21]. Castor reduces over 400 SLOC compared to EVF. As we have shown in Section 2, the reduction comes from the native support for pattern matching, generated dependency declarations, etc. More importantly, the instantiation burden for EVF is heavy if there are a lot of visitors and the dependencies are complex. In contrast, Castor completely removes the instantiation burden by generating companion objects automatically.

[3]Reduction rate = $\dfrac{\textbf{Scala}\text{ SLOC} - \text{Castor SLOC}}{\textbf{Scala}\text{ SLOC}} \times 100\%$

Table 2: SLOC evaluation of TAPL interpreters

Extracted	Castor	**EVF**	**Language**	Castor	**EVF**	**Scala**
bool	71	98	arith	31	33	106
extension	24	34	untyped	40	46	124
str	42	55	fulluntyped	18	47	256
let	48	47	tyarith	22	26	157
moreext	112	106	simplebool	24	38	212
nat	85	103	fullsimple	24	83	619
record	117	198	fullerror	68	105	396
top	79	86	bot	40	61	190
typed	82	138	rcdsubbot	30	39	257
varapp	40	65	fullsub	57	116	618
variant	136	161				
misc	212	172	**Total**	**1402**	**1857**	**2935**

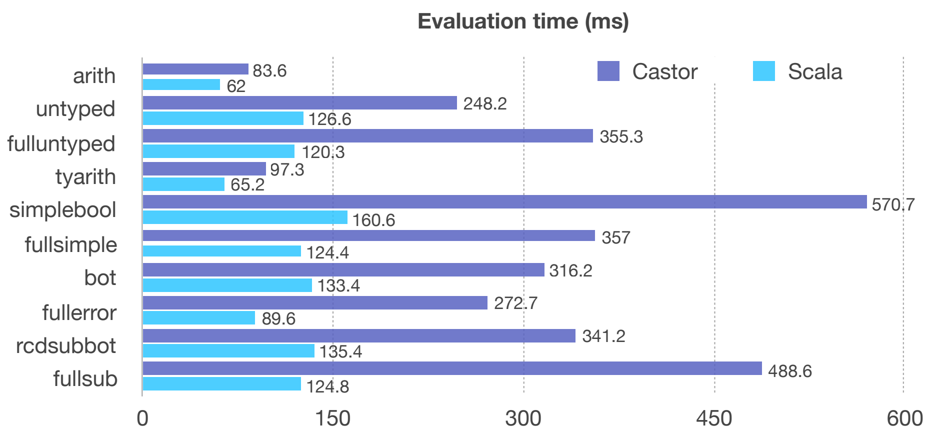

Figure 8: Performance evaluation of TAPL interpreters.

Q3. To measure the performance, we randomly generate 10,000 terms for each language and calculate the average evaluation time for 10 runs. The ScalaMeter[4] microbenchmark framework is used for performance measurements. The benchmark programs are compiled using Scala 2.12.7, JDK version 1.8.0_211 and are executed on a MacBook Pro with 2.3 GHz quad-core Intel Core i5 processor with 8 GB memory. Figure 8 compares the execution time in milliseconds. From the figure we can see that Castor implementations have a 1.35x (*arith*) to 3.92x (*fullsub*) slowdown with respect to the corresponding non-modular Scala implementations. The more features a modular implementation combines, the more significant the slowdown is. Figure 9 further compares the performance of the Scala Arith implementations discussed in Section 2. Obviously, modular implementations are slower than non-modular implementations. With the underlying optimizations, the implementation based on sealed case classes is

[4] `http://scalameter.github.io`

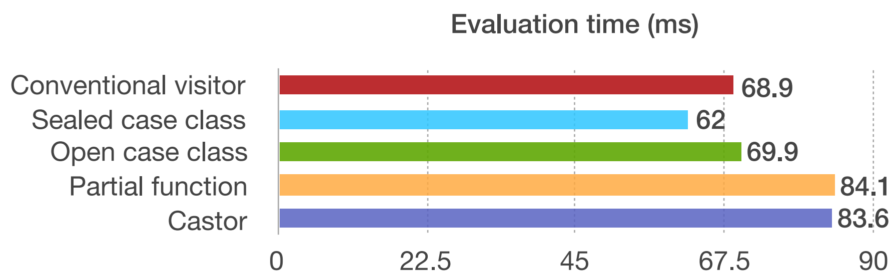

Figure 9: Performance evaluation of Arith.

faster than the implementation based on conventional visitors.

We believe that the performance penalty is mainly caused by method dispatching. A modular implementation typically has a complex inheritance hierarchy. Dispatching on a case needs to go across that hierarchy. Thus, the more complex the hierarchy is, the worse the performance is. Another source of performance penalty might be the use of functions instead of normal methods in visitors. Of course, more rigorous benchmarks need to be conducted to verify our guesses. One possible way to boost the performance is to turn TAPL interpreters into compilers via staging using the LMS framework [40]. This is currently not possible because LMS and Scalameta are incompatible in terms of the Scala compiler versions.

Threats to Validity. There are two major threats to the validity of our evaluation. The first threat is that measuring conciseness by counting SLOC may not be fair especially when different languages are used. We mitigate this threat by making the code style and the maximum character-per-line consistent for each implementation. The second threat is the representativeness of the TAPL interpreters. They are small languages for teaching purposes. It might still be questionable whether Castor scale to model larger languages that are actually used in practice. Nevertheless, TAPL interpreters have already covered a lot of core features that are available in mainstream languages.

8. Case Study II: UML Activity Diagrams

In the previous section, we evaluate the functional aspects of Castor. In this section, we evaluate the imperative aspects of Castor. To do so, we conduct another case study on a subset of the UML activity diagrams, which can be seen as a richer language than the FSM language discussed in Section 5. This case study examines hierarchical datatypes, imperative visitors and graphs.

8.1. Overview

An execution model of UML activity diagrams has been proposed as one of the challenges of the Transformation Tool Contest (TTC'15).

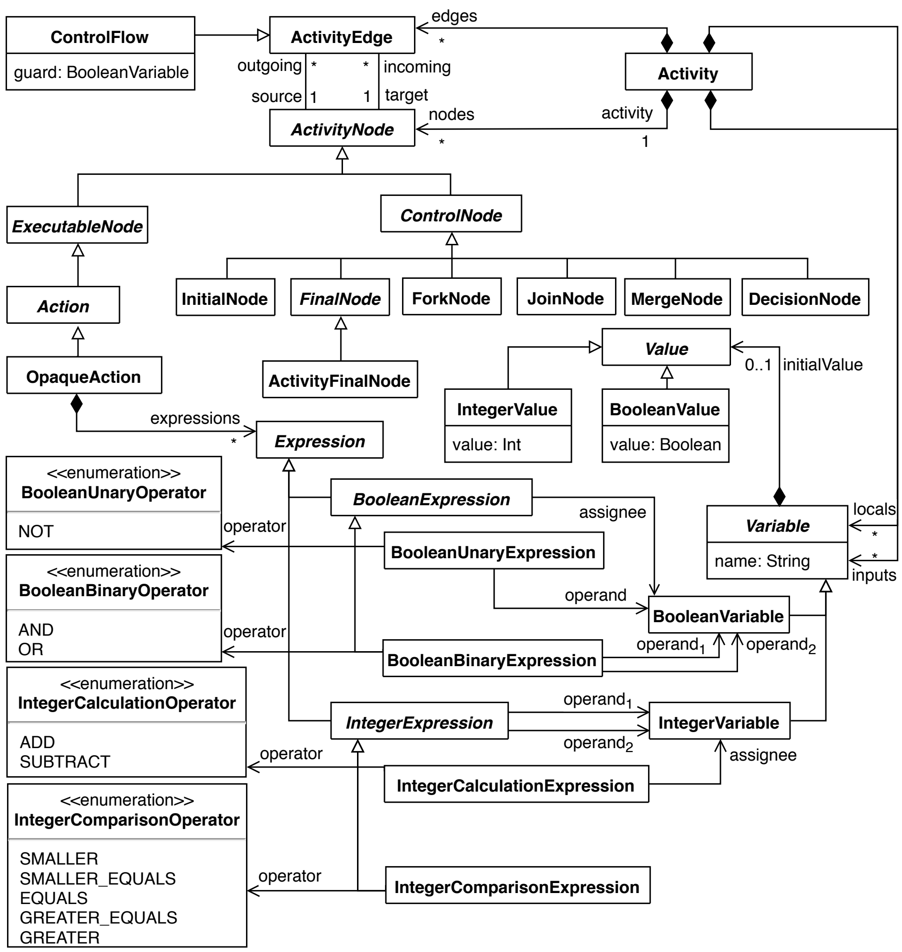

Figure 10: Metamodel of UML Activity Diagrams (an excerpt adapted after the TTC'15 document [41]).

Metamodel. Figure 10 shows the metamodel of UML activity diagrams, where ***Name*** denotes abstract classes and **Name** denotes concrete classes. An `Activity` object represents an instance of a UML activity diagram, which contains a sequence of `ActivityNodes` and `ActivityEdges`. `ExecutableNode` and `ControlNode` are two intermediate types of `ActivityNode` for classifying nodes that perform actions or control the flow. There are several concrete nodes. `InitialNode` and `ActivityFinalNode` are the start/end of activity diagrams; `DecisionNode` and `MergeNode` are the start/end of alternative branches; `ForkNode` and `JoinNode` are the start/end of concurrent branches. On the other hand, `OpaqueAction` sequentially executes a sequence of `Expressions`. `ActivityNodes` are connected by `ActivityEdges`. Similar to `GuardedTrans` discussed in Section 5.3, a `ControlFlow` is a specialized `ActivityEdge`, which is guarded by the current `BooleanValue` stored in a `BooleanVariable`. `Expressions` are also organized in a hierarchical way according to their types (`Boolean` or `Integer`) and the number of operands (`Unary` or `Binary`).

Goal and Challenges. The goal is to extend this simplified metamodel of UML activity diagrams with the dynamic execution semantics. The semantics is defined by performing transitions on activity nodes step by step using an imperative style. Several *runtime concepts* need to be introduced. Adding these runtime concepts poses two modularity challenges: *operation extensions* and *field extensions*. One example of an operation extension is `execute`, which is added to the `Expression` hierarchy for executing the calculation. One example of a field extension is a mutable boolean value `running`, which is added to `ActivityNode` for distinguishing triggered nodes from others.

Reference Implementation. The reference implementation[5] is written in Java with EMF [42]. The metamodel is described in Ecore from which Java interfaces are generated. Then semantics are encoded by defining classes that implement those interfaces using the INTERPRETER pattern [18]. The reference is non-modular because the INTERPRETER pattern facilitates adding new classes but lacks the ability to add new operations. Therefore, the reference implementation has to anticipate the operations on the metamodel. Moreover, consistent with what Figure 10 shows, operators were modeled as enumerations and recognized using `switch-case` clauses in Java, which are closed for extensions.

Refactored Implementation. Our refactoring focuses on the metamodel and semantics part only. Since the original implementation is written in Java, we first port it into Scala. We then refactor the ported implementation using CASTOR. Figure 11 gives an overview of the refactored implementation, which consists of 4 CASTOR components. Concretely, we make the following changes to the ported implementation for increasing modularity:

1. **Separate metamodel and operations.** With CASTOR, we do not need to foresee the operations on the metamodel since operations can be added modularly later. Thus, the refactored implementation separates the metamodel and operations upon it respectively in ***Model** and ***Lang**.

[5] `https://github.com/moliz/moliz.ttc2015`

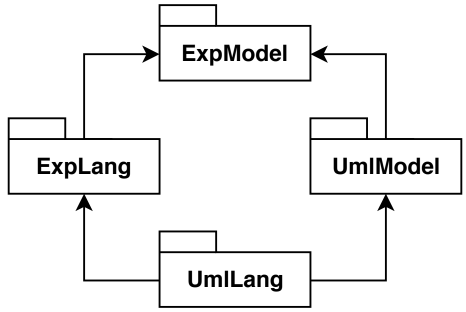

Figure 11: Refactored implementation.

2. **Expression language as an independently reusable component.** Values, variables and expressions are essentially a sublanguage independent of the UML activity diagrams. Instead of defining the expression sublanguage together with UML activity diagrams within a single `@family` component, we extract its metamodel into **ExpModel** and its semantics into **ExpLang**. This allows the expression sublanguage to be reused alone in other places.
3. **Overridden methods as visitors.** Methods that are overridden in the subclasses are rewritten as visitors, such as `isReady` and `fire` on `ActivityNode` and `execute` on `Expression`. Since only a few cases of `isReady` and `fire` are overridden whereas every case of `execute` is overridden, we use the default visitor (annotated as `@default`) for the former and the ordinary visitor (annotated as `@visit`) for the latter. For non-overridden methods, we move them out of a class and use an explicit argument to capture `this`.
4. **Operators as open datatypes.** Operators are refactored as `@adt` traits and their semantics are given by visitors for enabling extensions. This allows new kinds of operators such as multiplication to be added later.

8.2. Evaluation

We evaluate Castor's implementation by answering the following questions:

- **Q1.** Does the refactored implementation preserve the behavior of the ported implementation?
- **Q2.** Can Castor solve the modularity challenges?
- **Q3.** How does the refactoring affect the SLOC?
- **Q4.** Is the performance overhead reasonable?

Q1. To make sure that our refactoring does not affect the correctness of the implementation, we ran the test suite provided by the TTC'15 document. The test suite contains 6 small activity diagrams where all kinds of `ActivityNodes` and `Expressions` are covered. The refactored implementation passes all the tests in the test suite. This gives us some confidence that the refactored implementation preserves the behavior of the original implementation.

Table 3: Performance evaluation in milliseconds.

Name	Description	Interpreter	Castor
$test_1$	1000 sequential actions	22.1	56.6
$test_2$	100 parallel branches each with 10 actions	20.7	39.8
$test_3$	Similar to $test_2$ with a variable increased	22.8	39.9

Q2. For the operation extension challenge, the answer is yes. Operations are added by defining new visitors, which are fully modular. However, Castor does not address the field extension challenge very well. With the current version of Castor, we cannot extend existing classes with additional fields while keeping their names. The workaround is to introduce subclasses of different names. For example, if we want to extend `ActivityNode` with a field called `running`, we have to define a new class called `RuntimeActivityNode` that extends `ActivityNode` with `running`. The drawback is that `RuntimeActivityNode` and `ActivityNode` coexist and all existing operations need to be modified for handling `RuntimeActivityNode`. It is possible to have an alternative design for Castor, which does not introduce a new name while accomplishing field extensions in Castor. However, this brings some other complications. Such alternative design is discussed in Section 9.2.

Q3. The SLOC of the ported version and the refactored version are 489 and 411 respectively. Surprisingly, the refactoring brings extra modularity while reducing the SLOC. One reason is that in the ported version, methods are first declared in traits and then implemented in classes while the refactored version needs no prior declarations. Another reason is that by properly using Castor's default visitors and combined visitors, some definitions can be shortened. For example, `Execute` in the refactored version is a combined visitor for `Expression` and 4 operators.

Q4. We reuse the test suite from TTC'15 [41] which includes 3 large activity diagrams for measuring the performance. Table 3 gives a simple description for each test case and the average execution time for 10 runs (measured in milliseconds) for the two implementations using the same machine specified in Section 7. The Castor's implementation is around 2 to 3 times slower than the non-modular ported Interpreter implementation. The performance penalty would be reduced if we put everything in a single component. These results are similar to the results we get in Section 7 and further confirm that Castor's modular implementation introduces an acceptable performance penalty.

Threats to Validity. One threat to the validity of the evaluation is that the test suite is very small and might not be able to find out bugs that are introduced by refactoring. Also, directly comparing a Castor's implementation with respect to the reference implementation may be unfair since different programming languages are used. To exclude such language-wise factor on evaluation, we compared to the ported Scala implementation. As our focus is on the semantics part, irrelevant code like parsing is ignored.

9. Limitations and Design Options

In this section, we discuss the limitations of Castor. These limitations affect some of the design decisions we made that lead Castor to its current form. We discuss these design options and compromises.

9.1. Limitations

Castor has some limitations due to the use of metaprogramming and the restrictions from the current Scalameta library:

- **Unnecessary annotations.** With the current version of Scalameta, we are not able to get information from annotated parents. If parents' information were accessible, annotations `@adts` and `@ops` could be eliminated.

- **Boilerplate for nested composition.** Lacking of parents' information also disallows automatically composing nested members. Assuming that automatic nested composition is available, `Arith` can be simplified as:

  ```
  @family trait Arith extends Nat with Bool {
    @adt trait Tm { ... }
    @visit(Tm) trait Eval1 { ... }
  }
  ```

  By expressing the inheritance relationship once at the family level, `extend` clauses for members such as `super[Nat].Tm with super[Bool].Tm` can be inferred.

- **Imprecise error messages.** As Castor modifies the annotated programs, what the compiler reports are errors on the modified program rather than the original program. Reasoning about the error messages becomes harder as they are mispositioned and require some understanding of the generated code.

9.2. Design Options

Nested Patterns. There is an alternative way of writing nested patterns. For example, `tmIf` can be rewritten in the following way:

```
override def tmIf = x => x.t1 match {
  case TmTrue => x.t2
  case TmFalse => x.t3
  case t1 => TmIf(this(t1),x.t2,x.t3)
}
```

Instead of directly pattern matching on an `TmIf` object, we capture it first a variable `x` and then explicitly `match` on its subterm `t1`. For the case of `tmIf`, this alternative implementation is arguably less intuitive than the version we presented in Section 2.9. Nevertheless, this approach comes in handy when: 1) the object being matched contains a lot of fields and most of them are not interesting in nested patterns; 2) there are a lot of `case` clauses for nested patterns and repeating the top-level pattern in each `case` clause becomes tedious.

Specialized Visitors. Programming with visitors can be simplified using specialized visitors. The default visitors generated by Castor (annotated as `@default`) are an instance. In fact, there are more such specialized visitors. For example, visitors can be combined with *visitor combinators* [43]; boilerplate in querying and transforming the data structure can be eliminated by *traversal templates* [21]. Essentially, these specialized visitors can also be generated by Castor. Currently, only default visitors are generated because 1) in our experience they are most frequently used; 2) generating all other infrequently used specialized visitors increases the time of code generation and the size of generated code. Ideally, specialized visitors should be generated by need. Limited by current Scalameta, this is impossible for the moment.

Refinable Variants. As our visitor encoding shows, the key to extensibility is capturing concrete types with bounded type members for allowing future refinements. The same idea can also be applied to variants, where the visitor method signature refers to a type member instead of a class name. By doing this, we are able to extend that class with additional fields seamlessly by covariantly refining the type member to the new class. An application of refinable variants would be guarded transitions discussed in Section 5.3:

```
class Trans(event: String, to: State, var tm: Tm[Boolean] = TmTrue)
  extends super.Trans(event, to)
```

Instead of adding a new variant called `GuardedTrans`, we refine the existing `Trans`. The benefit is that existing visitors that do not concern about the additional parameter `tm` can be unchanged. In contrast, for the case of `GuardedTrans`, we have to update all existing visitors with an implementation of `guardedTrans`. The downside of supporting refinable variants in Castor is that it brings more book-keeping burden on variants for the user. We consider the price to pay is higher than the benefit it brings.

10. Related Work

Object-Oriented Pattern Matching. There are many attempts to bring notions similar to pattern matching into OOP. Multimethods [3, 44] allow a series of methods of the same signature to co-exist. The dispatching for these methods additionally takes the runtime type of arguments into consideration so that the most specific method is selected. Pattern matching on multiple arguments can be simulated with multimethods. However, it is unclear how to do deep patterns with multimethods. Also, multimethods significantly complicate the type system. As we have discussed in Section 2, case classes in Scala [31] provide an interesting blend between algebraic datatypes and class hierarchies. Sealed case classes are very much like classical algebraic datatypes, and facilitate exhaustiveness checking at the cost of a closed (non-extensible) set of variants. Open case classes support pattern matching for class hierarchies, which can modularly add new variants. However no exhaustiveness checking is possible for open case classes. Besides case classes, extractors [32] are another alternative pattern matching mechanism in Scala. An extractor is a companion `object` with a user-defined `unapply` method that specifies how to tear down that object. Unlike case classes whose `unapply` method is automated and hidden, extractors are flexible, independent of classes but verbose. There are also proposals to extend mainstream languages with pattern matching such as

Java. JMatch [45] extends Java with pattern matching using modal abstraction. JMatch methods additionally have backward modes that can compute the arguments from a given result, serving as patterns. Follow-up work [46] extends JMatch with exhaustiveness and totality checking on patterns in the presence of subtyping and inheritance. However, it requires a non-trivial language design with the help of an SMT solver. More recent OO languages like Newspeak [47] and Grace [48] are designed with first-class pattern matching, where patterns are objects and can easily be combined. To the best of our knowledge, none of these approaches fully meet the desirable properties summarized in Section 2.1.

Modular Church-Encoded Interpreters. Solutions to the Expression Problem based on Church encodings can also be used for developing modular interpreters. Well-known techniques are Finally Tagless [11], Object Algebras [9] and Polymorphic Embedding [12]. However, these techniques do not support pattern matching or dependencies, making it hard to define operations like small-step semantics discussed in Section 2. Although Kiselyov [23] show that operations requiring nested patterns can be rewritten as context-sensitive operations, the operations become much more convoluted. Typical workarounds on dependent operations are defining the operation together with the dependencies or using advanced features like intersection types and a merge operator [49, 50]. In contrast, Castor allows us to implement operations that need nested patterns and/or with dependencies in a simple, modular way.

Polymorphic Variants. OCaml supports polymorphic variants [51]. Unlike traditional variants, polymorphic variant constructors are defined individually and are not tied to a particular datatype. Garrigue [52] presents a solution to the Expression Problem using polymorphic variants. To correctly deal with recursive calls, open recursion and an explicit fixed-point operator must be used properly. Otherwise, the recursion may go to the original function rather than the extended one. This causes additional work for the programmer, especially when the operation has complex dependencies. In contrast, Castor handles open recursion easily through OO dynamic dispatching, reducing the burden of programmers significantly.

Open Datatypes and Open Functions. To solve the Expression Problem, Löh and Hinze [53] propose to extend Haskell with open datatypes and open functions. Different from classic closed datatypes and closed functions, the open counterparts decentralize the definition of datatypes and functions and there is a mechanism that reassembles the pieces into a complete definition. To avoid unanticipated captures caused by classic *first-fit* pattern matching, a *best-fit* scheme is proposed, which rearranges patterns according to their specificness rather than the order (e.g. wildcards are least specific). However open datatypes and open functions are not supported in standard Haskell and more importantly, they do not support separate compilation: all source files of variants belonging to the same datatype must be available for code generation.

Data Types à la Carte (DTC). DTC [54] encodes composable datatypes using existing features of Haskell. The idea is to express extensible datatypes as a fixpoint of coproducts of functors. While it is possible to define operations that have dependencies or require nested pattern matching with DTC, the encoding becomes complicated and needs significant machinery. There is some follow-up work that tries to equip DTC with additional power. Bahr and Hvitved [55] extend DTC with GADTs [37] and automatically generates boilerplate using Template Haskell [56]. Oliveira et al. [57] use list-of-functors instead of co-products to better simulate OOP features including subtyping, inheritance, and overriding.

Language Workbenches. To reduce the engineering effort involved in software language development, language workbenches [58, 59] have been proposed. Modularity is an important concern in language workbenches for allowing existing language components to be reused in developing new languages [60]. Traditionally most of the work on language workbenches has focused on *syntatic modularity* approaches. More *semantic modularity* aspects such as separate compilation and modular typechecking are not well addressed. However, more recent work on language workbenches has started to incorporate semantic modularity techniques. We compare our work next, to the language workbenches that employ semantic modularization techniques. With Neverlang [61], users do not directly program with visitors. Instead, they have to use a DSL and learn specific concepts such as slice and roles. MontiCore [62] generates the visitor infrastructure from its grammar specification. To address the extensibility issue, MontiCore overrides the `accept` method and uses casts for choosing the right visitor for extended variants, thus is not type-safe. Also, MontiCore supports imperative style visitors only. Alex [63] also provides a form of semantic modularity based on the *Revisitor* pattern [64], which can be viewed as a combination of Object Algebras and Walkabout [65]. By moving the dispatching method from the class hierarchy to the visitor interface, the *Revisitor* pattern can work for legacy class hierarchies that do not anticipate the usage of visitors. However, the dispatching method generated by Alex is implemented using casts and has to be modified whenever new variants are added, thus is neither modular nor type-safe. Castor fully supports semantic modularity and allows users to do the development using their familiar language with a few annotations. For the moment, Castor still lacks much of the functionality for various other aspects of language implementations that are covered by language workbenches. Nevertheless, the modularization techniques employed by Castor could be useful in the context of language workbenches to improve reuse and type-safety of language components, in the same way that visitors are used in Neverlang and Revisitors are used in Alex.

11. Conclusion and Future Work

In this paper, we have presented Castor, a Scala framework for programming with extensible, generative visitors using simple annotations. Visitors written with Castor are type-safe, concise, exhaustive, extensible and composable. Moreover, both functional and imperative style visitors are supported. We have shown how to use Castor in designing a better pattern matching mechanism in an OO context, developing modular well-typed EDSLs, doing extensible programming on graphs, etc. The effectiveness of Castor is validated by our case studies on TAPL interpreters and UML activity diagrams. While Castor is practical and serves the purpose of programming with visitors, there are important drawbacks on such a meta-programming, library-based approach: error reporting is imprecise; the syntax and typing of Scala cannot be changed to enforce certain restrictions. In future work, we would like to design a language with a better surface syntax that supports first-class visitors. Another direction of future work is to grow CASTOR into a language workbench by additionally supporting syntax and associated tools development.

Acknowledgement

We thank the reviewers for their helpful comments that significantly improve the presentation of this paper. This work was funded by Hong Kong Research Grant Council projects number 17210617 and 17258816.

References

[1] P. Wadler, The Expression Problem, Email, discussion on the Java Genericity mailing list (Nov. 1998).

[2] O. L. Madsen, B. Moller-Pedersen, Virtual classes: A powerful mechanism in object-oriented programming, in: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPSLA '89, ACM, New York, NY, USA, 1989, pp. 397–406. doi:10.1145/74877.74919.
URL http://doi.acm.org/10.1145/74877.74919

[3] C. Chambers, Object-oriented multi-methods in cecil, in: European Conference on Object-Oriented Programming, 1992.

[4] E. Ernst, Family polymorphism, in: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, Springer-Verlag, London, UK, UK, 2001, pp. 303–326.
URL http://dl.acm.org/citation.cfm?id=646158.680013

[5] G. Bracha, W. Cook, Mixin-based inheritance, in: Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications, OOPSLA/ECOOP '90, ACM, New York, NY, USA, 1990, pp. 303–311. doi:10.1145/97945.97982.
URL http://doi.acm.org/10.1145/97945.97982

[6] A. Moors, F. Piessens, M. Odersky, Generics of a higher kind, in: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08, ACM, New York, NY, USA, 2008, pp. 423–438. doi:10.1145/1449764.1449798.
URL http://doi.acm.org/10.1145/1449764.1449798

[7] K. K. Thorup, Genericity in java with virtual types, in: European Conference on Object-Oriented Programming, Springer, 1997, pp. 444–471.

[8] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Trans. Program. Lang. Syst. 28 (2) (2006) 331–388. doi:10.1145/1119479.1119483.
URL http://doi.acm.org/10.1145/1119479.1119483

[9] B. C. d. S. Oliveira, W. R. Cook, Extensibility for the masses: Practical extensibility with object algebras, in: Proceedings of the 26th European Conference on Object-Oriented Programming, 2012.

[10] B. C. d. S. Oliveira, Modular visitor components, in: Proceedings of the 23rd European Conference on Object-Oriented Programming, 2009.

[11] J. Carette, O. Kiselyov, C.-c. Shan, Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages, Journal of Functional Programming 19 (5) (2009) 509–543.

[12] C. Hofer, K. Ostermann, T. Rendel, A. Moors, Polymorphic embedding of dsls, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, 2008.

[13] A. Church, An unsolvable problem of elementary number theory, American journal of mathematics 58 (2) (1936) 345–363.

[14] D. Scott, A system of functional abstraction, Unpublished manuscript (1963).

[15] R. Hinze, Generics for the Masses, Journal of Functional Programming 16 (4-5) (2006) 451–483. doi:10.1017/S0956796806006022.

[16] B. C. d. S. Oliveira, R. Hinze, A. Löh, Extensible and Modular Generics for the Masses, in: Trends in Functional Programming, 2006, pp. 199–216.

[17] B. C. d. S. Oliveira, J. Gibbons, Typecase: A design pattern for type-indexed functions, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, Haskell '05, 2005.

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

[19] P. Buchlovsky, H. Thielecke, A type-theoretic reconstruction of the visitor pattern, Electron. Notes Theor. Comput. Sci. 155 (2006) 309–329. doi:10.1016/j.entcs.2005.11.061.
URL http://dx.doi.org/10.1016/j.entcs.2005.11.061

[20] J. Gibbons, Origami programming, 2003, pp. 41–60.
URL http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/origami.pdf

[21] W. Zhang, B. C. d. S. Oliveira, Evf: An extensible and expressive visitor framework for programming language reuse, in: European Conference on Object-Oriented Programming, 2017.

[22] B. C. Pierce, Types and programming languages, MIT press, 2002.

[23] O. Kiselyov, Typed tagless final interpreters, in: Generic and Indexed Programming, Springer, 2012, pp. 130–174.

[24] T. Millstein, C. Bleckner, C. Chambers, Modular typechecking for hierarchically extensible datatypes and functions, ACM Trans. Program. Lang. Syst. 26 (5) (Sep. 2004).

[25] W. Zhang, B. C. d. S. Oliveira, Pattern matching in an open world, in: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, 2018.

[26] R. Milner, M. Tofte, R. Harper, D. Macqueen, The definition of standard ml-revised (1997).

[27] S. P. Jones, Haskell 98 language and libraries: the revised report, Cambridge University Press, 2003.

[28] B. Meyer, K. Arnout, Componentization: the visitor example, Computer 39 (7) (2006) 23–30.

[29] T. Pati, J. H. Hill, A survey report of enhancements to the visitor software design pattern, Software: Practice and Experience 44 (6) (2014) 699–733.

[30] R. C. Martin, The Principles, Patterns, and Practices of Agile Software Development, Prentice Hall, 2002.

[31] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the scala programming language, Tech. rep. (2004).

[32] B. Emir, M. Odersky, J. Williams, Matching objects with patterns, in: European Conference on Object-Oriented Programming, 2007.

[33] M. Zenger, M. Odersky, Extensible algebraic datatypes with defaults, in: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, 2001.

[34] M. Odersky, M. Zenger, Independently extensible solutions to the expression problem, in: The 12th International Workshop on Foundations of Object-Oriented Languages, 2005.

[35] C. Hofer, K. Ostermann, Modular domain-specific language components in scala, in: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE '10, 2010.

[36] M. E. Nordberg III, Variations on the visitor pattern, in: PLoP'96 Writer's Workshop, Vol. 154, 1996.

[37] H. Xi, C. Chen, G. Chen, Guarded recursive datatype constructors, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '03, 2003.

[38] F. Pfenning, C. Elliott, Higher-order abstract syntax, in: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, ACM, New York, NY, USA, 1988, pp. 199–208. doi:10.1145/53990.54010.

[39] E. Burmako, Unification of compile-time and runtime metaprogramming in scala, Ph.D. thesis, EPFL (2017).

[40] T. Rompf, M. Odersky, Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, in: In GPCE, 2010.

[41] T. Mayerhofer, M. Wimmer, The ttc 2015 model execution case., in: TTC@ STAF, 2015, pp. 2–18.

[42] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[43] J. Visser, Visitor combination and traversal control, in: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01, ACM, New York, NY, USA, 2001, pp. 270–282. doi:10.1145/504282.504302.

[44] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, Multijava: Modular open classes and symmetric multiple dispatch for java, in: ACM Sigplan Notices, Vol. 35, ACM, 2000, pp. 130–145.

[45] J. Liu, A. C. Myers, Jmatch: Iterable abstract pattern matching for java, in: PADL, 2003.

[46] C. Isradisaikul, A. C. Myers, Reconciling exhaustive pattern matching with objects, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, 2013.

[47] F. Geller, R. Hirschfeld, G. Bracha, Pattern Matching for an object-oriented and dynamically typed programming language, no. 36, Universitätsverlag Potsdam, 2010.

[48] M. Homer, J. Noble, K. B. Bruce, A. P. Black, D. J. Pearce, Patterns as objects in grace, in: Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, New York, NY, USA, 2012, pp. 17–28.

[49] B. C. d. S. Oliveira, T. v. d. Storm, A. Loh, W. R. Cook, Feature-oriented programming with object algebras, in: Proceedings of the 27th European Conference on Object-Oriented Programming, 2013.

[50] T. Rendel, J. I. Brachthäuser, K. 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] J. Garrigue, Programming with polymorphic variants, in: ML Workshop, 1998.

[52] J. Garrigue, Code reuse through polymorphic variants, in: Workshop on Foundations of Software Engineering, 2000.

[53] A. Löh, R. Hinze, Open data types and open functions, in: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming, 2006.

[54] W. Swierstra, Data types à la carte, Journal of functional programming 18 (4) (2008) 423–436.

[55] P. Bahr, T. Hvitved, Compositional data types, in: Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, ACM, 2011, pp. 83–94.

[56] T. Sheard, S. P. Jones, Template meta-programming for haskell, in: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, 2002.

[57] B. C. d. S. Oliveira, S.-C. Mu, S.-H. You, Modular reifiable matching: A list-of-functors approach to two-level types, in: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, Haskell '15, 2015.

[58] M. Fowler, Language workbenches: The killer-app for domain specific languages, http://martinfowler.com/articles/languageWorkbench.html (2005).

[59] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al., The state of the art in language workbenches, in: International Conference on Software Language Engineering, 2013.

[60] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, et al., Concern-oriented language development (cold): Fostering reuse in language engineering, Computer Languages, Systems & Structures 54 (2018) 139–155.

[61] E. Vacchi, W. Cazzola, Neverlang: A framework for feature-oriented language development, Computer Languages, Systems & Structures 43 (2015) 1–40.

[62] R. Heim, P. M. S. Nazari, B. Rumpe, A. Wortmann, Compositional language engineering using generated, extensible, static type-safe visitors, in: European Conference on Modelling Foundations and Applications, 2016.

[63] M. Leduc, T. Degueule, B. Combemale, Modular language composition for the masses, in: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, ACM, 2018, pp. 47–59.

[64] M. Leduc, T. Degueule, B. Combemale, T. Van Der Storm, O. Barais, Revisiting visitors for modular extension of executable dsmls, in: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2017, pp. 112–122.

[65] J. Palsberg, C. B. Jay, The essence of the visitor pattern, in: Proceedings of the 22nd International Computer Software and Applications Conference, 1998.