Pattern Matching in an Open World*

Weixin Zhang
The University of Hong Kong
Hong Kong, China
wxzhang2@cs.hku.hk

Bruno C. d. S. Oliveira
The University of Hong Kong
Hong Kong, China
bruno@cs.hku.hk

Abstract

Pattern matching is a pervasive and useful feature in functional programming. There have been many attempts to bring similar notions to Object-Oriented Programming (OOP) in the past. However, a key challenge in OOP is how pattern matching can coexist with the open nature of OOP data structures, while at the same time guaranteeing other desirable properties for pattern matching.

This paper discusses several desirable properties for pattern matching in an OOP context and shows how existing approaches are lacking some of these properties. We argue that the traditional semantics of pattern matching, which is based on the *order* of patterns and adopted by many approaches, is in conflict 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, is worthwhile considering in an OOP context. To compensate for the absence of ordered patterns we propose a complementary mechanism for case analysis with defaults, which can be used when nested and/or multiple case analysis is needed. To illustrate our points we develop Castor: a meta-programming library in Scala that adopts both ideas. Castor generates code that uses type-safe extensible visitors, and largely removes boilerplate code typically associated with visitors. We illustrate the applicability of our approach with a case study modularizing the interpreters in the famous book "Types and Programming Languages".

CCS Concepts • **Software and its engineering → Language features**;

Keywords Pattern matching, Meta-programming, OOP

*This work was funded by Hong Kong Research Grant Council projects number 17210617 and 17258816.

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278124

ACM Reference Format:
Weixin Zhang and Bruno C. d. S. Oliveira. 2018. Pattern Matching in an Open World. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3278122.3278124

1 Introduction

Pattern matching is a pervasive and useful feature in functional programming. Languages such as Haskell [20] or ML [24] use algebraic datatypes to model data structures, and pattern matching to process such data structures. Algebraic datatypes and pattern matching allow concise programs for many applications. For example, compilers, interpreters or program analysis, often require extensive analysis on a complex Abstract Syntax Tree (AST) structure. In functional languages, ASTs are modeled with algebraic datatypes. With built-in support for pattern matching, analyzing and manipulating an AST can be done in a concise way.

Object-Oriented Programming (OOP) often 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.

There have been many attempts to bring notions similar to pattern matching to OOP in the past. The Visitor pattern [10] is frequently used as a poor man's approach to pattern matching in OO languages. However, classic formulations of the Visitor pattern have a high notational overhead, and also lack extensibility for dealing with new data variants. More recently several new designs for *extensible visitors* [15, 26, 27, 42] provide variations of the Visitor pattern that allow for the modular addition of new variants. However, these techniques do not solve the high notational overhead problem and do not provide a concise notation for pattern matching. Case classes in Scala [25] provide an interesting blend between algebraic datatypes and class hierarchies. Case classes come in different flavors. 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. *Multiple dispatching* [8] and *multi-methods* [7] provide OOP alternatives to pattern matching, enabling method dispatching to be determined by multiple arguments at run-time (rather than just one). This facilitates the definition of binary (and n-ary) operations such as equality, but it does not provide immediate support for nested patterns. Furthermore, type systems supporting multiple dispatching add significant complexity. Finally, there are also approaches [19, 23] that attempt to do a more principled design that integrates pattern matching and OOP while preserving the ability to add new variants and the ability to check for exhaustiveness. However, such approaches require new non-trivial language designs and thus cannot be used in existing languages such as Scala.

This paper starts by identifying several desirable properties for pattern matching 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.
- **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.

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.

Therefore we suggest two different mechanisms to deal with patterns in an OO context. Firstly, we suggest a more restricted, top-level pattern matching model, where the order of patterns is irrelevant. Secondly, to compensate for the absence of ordered patterns we propose a second mechanism for case analysis with defaults, which can be used when nested and/or multiple case analysis is needed. The second mechanism is directly inspired by Zenger and Odersky [40]'s idea of *Extensible Algebraic Datatypes with Defaults* (EADDs). In EADDs the key idea is that if pattern matching always comes with a default then it is always exhaustive. However, the key problem with EADDs is that not all operations have a good default. In our approach, top-level pattern matching does not force programmers to define a default, while still retaining exhaustiveness. However, we argue that, in practice, many operations that require nested patterns tend to have a good default for the nested patterns. Thus we propose applying the EADDs idea *only* to nested patterns.

To illustrate our points we develop Castor[1]: a metaprogramming library in Scala that adopts both ideas, and to a large extent, meets all the properties summarized above. The key idea is to combine case classes with extensible visitors so that top-level case analysis is done using visitors, while nested case analysis is done using Scala's own built-in pattern matching. Castor eliminates the complexity and verbosity of visitors by providing users with annotations. Through macro annotation processing, the annotated program is transformed and boilerplate automatically generated. We illustrate the applicability of our approach with a case study modularizing the interpreters in the famous book "Types and Programming Languages" (TAPL) [31].

In summary, this paper makes the following contributions:

- **Desirable properties for open pattern matching:** We summarize the desirable properties of pattern matching and evaluate existing approaches accordingly (Section 2).
- **The Castor metaprogramming library:** We present a novel encoding for modular pattern matching based on an encoding of extensible visitors. The pattern matching encoding is automated using metaprogramming (Section 4).
- **Non-trivial modular operations:** We show how to use Castor in defining non-trivial pattern matching operations, as well as dependencies (Section 3).
- **Case study:** We conduct a case study on TAPL that illustrate the effectiveness of Castor (Section 5).

Source code for Castor and case study is available at:

https://github.com/wxzh/Castor

2 Pattern Matching in Scala: An Evaluation

This section reviews existing approaches to pattern matching in Scala. To facilitate our discussion, a running example from TAPL [31]—an untyped, arithmetic language called Arith—is used. Our goal is to model the syntax and semantics of Arith in a concise and modular manner. None of the existing Scala approaches, including the Visitor pattern, sealed case classes, open case classes and partial functions, fully accomplishes the task. We discuss why these approaches fail and describe the desirable properties for a better solution.

[1] Castor stands for CASe class visiTOR

$$t ::= 0 \mid \texttt{succ}\ t \mid \texttt{pred}\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{iszero}\ t \qquad nv ::= 0 \mid \texttt{succ}\ nv$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \qquad \frac{}{\texttt{pred}\ 0 \longrightarrow 0}\ \textsc{PredZero} \qquad \frac{}{\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1}\ \textsc{PredSucc} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'}\ \textsc{Pred}$$

$$\frac{}{\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_2} \qquad \frac{}{\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_3} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}$$

$$\frac{}{\texttt{iszero}\ 0 \longrightarrow \texttt{true}} \qquad \frac{}{\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false}} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'}$$

Figure 1. The syntax and semantics of Arith.

2.1 Running Example: Arith

The syntax and semantics of Arith are formalized in Figure 1. 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 one 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 4 desirable properties of pattern matching summarized in Section 1 because: 1) The small-step style semantics is best expressed with a concise *nested case analysis* on terms; 2) Arith is a unification of two sublanguages, Nat (zero, successor and predecessor) and Bool (true, false, and conditional) with an extension (zero test). An ideal implementation of Arith is to have Nat and Bool *separately defined* and *modularly reused*.

2.2 The Visitor Pattern

The Visitor pattern [10] is often used in OOP languages for simulating pattern matching. Figure 2 implements the Nat sublanguage using the Visitor pattern. The class hierarchy models the abstract syntax of Nat, where the abstract class `Tm` represents the datatype of terms and syntactic constructs of terms are concrete subclasses of `Tm`. A generic `accept` method is defined throughout the class hierarchy, which is implemented by invoking the corresponding lowercase method defined on the visitor instance it takes. These lowercase methods are declared in the visitor interface `TmVisit`. Operations over `Tm` are concrete implementations of `TmVisit`. For example, object `nv` instantiates the type parameter `A` of `TmVisit` as `Boolean` and implements each visit method accordingly. For recursive cases like `tmSucc`, we call `t.accept(nv)`.

Discussion of the Approach. A well-known criticism of the Visitor pattern is its verbosity, which is manifested in encoding the small-step semantics in `eval1`. Defining `tmZero`

```
object NoRuleApplies extends Exception
// Class hierarchy
abstract class Tm {
  def accept[A](v: TmVisit[A]): A }
object TmZero extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmZero }
class TmSucc(t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmSucc(t) }
class TmPred(t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmPred(t) }
// Visitor interface
trait TmVisit[A] {
  def tmZero: A
  def tmSucc(t: Tm): A
  def tmPred(t: Tm): A }
// Numeric value checking visitor
object nv extends TmVisit[Boolean] {
  def tmZero = true
  def tmSucc(t: Tm) = t.accept(nv)
  def tmPred(t: Tm) = false }
// Small-step evaluation visitor
object eval1 extends TmVisit[Tm] {
  def tmZero = throw NoRuleApplies
  def tmSucc(t: Tm) = new TmSucc(t.accept(eval1))
  def tmPred(t: Tm) = t.accept(new TmVisit[Tm] {
    // Anonymous visitor
    def tmZero = TmZero
    def tmSucc(t1: Tm) = // PredSucc
      if (t1.accept(nv)) t1
      else new TmPred(t.accept(eval1))
    def tmPred(t1: Tm) = new TmPred(t.accept(eval1))
})}
```

Figure 2. Implementing Nat with the Visitor pattern.

and `tmSucc` for `eval1` is easy by throwing an exception and calling `eval1` on the inner term respectively. However, defining `tmPred` is tricky! Take the PredSucc rule for example, which cancels a pair of predecessor and successor application to a numeric value. As a visitor recognizes only one level representation of a term, it is insufficient to implement rules that require deep case analysis like PredSucc. To further reveal the shape of the inner term, an auxiliary anonymous visitor is hence needed. Then rules like PredSucc are possible to be specified inside that anonymous visitor.

```
// Sealed case class hierarchy
sealed abstract class Tm
case object TmZero extends Tm
case class TmSucc(t: Tm) extends Tm
case class TmPred(t: Tm) extends Tm
// Numeric value checking function
def nv(t: Tm): Boolean = t match {
  case TmZero => true
  case TmSucc(t1) => nv(t1)
  case _ => false }
// Small-step evaluation function
def eval1(t: Tm): Tm = t match {
  case TmSucc(t1) => TmSucc(eval1(t1))
  case TmPred(TmZero) => TmZero
  case TmPred(TmSucc(t1)) if nv(t1) => t1 // PredSucc
  case TmPred(t1) => TmPred(eval1(t1))
  case _ => throw NoRuleApplies }
```

Figure 3. Implementing Nat with sealed case classes.

A problem arises when we try to reuse the Nat implementation above in implementing Arith. The Visitor pattern suffers from the Expression Problem (EP) [38]: 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 the `Tm` hierarchy is tightly coupled with the `TmVisit` interface. When trying to add new subclasses to the `Tm` hierarchy, we are unable 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 EP. Thus, the implementation is neither extensible nor composable. Even if modification is allowed, the implementation would become much more tedious. There is a lot of boilerplate needs to be written for boring cases (e.g. cases that return `false` in `nv`). Nevertheless, exhaustiveness is enforced since a concrete visitor is an `object`, which must implement *all* visit methods exposed by the visitor interface.

2.3 Sealed Case Classes

Pattern matching, a feature originally from functional languages, offers the ability to "inspect and decompose data simultaneously". This ability makes it possible to model Nat concisely. As a programming language that unifies functional and OO paradigms, Scala [25] supports first-class pattern matching via case classes/extractors [9]. Figure 3 shows an implementation using Scala's sealed case classes, which is close to an implementation written in a pure functional language like Haskell or ML. A case class hierarchy models the abstract syntax. The `case` keyword triggers the compiler to automatically inject methods into the class/object, including a constructor method (`apply`) and an extractor method (`unapply`). The injected constructor method simplifies the creation of terms. For example, a successor application to constant zero can be constructed via `TmSucc(TmZero)`. Conversely, the injected extractor enables tearing down a term via pattern matching, as illustrated by the implementation of `nv`. 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 `TmZero` and returns a `true` eventually. A *wildcard pattern* (`_`) is used for handling boring cases altogether.

Discussion of the Approach. The strength of pattern matching shines in encoding the small-step semantics. With the help of pattern matching, the overall definition of `eval1` is a direct mapping from the formalization shown in Figure 1. As an example, the PredSucc rule is concisely and precisely described by a *deep* pattern (`TmPred(TmSucc(t1))`) with a *guard* (`if nv(t1)`). Furthermore, sealed case classes facilitate exhaustiveness checking on patterns. 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. 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.

2.4 Open Case Classes

Case classes in Scala can also be open, as the `sealed` keyword is optional. By removing `sealed`, we exchange exhaustiveness checking for the ability to add new variants in separate files. Combined with EADDs [40], it is possible to implement Arith in a modular manner. The idea is to use a default (wildcard pattern) 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. Figure 4 shows how to modularize Arith using open case classes. Note that `Tm` is declared as a top-level open datatype.

Discussion of the Approach. One nice aspect about this approach is that sublanguages, Nat and Bool, are now implemented separately into two traits `Nat` and `Bool`. `Nat` and `Bool` introduce their respective variants of `Tm` and a corresponding definition of `eval1`. The definition `nv` defined by `Nat` works well in `Arith`, as it happens to have a very good default that automatically works for extended cases. For instance, calling `nv(TmFalse)` returns `false` as expected.

Unfortunately, `eval1` is more problematic. In the general case defining `Arith` in terms of `Nat` and `Bool` causes problems as EADDs do not work well for such *non-linear extensions*. `eval1` needs to be overridden for combining definitions from

```
abstract class Tm
trait Nat {
  case object TmZero extends Tm
  case class TmSucc(t: Tm) extends Tm
  case class TmPred(t: Tm) extends Tm
  def nv(t: Tm): Boolean = ... // Same as in Figure 3
  def eval1(t: Tm): Tm = ... // Same as in Figure 3
}
trait Bool {
  case object TmTrue extends Tm
  case object TmFalse extends Tm
  case class TmIf(t1: Tm,t2: Tm,t3: Tm) extends Tm
  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 _ => throw NoRuleApplies
}}
trait Arith extends Nat with Bool { // Unification
  case class TmIsZero(t: Tm) extends Tm
  override def eval1(t: Tm) = t match {
    case _: TmSucc => super[Nat].eval1(t)
    case _: TmPred => super[Nat].eval1(t)
    case _: TmIf => super[Bool].eval1(t)
    case TmIsZero(TmZero) => TmTrue
    case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
    case TmIsZero(t1) => TmIsZero(eval1(t1))
    case _ => throw NoRuleApplies
}}
```

Figure 4. Implementing ARITH with open case classes.

`Nat` and `Bool` as well as complementing rules for the zero test. This turns out to be quite tedious and error-prone: we have to recognize interesting old cases (`TmSucc`, `TmPred` and `TmIf`) using *typecases* and delegate appropriately to either `Nat` or `Bool` via a `super` call. If the programmer forgets delegating any of those cases, then the pattern matching falls into the wildcard pattern (the last case), throwing a `NoRuleApplies` exception. In this situation the semantics of pattern matching and wildcard patterns are to blame: since pattern matching just follows the order of patterns, once we make a `super` call to a definition with wildcards then all cases will be covered. Therefore to workaround this problem we have to carefully delegate the cases one-by-one to the `super` calls. 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.

2.5 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 can be adapted using `PartialFunction` with a few changes, e.g. `eval1` in `Bool`:

```
def eval1: PartialFunction[Tm,Tm] = {
  ... // Cases for TmIf
  case TmTrue => throw NoRuleApplies
  case TmFalse => throw NoRuleApplies }
```

`eval1` is a partial function of type `PartialFunction[Tm,Tm]`. A value of `PartialFunction[Tm,Tm]` is constructed using the anonymous function syntax, where the argument `Tm` is directly pattern matched. The convenience of wildcard patterns is lost: wildcards are replaced by named patterns to avoid shadowing other partial functions to be combined.

Discussion of the Approach. Partial functions make the composition of features work more smoothly, avoiding the problems with `eval1` for the open case classes approach:

```
override def eval1 = super[Nat].eval1 orElse
                     super[Bool].eval1 orElse
                     { ... /* Cases for TmIsZero */ }
```

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

Still, this implementation is not very satisfactory. `orElse` is left-biased, thus the *combination 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). `orElse` gives no warning when composing such overlapped partial functions and 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 from both `f` and `g`, which restricts the reusability of partial functions.

Table 1. Pattern matching support comparison

	Conciseness	Exhaustiveness	Extensibility	Composability
VISITOR	○	●	○	○
Sealed case class	●	●	○	○
Open case class	●	○	●	○
Partial function	●	○	●	◐
CASTOR	●	◐*	●	●

●= good, ◐= neutral, ○= bad

* 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.

2.6 Discussion

So far, we have presented (partial) implementations of ARITH using the VISITOR pattern, sealed case classes, open case classes, partial functions. Unfortunately, none of these implementations fully meets the desirable properties—conciseness, exhaustiveness, extensibility and composability—summarized in Section 1. Using these four properties as criteria, Table 1 compares the pattern matching support of these approaches.

Key Observations. Conciseness and exhaustiveness are somehow conflicting with each other. The support for guards brings conciseness but, at the same time, complicates exhaustiveness checking. A guard might call some function whose execution result is unknown at compile-time, making the

reachability of that `case` clause hard to check. Regarding extensibility and composability, essentially what makes pattern matching hard to be extended or composed is that the `case` clauses are *order-sensitive* and gathered in *one definition*.

We observe that it is useful to distinguish between top-level (shallow) patterns and nested (deep) patterns. Top-level patterns should be order-insensitive and split into multiple definitions so that they can be easily composed. For many functions nested patterns often have a good default. That is the case for the nested patterns in `eval1` as well as other examples illustrated in Section 3. While there are operations for which sometimes nested patterns do not have good defaults, in our personal experience and also the extended case study in Section 5 these operations are not very common in practice. Therefore we propose an approach for nested patterns that offers conciseness for the common case: nested patterns should come with a default so that they would work for variant extensions. We apply this key insight in designing the Castor framework, which turns out to be compared favorably in terms of the four properties among the approaches. An overview of Castor will come next in Section 3.

3 An Overview of Castor

This section presents the Castor framework. We first show how to model the Arith language discussed in Section 2 using Castor and discuss how this implementation meets the desirable properties. We then show how to define various types of operations that pose previously identified challenges in a modular setting. In particular, we show that dependent operations [30], context-sensitive operations [18] and multi-sorted languages [28] can be nicely modeled in Castor.

3.1 Arith with Castor

A nice aspect of the Visitor pattern is that it decentralizes pattern matching into multiple, order-insensitive methods. Taking the best of both worlds, Castor combines open case classes with extensible visitors [15, 26, 27]. To reduce the complexity and verbosity incurred by visitors, Castor employs metaprogramming for generating boilerplate.

Recall the Arith language shown in Figure 1. With Castor, it is possible to model Arith in a concise and modular way, as shown in Figure 5. A Castor component is a trait annotated with `@family`. Component `Term` serves as the root of all terms, where an open data type `Tm` is declared and marked as `@adt`. The signature of the small-step evaluator is specified by the inner trait `Eval1`, where `@default(Tm)` denotes that `Eval1` is an operation on `Tm` with a default behavior. The output type of `Eval1` is declared by setting the abstract type `OTm` as `Tm`. The default behavior is specified via the `otherwise` method, which throws a `NoRuleApplies` exception. `Nat` and `Bool` are independent extensions to `Term` that are defined separately as two Castor components. `Nat`, for example, extends the data type `Tm` with `TmZero`, `TmSucc` and `TmPred`

```
@family trait Term {
  @adt trait Tm // Datatype declaration
  @default(Tm) trait Eval1 { // Signature
    type OTm = Tm
    def otherwise = _ => throw NoRuleApplies
}}
@family trait Nat extends Term {
  // Datatype extension
  @adt trait Tm extends super.Tm {
    def TmZero: Tm
    def TmSucc: Tm => Tm
    def TmPred: Tm => Tm
  }
  def nv(t: Tm): Boolean = ... // Same as in Figure 3
  @default(Tm) trait Eval1 extends super.Eval1 {
    override def tmSucc = t => TmSucc(this(t))
    override def tmPred = {
      case TmZero => TmZero
      case TmSucc(t) if nv(t) => t
      case t => TmPred(this(t))
}}}
@family trait Bool extends Term {
  @adt trait Tm extends super.Tm {
    def TmTrue: Tm
    def TmFalse: Tm
    def TmIf: (Tm,Tm,Tm) => Tm
  }
  @default(Tm) trait Eval1 extends super.Eval1 {
    override def tmIf = {
      case (TmTrue,t2,_) => t2
      case (TmFalse,_,t3) => t3
      case (t1,t2,t3) => TmIf(this(t1),t2,t3)
}}}
@family trait Arith extends Nat with Bool {
  @adt trait Tm extends super[Nat].Tm
               with super[Bool].Tm {
    def TmIsZero: Tm => Tm
  }
  @visit(Tm) trait Eval1 extends super[Nat].Eval1
                        with super[Bool].Eval1 {
    def tmIsZero = {
      case TmZero => TmTrue
      case TmSucc(t) if nv(t) => TmFalse
      case t => TmIsZero(this(t))
}}}
```

Figure 5. Implementing Arith with Castor.

methods. These capitalized methods use function types to express how to construct these variants. As opposed to the case classes counterpart, methods are more compact.

The combination of case classes and visitors provides Castor with flexibility in defining operations over `Tm`. Operations that focus on a fixed subset of variants and have a good default like `nv` are defined as functions. Ordinary operations like `Eval1` are defined as visitors for retaining composability. But unlike normal visitors, nested case analysis is much simplified via (nested) pattern matching rather than auxiliary

visitors. Take the evaluation of a predecessor term for example. When a predecessor is processed by `Eval1`, it will be recognized and dispatched to the `tmPred` method. Then its inner term is pattern matched using several `case` clauses via Scala's anonymous function syntax. As these are `case` clauses, deep patterns and guards can be used. To apply `Eval1` on the inner term, we call `this(t)`. To restore the convenience of wildcards for visitors, Castor provides an annotation `@default`, which provides a default implementation for all *known* variants. By annotating `@default` and inheriting `otherwise` from `Term`'s `Eval1`, we only need to override interesting cases.

Definitions from `Nat` and `Bool` are easily combined in `Arith` through Scala's mixin composition. Datatype definitions are merged with a new constructor for the zero test. With top-level patterns split into different methods, `Eval1` from `Nat` and `Bool` are merged without conflicts. The only thing we need to do is to complement the case for zero test. As the zero test is a case that requires programmer written code, `Arith`'s `Eval1` is annotated with `@visit` rather than `@default`.

Client Code. A Castor component can be directly imported in client code. Here are some tests on `Arith`:

```
import Arith._
val tm = TmIsZero(
  TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
eval1(tm) // TmIsZero(TmPred(TmSucc(TmZero)))
eval1(eval1(tm)) // TmIsZero(TmZero)
eval1(eval1(eval1(tm))) // TmTrue
```

By importing `Arith`, we are able to construct a term using all the variants including those from `Nat` and `Bool`. Castor's visitors are used like normal functions with their lowercase name. The constructed term is evaluated step by step using `eval1` and the result for each step is shown in the comments.

Discussion. Here we discuss how Castor addresses the four desirable 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. Top-level patterns are methods, whose exhaustiveness is checked in the code generation phase by the Scala compiler (see Section 4 for details). For nested patterns using `case` clauses, a default must be provided. However, this is neither statically enforced by Scala nor Castor. Note, however, that with specialized language support it is possible to enforce that nested always provide a default. This is precisely what EADDs [40] 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 (see Section 4 for details).

```
@family @adts(Tm) @ops(Eval1)
trait PrintArith extends Arith {
  @default(Tm) trait PtmTerm {
    type OTm = String
    def otherwise = ptmAppTerm(_)
    override def tmIf = "if " + this(_) +
      " then " + this(_) + " else " + this(_)
  }
  @default(Tm) trait PtmAppTerm {
    type OTm = String
    def otherwise = ptmATerm(_)
    override def tmPred = "pred " + ptmATerm(_)
    override def tmSucc = "succ " + ptmATerm(_)
    override def tmIsZero = "iszero " + ptmATerm(_)
  }
  @default(Tm) trait PtmATerm {
    type OTm = String
    def otherwise = "(" + ptmTerm(_) + ")"
    override def tmZero = "0"
    override def tmTrue = "true"
    override def tmFalse = "false"
}}
```

Figure 6. Pretty printer for Arith.

- **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.

3.2 Pretty Printing: Operations with Dependencies

Operations that depend (or even mutually depend) on other operations pose additional challenges for modularity [30]. Castor fully supports modular dependent operations.

Consider implementing a pretty printer for Arith. To print out parenthesis only when necessary, we classify terms according to whether they are primitives or applications. Figure 6 gives the implementation, which illustrates how to add new (possibly dependent) operations with Castor. The pretty printer consists of three *mutually* dependent visitors: `PtmTerm`, `PtmAppTerm` and `PtmATerm`, each of which focuses on a subset of terms and delegates remaining cases to others. Castor allows operations with complex dependencies to be defined in a natural and modular way. We can directly refer to other visitors in scope via their lowercase name and use them as normal functions. For instance, visitor `PtmTerm` delegates its boring cases to visitor `PtmAppTerm` by supplying the term to `ptmAppTerm`. The magic under the hood is that Castor generates a lowercase `val` declaration for each visitor, which allows the visitor to be used elsewhere. `@adt` and `@ops` are auxiliary annotations that provide information for Castor in generating code. More details will come in Section 4.

3.3 Structural Equality

Structural equality is an operation that checks whether two terms are constructed consistently. We can already compare the structural equality of two terms using == thanks to the `equals` method injected by the `case` keyword. Still, manually encoding structural equality is interesting as it shows how to pattern match on multiple arguments. A Castor implementation of structural equality for Arith is:

```
@family @adts(Tm) @ops(Eval1)
trait EqArith extends Arith {
  @visit(Tm) trait Equal {
    type OTm = Tm => Boolean
    def tmZero = {
      case TmZero => true
      case _ => false }
    def tmSucc = t => {
      case TmSucc(s) => this(t)(s)
      case _ => false }
    ... // Definition for other cases elided
}}
```

`Equal` is a *context-sensitive* visitor [15] whose context is the second term being compared. To capture the context, we instantiate the output type `OTm` as a function type `Tm => Boolean`. Pattern matching on the two terms is done differently: the shape of the first term is revealed by the visit methods whereas the shape of the second term is revealed by `case` clauses. We then recursively compare their subterms if they fall in the same pattern; otherwise, a `false` is returned.

Note that all cases share the same default (`false`). We could use Scala's pattern matching mechanism to avoid duplicating that default by overriding the generated `apply` method:

```
override def apply(t: Tm) = s => {
  try { t.accept(this)(s) }
  catch { case _: MatchError => false }}
```

When two terms are constructed differently, a `MatchError` exception is thrown. The `apply` method catches that exception and returns `false`. However, we prefer to use nested case analysis with a default to stick to the core ideas in Castor.

3.4 Typed Arith

The Arith language presented so far allows erroneous terms like `TmPred(TmTrue)` to be constructed. To rule out erroneous terms, we introduce *types* and a type-checking operation to the Arith language. The introduction of types evolves Arith from an untyped language to a typed language:

```
@family @adts(Tm) @ops(Eval1)
trait TyArith extends Arith {
  @adt trait Ty {
    def TyNat: Ty
    def TyBool: Ty
  }
  @visit(Tm) trait Typeof {
    type OTm = Option[Ty]
    def tmZero = Some(TyNat)
    def tmSucc = t => this(t) match {
      case ty@Some(TyNat) => ty
      case _ => None }
    def tmPred = tmSucc // Case definition reused
    def tmTrue = Some(TyBool)
    def tmFalse = tmTrue // Case definition reused
    def tmIf = (t1,t2,t3) =>
      (this(t1),this(t2),this(t3)) match {
        case (Some(TyBool),o2,o3) if o2==o3 => o2
        case _ => None }
    def tmIsZero = t => this(t) match {
      case Some(TyNat) => Some(TyBool)
      case _ => None }
}}
```

Like `Tm`, `Ty` is a datatype for representing types. Two concrete types, `TyNat` and `TyBool`, are introduced for classifying terms that produces numbers or boolean values. A visitor `Typeof` is defined for type checking 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. One interesting thing to notice is that for variants that share the same signature and behavior, it is sufficient to define the behavior once and reuse the definition for other variants, as illustrated by `tmSucc` and `tmPred`. Such reuse is, however, hard to achieve for `case` clauses because they are not referable.

4 Extensible Visitors and Code Generation

The power of Castor comes from the underlying extensible visitors. The extensible visitor encoding combines ideas from previous work [15, 26, 27, 42] for better supporting pattern matching. Furthermore, Castor employs Scalameta [1, 5], a modern Scala meta-programming library, for generating the boilerplate required by the visitor encoding. This section first presents the encoding by explaining the generated code for Arith shown in Figure 5 and then formalizes the code generation and discusses the limitations of Castor.

4.1 An Encoding of Extensible Visitors

Recall the Castor implementation of Arith shown in Figure 5. Let us first have a look at the generated code for `Term`:

```
trait Term {
  type TmV <: TmVisit
  abstract class 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 otherwise: Tm => OTm }
  trait Eval1 extends TmDefault { _: TmV => ... }
  val eval1: Eval1
}
```

The visitor encoding presented here is slightly different from the one shown in Figure 2. The use of several Scala-specific features requires explanations. Instead of directly using `TmVisit` in declaring the `accept` method, we use `TmV`-an *abstract type* bounded by `TmVisit`. This decouples `Tm` from

a specific visitor interface, allowing covariant refinement on the upper bound of TmV to account for new data variants. The return type of the visit methods is parameterized by an abstract type OTm rather than a type parameter. Hence 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(t) as a shorthand of t.accept(v), where t 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, which extends TmVisit with an otherwise method for specifying the default behavior. Eval1 is a visitor annotated with **@default**, thus extending TmDefault with a self-type annotation. There is a corresponding **val** with lowercase name generated for Eval1, which not only allows the visitor to be used like normal functions but also facilitates exhaustiveness checking, as seen later.

The encoding makes more sense with the following generated code for Nat:

```
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: Tm => OTm
    def tmPred: Tm => OTm }
  trait TmDefault extends TmVisit
                  with super.TmDefault { _: TmV =>
    def tmZero = otherwise(TmZero)
    def tmSucc = t => otherwise(TmSucc(t))
    def tmPred = t => otherwise(TmPred(t)) }
  def nv(t: Tm): Boolean = ...
  trait Eval1 extends TmDefault
              with super.Eval1 { _: TmV => ...  }
}
```

The constructors of Tm are transformed to case classes/objects that extend Tm. To implement the accept method, TmVisit is extended with lowercase visit methods one for each constructor. The upper bound of TmV is refined as the new TmVisit to allow invocations on extended visit methods in implementing accept for new subclasses of Tm. TmDefault is a default visitor that provides an implementation for each visit method. The default implementation reconstructs the term and passes it to the otherwise method. nv remains unchanged while Eval1 is modified by extending TmDefault and annotating itself as TmV. Bool and Arith are transformed in the same way. Their definitions are elided for space reasons.

Companion Object. Besides modifying the trait annotated with **@family**, Castor also automatically generates a companion object for it, e.g. Nat:

$$
\begin{array}{lcl}
Fam & ::= & \texttt{@family @adts}(\overline{D})\ \texttt{@ops}(\overline{V}) \\
 & & \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\{\ \overline{Adt}\ \overline{Vis}\} \\
Adt & ::= & \texttt{@adt trait}\ D\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}D}\{\ \overline{Ctr}\} \\
Vis & ::= & \texttt{@}a(D)\ \texttt{trait}\ V\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}V}\{\ \ldots\} \\
a & ::= & \texttt{default} \mid \texttt{visit} \\
Ctr & ::= & \texttt{def}\ C\texttt{:}D \mid \texttt{def}\ C\texttt{:}(\overline{T})\texttt{=>}D \\
T & ::= & D \mid \texttt{Int} \mid \texttt{Boolean} \mid T\texttt{=>}T
\end{array}
$$

Figure 7. Syntax.

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

Castor tries to bind all the abstract types to their corresponding visitor interfaces. Moreover, the **val** declarations are met by singleton objects that extend traits with capitalized names. Automatically generating companion objects is useful for two reasons. Firstly, it provides *exhaustiveness checking* for concrete visitors. That is, if a visitor does not implement all the visit methods, the object creation fails, with the missing methods being reported to the user. Secondly, the companion objects can be directly imported into client code, simplifying the usage of Castor components.

4.2 Formalized Code Generation

We can see that although the extensible visitor encoding is powerful, directly programming with it is cumbersome. Moreover, the encoding relies on advanced features of Scala, making it less accessible to novice Scala programmers. To reduce the complexity and verbosity of the encoding, Castor employs Scalameta based macro annotations [1, 5] in generating code. With Scalameta, we are able to modify the parsed source program before type-checking takes place.

Figure 7 describes valid Scala programs accepted by Castor. Uppercase meta-variables range over capitalized names. $\overline{A}$ is written as a shorthand for sequence $A_1 \bullet \ldots \bullet A_n$, where $\bullet$ denotes **with**, comma or semicolon according to the context. Figure 8 formalizes the translation. We use semantic brackets ($[\![\cdot]\!]$) in defining the translation rules and angle brackets (<>) for processing sequences. The translation is quite straightforward. One can easily see that processing Term and Nat through Figure 8 generates the code shown previously. Here we briefly discuss some interesting cases. A **trait** is recognized as a *base case* if it extends nothing. Base cases need extra declarations such as **abstract class** for datatypes or **val** declaration for visitors. Non-argument constructors are translated to **case object**s rather than **case class**es.

4.3 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.

$$
\begin{array}{l}
[\![\texttt{@family @adts}(\overline{D})\ \texttt{@ops}(\overline{V})\ \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\{\ \overline{Adt}\ \overline{Vis}\}]\!] = \\
\quad \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\{\ [\![\overline{Adt}]\!]\ [\![\overline{Vis}]\!]\} \\
\quad \texttt{object}\ F\ \texttt{extends}\ F\{ \\
\qquad \langle \texttt{type}\ D\texttt{V} = 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\} \\
[\![\texttt{@adt trait}\ D\{\ \overline{Ctr}\}]\!] = \\
\quad \texttt{type}\ D\texttt{V} <: D\texttt{Visit} \\
\quad \texttt{abstract class}\ D\{\ \texttt{def accept(v:}D\texttt{V): v.0}D\} \\
\quad [\![\overline{Ctr}]\!] \\
\quad \texttt{trait}\ D\texttt{Visit}\{\ \_: D\texttt{V} \Rightarrow \\
\qquad \texttt{type 0}D \\
\qquad \texttt{def apply(x:}D\texttt{) = x.accept(this)} \\
\qquad [\![\overline{Ctr}]\!]_{visit}\} \\
\quad \texttt{trait}\ D\texttt{Default}\ \texttt{extends}\ D\texttt{Visit}\{\ \_: D\texttt{V} \Rightarrow \\
\qquad \texttt{def otherwise:}D \Rightarrow \texttt{0}D \\
\qquad [\![\overline{Ctr}]\!]_{default}\} \\
[\![\texttt{@adt trait}\ D\ \texttt{extends super[}F\texttt{].}D\{\ \overline{Ctr}\}]\!] = \\
\quad \texttt{type}\ D\texttt{V} <: D\texttt{Visit} \\
\quad [\![\overline{Ctr}]\!] \\
\quad \texttt{trait}\ D\texttt{Visit}\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}D\texttt{Visit}} \\
\quad \{\ \_: D\texttt{V} \Rightarrow [\![\overline{Ctr}]\!]_{visit}\} \\
\quad \texttt{trait}\ D\texttt{Default}\ \texttt{extends}\ D\texttt{Visit}\ \texttt{with}\ \overline{\texttt{super[}F\texttt{].}D\texttt{Default}} \\
\quad \{\ \_: D\texttt{V} \Rightarrow [\![\overline{Ctr}]\!]_{default}\} \\
[\![\texttt{def}\ C : D]\!] = \\
\quad \texttt{case object}\ C\ \texttt{extends}\ D\{\ \texttt{def accept(v:}D\texttt{V) = v.}c\} \\
[\![\texttt{def}\ C : (\overline{T}) \Rightarrow D]\!] = \\
\quad \texttt{case class}\ C(\overline{x:T})\ \texttt{extends}\ D\{\ \texttt{def accept(v:}D\texttt{V) = v.}c(\overline{x})\} \\
[\![\texttt{def}\ C : D]\!]_{visit} = \texttt{def}\ c : \texttt{0}D \\
[\![\texttt{def}\ C : (\overline{T}) \Rightarrow D]\!]_{visit} = \texttt{def}\ c : (\overline{T}) \Rightarrow \texttt{0}D \\
[\![\texttt{def}\ C : D]\!]_{default} = \texttt{def}\ c = \texttt{otherwise}(C) \\
[\![\texttt{def}\ C : (\overline{T}) \Rightarrow D]\!]_{default} = \texttt{def}\ c = (\overline{x}) \Rightarrow \texttt{otherwise}(C(\overline{x})) \\
[\![\texttt{@}a(D)\ \texttt{trait}\ V\{\ \ldots\}]\!] = \\
\quad \texttt{trait}\ V\ \texttt{extends}\ DA\{\ \_: D\texttt{V} \Rightarrow \ldots\} \\
\quad \texttt{val}\ v : V \\
[\![\texttt{@}a(D)\ \texttt{trait}\ V\ \texttt{extends}\ \overline{\texttt{super[}F\texttt{].}V}\{\ \ldots\}]\!] = \\
\quad \texttt{trait}\ V\ \texttt{extends}\ DA\ \texttt{with}\ \overline{\texttt{super[}F\texttt{].}V}\{\ \_: D\texttt{V} \Rightarrow \ldots\} \\
[\![\overline{X}]\!] = \langle [\![X]\!] \mid X \in \overline{X} \rangle
\end{array}
$$

Figure 8. Translation.

- **Boilerplate nested composition.** Lacking of parents' information also disallows automatically composing nested members. Assuming that automatic nested composition is available, `Arith` shown in Figure 5 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.

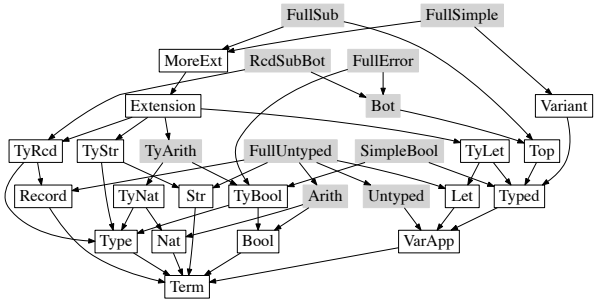

Figure 9. Simplified inheritance graph.

5 Case Study and Evaluation

To give some evidence on the expressivity and effectiveness of Castor, we conduct a case study on TAPL [31]. Examples shown in previous sections are directly from or greatly inspired by the TAPL case study. TAPL are a good benchmark for assessing the capabilities of modular pattern matching and has been adopted by EVF [42]. 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 3, we are able to refactor the non-modular implementation into a modular manner. Our evaluation shows that the refactored version significantly reduces the SLOC. However, at the moment, improved modularity does come at some performance penalty.

5.1 Overview

The Scala implementation of TAPL [3] strictly follows the original OCaml version, which uses sealed case classes and pattern matching. The first 10 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, fix points, error handling, subtyping, etc. Features are shared among these 10 languages. For example, *arith* is included by *fulluntyped*, *fullsimple* and *fullsub*. However, such featuring sharing is achieved via duplicating code, causing problems like:

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**

- **Inconsistent definitions.** Lambdas are printed as `"lambda"` in all languages except *untyped*, which is `"\"`.
- **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 9 gives a simplified high-level overview of the refactored implementation. The candidate languages are represented as gray boxes and extracted features/sub-languages are represented as white boxes. From Figure 9 we can see that the interactions between languages (revealed by the arrows) are quite intense.

5.2 Evaluation

We evaluate Castor by answering the following questions:

- **Q1.** Is Castor effective in reducing SLOC?
- **Q2.** How does Castor compare to EVF [42]?
- **Q3.** How much performance penalty does Castor incur?

Q1. Table 2 reports the SLOC comparison results. With shared features modularly extracted and reused, Castor reduces over *half* of the total SLOC compared to the non-modular implementation written in plain Scala.

Q2. Table 2 also compares Castor with EVF [42], a modular visitor framework in Java. Like Castor, EVF automatically generates boilerplate code associated with visitors. However, better support for pattern matching and more concise syntax for Castor result in around 25% SLOC reduction with respect to EVF. Moreover, EVF requires manual instantiation from Java interfaces to classes for creating objects. The instantiation burden can be quite heavy for feature-rich languages. Castor completely removes the burden of instantiation by generating companion objects automatically.

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 [2] microbenchmark framework is used for performance measurements. The benchmark programs are compiled using Scala 2.12.3 and executed on a MacBook Pro with 2.6 GHz Intel Core i5 with 8 GB memory. Figure 10 compares the execution time in

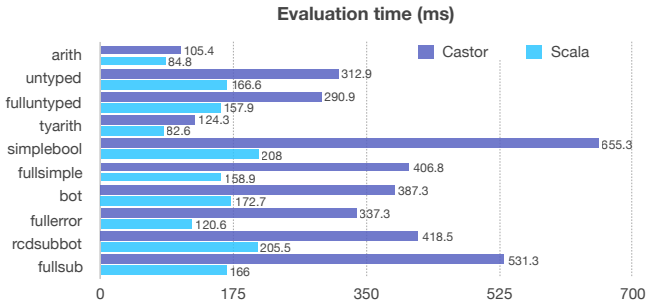

Figure 10. Performance evaluation of TAPL interpreters.

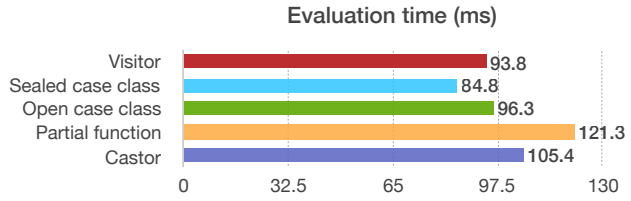

Figure 11. Performance evaluation of Arith.

milliseconds. From the figure we can see that Castor implementations have a 1.24x (*arith*) to 3.2x (*fullsub*) slowdown with respect to the corresponding non-modular Scala implementations. Figure 11 further compares the performance of the Arith implementations discussed in Section 2. Without surprise, modular implementations are slower than non-modular implementations. With underlying optimizations, implementations based on case classes are faster than implementations based on visitors. The implementation in partial function is worst due to the heavy use of exception handling.

Discussion. 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. Another source of performance 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 [33].

6 Related Work

Polymorphic Variants. OCaml supports polymorphic variants [11]. Unlike traditional variants, polymorphic variant constructors are defined individually and are not tied to a particular datatype. Garrigue [12] presents a solution to the EP 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 Data Types and Open Functions. To solve the EP, Löh and Hinze [22] propose to extend Haskell with open datatypes and open functions. Different from classic closed datatypes and closed functions, the open variants 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: the source for all files with variants of a datatype must be available for generating code.

Data Types à la Carte (DTC). DTC [37] encodes composable datatypes using existing features of Haskell. The idea is to express extensible datatypes as a fixpoint of co-products of functors. While it is possible to define operations that have dependencies and/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 [4] extend DTC with GADTs [39] and automatically generates boilerplate using Template Haskell [35]. Oliveira et al. [29] use list-of-functors instead of co-products to better simulate OOP features subtyping, inheritance and overriding.

Modular Church-Encoded Interpreters. Solutions to the EP based on Church encodings can also be used for developing modular interpreters. Well-known techniques are finally tagless [6], Object Algebras [28] and Polymorphic Embedding [16]. However, these techniques do not support pattern matching and/or dependencies, making it hard to define operations like small-step semantics discussed in Section 2. Typical workarounds are defining the operation together with the dependencies or use advanced features like intersection types and a merge operator [30, 32]. In contrast, Castor allows us to implement operations that need nested patterns and/or with dependencies in a simple, modular way.

Extractors. Extractors [9] are an alternative pattern matching mechanism in Scala. Extractors are `object`s with a user-defined `unapply` method that specifies how to tear down an object. Compared to case classes, extractors are flexible, independent of classes but verbose. Emir et al. [9] further compare case classes and extractors with other four OO pattern matching techniques according to conciseness, maintainability and performance. Their results show that case classes and extractors are complementary in terms of these criteria. Two criteria that we consider important—*exhaustiveness* and *composability*—are not addressed. Like open case classes, extractors do not meet these two properties as well.

Other Approaches to Pattern Matching in OOP. There are many attempts to introduce pattern matching into mainstream OOP languages like Java [14, 21] and C++ [36]. Some new OOP languages are designed with first-class pattern matching such as Newspeak [13], Fortress [34] or Grace [17]. Yet, how to cooperate pattern matching with the open nature of OOP class hierarchy while preserving the desirable properties summarized in Section 1 is still challenging.

Extensible Visitors. Section 4 presents an extensible visitor encoding for modular pattern matching. The basis of the encoding comes from [26] and the use of case classes and the way to express dependencies are inspired by [15, 27]. A common problem of these encodings is that although powerful, they are too complicated for practical use. In contrast, Castor simplifies the encoding through meta-programming. Similar idea has been adopted in EVF [42] which generates boilerplate associated with visitors through Java annotation processor. Unlike Castor, nested pattern matching in EVF is simulated via constructing anonymous visitors with fluent setters and Java 8's lambdas [41]. Consequently, deep patterns and guards are not supported and the notation overhead is much higher than Scala's `case` clauses. EVF also generates various traversal templates for eliminating boilerplate in querying and transforming AST structures. Essentially, these traversal templates can be ported to Castor by generating code and providing annotations similar to `@default`.

7 Conclusion and Future Work

This work argues that pattern matching in an extensible setting would benefit from an unordered semantics for (top-level) patterns. In essence, extensibility and composability do not interact well with ordered patterns. To accommodate for the convenience of nested patterns, we propose a second case analysis mechanism with defaults. We argue that such nested patterns often have good default for most operations in practice. This is partly validated by our case study, where practically all operations that used nested case analysis had good defaults for such nested patterns. We applied this idea in designing Castor, a Scala meta-programming framework that allows programmers to write concise, exhaustive, extensible and composable pattern matching code. While Castor is practical and serves the purpose of demonstrating our points regarding pattern matching, 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.

A worthwhile path for future work would be to study a more principled language design that builds on the ideas of this work. Another path would be to support GADTs [39]. There is some initial support for GADTs that allows simple, typed, embedded DSLs to be modeled with Castor. For space reasons, we do not discuss such support in this paper. We would like continue the investigation and see if more realistic embedded DSLs can be modeled with Castor.

References

[1] [n. d.]. Scalameta. http://scalameta.org/. ([n. d.]).

[2] [n. d.]. ScalaMeter. http://scalameter.github.io/. ([n. d.]).

[3] [n. d.]. TAPL Scala. https://github.com/ilya-klyuchnikov/tapl-scala/. ([n. d.]).

[4] Patrick Bahr and Tom Hvitved. 2011. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*. ACM, 83–94.

[5] Eugene Burmako. 2017. *Unification of Compile-Time and Runtime Metaprogramming in Scala*. Ph.D. Dissertation. EPFL.

[6] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[7] Craig Chambers. 1992. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming*.

[8] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM Sigplan Notices*, Vol. 35. ACM, 130–145.

[9] Burak Emir, Martin Odersky, and John Williams. 2007. Matching objects with patterns. In *European Conference on Object-Oriented Programming*.

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

[11] Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*.

[12] Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*.

[13] Felix Geller, Robert Hirschfeld, and Gilad Bracha. 2010. *Pattern Matching for an object-oriented and dynamically typed programming language*. Number 36. Universitätsverlag Potsdam.

[14] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. 2008. Matchete: Paths through the pattern matching jungle. In *International Symposium on Practical Aspects of Declarative Languages*.

[15] Christian Hofer and Klaus Ostermann. 2010. Modular Domain-specific Language Components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*.

[16] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of Dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '08)*.

[17] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns As Objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. New York, NY, USA, 17–28.

[18] Pablo Inostroza and Tijs van der Storm. 2015. Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*.

[19] Chinawat Isradisaikul and Andrew C. Myers. 2013. Reconciling Exhaustive Pattern Matching with Objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*.

[20] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[21] Jed Liu and Andrew C Myers. 2003. JMatch: Iterable abstract pattern matching for Java. In *PADL*.

[22] Andres Löh and Ralf Hinze. 2006. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*.

[23] Todd Millstein, Colin Bleckner, and Craig Chambers. 2004. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. *ACM Trans. Program. Lang. Syst.* 26, 5 (Sept. 2004).

[24] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. 1997. The Definition of Standard ML-Revised. (1997).

[25] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.

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

[27] Bruno C. d. S. Oliveira. 2009. Modular Visitor Components. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*.

[28] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*.

[29] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular Reifiable Matching: A List-of-functors Approach to Two-level Types. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*.

[30] Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming*.

[31] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[32] Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*.

[33] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *In GPCE*.

[34] Sukyoung Ryu, Changhee Park, and Guy L Steele Jr. 2010. Adding pattern matching to existing object-oriented languages. In *ACM SIGPLAN Foundations of Object-Oriented Languages Workshop*.

[35] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*.

[36] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. 2013. Open Pattern Matching for C++. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences (GPCE '13)*.

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

[38] Philip Wadler. 1998. The Expression Problem. Email. (Nov. 1998). Discussion on the Java Genericity mailing list.

[39] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*.

[40] Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*.

[41] Weixin Zhang. 2017. Extensible domain-specific languages in object-oriented programming. *HKU Theses Online (HKUTO)* (2017).

[42] Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *European Conference on Object-Oriented Programming*.