Pattern Matching in an Open World

ANONYMOUS AUTHOR(S)

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 metaprogramming 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”.

Additional Key Words and Phrases: Dummy keyword – please provide 1–5 keywords

ACM Reference Format:

1 INTRODUCTION

Pattern matching is a pervasive and useful feature in functional programming. Languages such as Haskell [Jones 2003] or ML [Milner et al. 1997] 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, operations used in 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 Object-Oriented Programming (OOP) in the past. The Visitor pattern [Gamma et al. 1994] is frequently used as poor man’s approach to pattern matching in object-oriented 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 [Hofer and Ostermann 2010; Odersky and Zenger 2005; Oliveira 2009; Zhang and Oliveira 2017] 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 [Odersky et al. 2004] 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 [Clifton et al. 2000] and multi-methods [Chambers 1992] 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 [Isradisaikul and Myers 2013; Millstein et al. 2004] 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. In particular we argue that a formulation for OOP pattern matching should have the following properties:

- **Conciseness.** Patterns should be described in a concise way with potential support for wildcards, deep patterns and guards.
- **Totality.** Patterns should be total to avoid runtime matching failure. The totality of patterns should be statically verified by the compiler. If the patterns are incomplete, programmers should be warned with the missing cases.
- **Extensibility.** Data types should be extensible in the sense that new data variants are added while existing operations are reused without modification.
- **Composability.** Patterns should be composable so that complex patterns are built from smaller pieces. When composing overlapped patterns, programmers should be warned about possible redundancies.

We show that many approaches that are widely used in practice 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. Pattern matching in functional languages such as ML and Haskell works this way. However when the set of variants is not predefined a priori then relying or 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 totality 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 [2001]'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 total. 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 totality. 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) [Pierce 2002].

Our main goal in this paper is to argue for a different perspective for OO pattern matching. In particular the idea of unordered patterns is brought forward as an approach that plays along well with extensibility. The secondary outcome of this work is the CASTOR library. While CASTOR is practical and serves the purpose of demonstrating our points regarding pattern matching, there are important drawbacks on such a library-based approach. Perhaps most importantly, the metaprogramming approach has consequences in terms of error reporting, since sometimes errors are reported in terms of generated code. Another obvious drawback is that syntax and typing of Scala cannot be changed. As a consequence, certain restrictions on typing cannot be enforced, and the syntax is not as good as a new language design. A worthwhile path for future work would be to study a more principled language design that builds the ideas of this work into a new language design.

**Contributions.** 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 using these properties (Section 2).
- **The CASTOR metaprogramming library:** We improve an encoding of extensible visitors and build a framework called CASTOR on top of it (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).

The full source code for CASTOR and case study is available at\(^2\):

\[...\]

\(^1\)CASTOR stands for CASe class visiTOR

\(^2\)Note to reviewers: The URL is hidden for anonymity. Please check the supplementary material!

\[2\]EVALUATING EXISTING APPROACHES TO PATTERN MATCHING IN SCALA

In this section we review existing approaches to pattern matching in Scala. To facilitate our discussion, we use a running example. The running example comes from TAPL [Pierce 2002] - the very first, untyped, arithmetic language called ARITH. The goal is to model the syntax and semantics
of \textsc{Arith} in a concise and modular manner. None of the existing Scala approaches, including the \textsc{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.

### 2.1 Running Example: \textsc{Arith}

The syntax and semantics of \textsc{Arith} are formalized in Figure 1. \textsc{Arith} has the following syntactic forms: constant zero, successor and predecessor, constant true, constant false, conditional and zero test. The definition \texttt{nv} identifies 0 and successive application of \texttt{succ} to 0 as numeric values. The operational semantics of \textsc{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, reduction rules \texttt{PredZero}, \texttt{PredSucc} and \texttt{Pred} are all defined on a predecessor term. How a predecessor (\texttt{pred t}) is going to be evaluated in the next step is determined by the shape of its inner term \texttt{t}. If \texttt{t} is a successor application to a numeric value, then \texttt{PredSucc} will be applied, etc.

\textsc{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 \textit{nested case analysis} on terms; 2) \textsc{Arith} is a unification of two sub-languages, \textsc{Nat} (zero, successor and predecessor) and \textsc{Bool} (true, false, and conditional) with an extension (zero test). An ideal implementation of \textsc{Arith} is to have \textsc{Nat} and \textsc{Bool} \textit{separately defined} and \textit{modularly reused}. 

\begin{figure}
\centering
\begin{align*}
t & ::= \quad \text{terms:} \\
0 & \quad \text{constant zero} \\
\text{succ } t & \quad \text{successor} \\
\text{pred } t & \quad \text{predecessor} \\
\text{true} & \quad \text{constant true} \\
\text{false} & \quad \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \quad \text{conditional} \\
iszero t & \quad \text{zero test}
\end{align*}
\end{figure}

\begin{figure}
\centering
\begin{align*}
nv & ::= \quad \text{numeric values:} \\
0 & \quad \text{zero value} \\
\text{succ } nv & \quad \text{successor value}
\end{align*}
\end{figure}

Fig. 1. The syntax and semantics of \textsc{Arith}.

\begin{align*}
t_1 & \rightarrow t_1' \quad \text{Succ} \\
\text{succ } t_1 & \rightarrow \text{succ } t_1' \\
pred 0 & \rightarrow 0 \quad \text{PredZero} \\
pred t_1 & \rightarrow \text{pred } t_1' \quad \text{Pred} \\
\text{if } \text{true} \text{ then } t_2 \text{ else } t_3 & \rightarrow t_2 \quad \text{IfTrue} \\
\text{if } \text{false} \text{ then } t_2 \text{ else } t_3 & \rightarrow t_3 \quad \text{IfFalse} \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \rightarrow t_1' \quad \text{If} \\
iszero 0 & \rightarrow \text{true} \quad \text{IszeroZero} \\
iszero (\text{succ } nv_1) & \rightarrow \text{false} \quad \text{IszeroSucc} \\
iszero t_1 & \rightarrow \text{iszero } t_1' \quad \text{Iszero}
\end{align*}
2.2 The Visitor Pattern

The Visitor pattern [Gamma et al. 1994] is often used in mainstream OOP languages for separating operations from the data structure being traversed. We start with an implementation of the Nat sublanguage using the Visitor pattern, which is given in Figure 2. A class hierarchy is defined for modeling the abstract syntax of Nat, where the abstract class Tm represents the data type 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 the so-called visit methods declared in the visitor interface TmVisit.

Operations over Tm are defined as concrete implementations of TmVisit. As an example, object nv instantiates the type parameter T 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 and tmSucc for eval1 is easy by throwing an exception and calling eval1 on the inner term respectively. On the other hand, defining tmPred is tricky! Take the PredSucc rule for example, which cancels a pair of predecessor and successor application to a numeric value. A visitor recognizes only one level representation of a term, which is not sufficient 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.

A problem arises when we try to reuse the Nat implementation above in implementing Arith. The Visitor pattern suffers from the Expression Problem (EP) [Wadler 1998]: 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
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
    case TmPred(t1) => TmPred(eval1(t1))
    case _ => throw NoRuleApplies
  }

Fig. 3. Implementing Nat with sealed case class.

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 to be written for boring cases (e.g. cases that return false in nv). Nevertheless, one nice aspect of the Visitor implementation is that totality 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 in a concise manner. As a programming language that unifies functional and object-oriented paradigms, Scala [Odersky et al. 2004] supports first-class pattern matching via case classes/extractors [Emir et al. 2007]. 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 basically 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 totality 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 totality is the...
inability to add new variants of $\text{Tm}$ 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 the exhaustiveness checking for the ability to add new variants in separate files. Combined with the design pattern proposed by Zenger and Odersky [2001], it is possible to implement $\text{Arith}$ 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 $\text{Arith}$ using open case classes. Note that $\text{Tm}$ is declared as a top-level open data type.

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

Unfortunately, $\text{eval1}$ is more problematic. In the general case defining $\text{Arith}$ in terms of $\text{Nat}$ and $\text{Bool}$ causes problems as Zenger and Odersky [2001]'s approach does not work well for such non-linear extensions. $\text{eval1}$ needs to be overridden for combining definitions from $\text{Nat}$ and $\text{Bool}$ 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 ($\text{TmSucc}$, $\text{TmPred}$, and $\text{TmIf}$) using `typecases` and delegate appropriately to either $\text{Nat}$ or $\text{Bool}$ 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 $\text{Nat}$ and $\text{Bool}$, one may turn to Scala’s `PartialFunction`. `PartialFunction` provides an `orElse` method for composing partial functions, which tries the composed partial functions sequentially until a `MatchError` is raised. Figure 5 adapts the open case class counterpart into a version using `PartialFunction`, with the differences highlighted in gray. Notice that $\text{eval1}$ now returns a partial function of type `PartialFunction[\text{Tm}, \text{Tm}]`. A value of `PartialFunction[\text{Tm}, \text{Tm}]` is constructed using the anonymous function syntax, where the argument $\text{Tm}$ is directly pattern matched.

**Discussion of the Approach.** One nice thing about this approach is that the composition of features now works more smoothly, avoiding the problems with $\text{eval1}$ for the open case classes approach. Composing $\text{eval1}$ in $\text{Arith}$ becomes easier, by chaining $\text{eval1}$ from $\text{Nat}$ and $\text{Bool}$ as well as a new partial function for zero test using the `orElse` combinator. However, the convenience of wildcard patterns is lost: wildcards are replaced by named patterns to avoid shadowing other partial functions to be combined.
abstract class Tm
// Nat extension
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 = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  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
    case TmPred(t1) => TmPred(eval1(t1))
    case _ => throw NoRuleApplies
  }
}

// Bool extension
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
  }
}

// Unification of two extensions
trait Arith extends Nat with Bool {
  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
  }
}

Fig. 4. Implementing ARITH with open case classes.

abstract class Tm
// Nat extension
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 = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  def eval1(t: Tm): PartialFunction[Tm, Tm] = {
    case TmSucc(t1) => TmSucc(eval1(t1))
    case TmPred(TmZero) => TmZero
    case TmPred(TmSucc(t1)) if nv(t1) => t1
    case TmPred(t1) => TmPred(eval1(t1))
    case TmZero => throw NoRuleApplies
  }
}

// Bool extension
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): PartialFunction[Tm, Tm] = {
    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
  }
}

// Unification of two extensions
trait Arith extends Nat with Bool {
  case class TmIsZero(t: Tm) extends Tm
  override def eval1(t: Tm) = 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))
    case _ => throw NoRuleApplies
  }
}

Fig. 5. Implementing ARITH with partial function.
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.

### 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, totality, extensibility and composability—summarized in Section 1. Using these four properties as criteria, Table 1 compares the pattern matching support of these approaches.

<table>
<thead>
<tr>
<th></th>
<th>Conciseness</th>
<th>Totality</th>
<th>Extensibility</th>
<th>Composability</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>Visitor</code> pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Closed case classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Open case classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Partial functions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>CASTOR</strong></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 1. Pattern matching support comparison: ●- good ○- neutral ●- bad.

* CASTOR’s only gets half score on totality 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 totality.

Key Observations. Conciseness and totality are somehow conflicting with each other. The support for guards brings conciseness but, at the same time, complicates totality 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 `eval` 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

In this section, we present the `CASTOR` framework. We first show how to use `CASTOR` to model the `Arith` language discussed in Section 2 and discuss how this implementation addresses the...
Fig. 6. Implementing Arith with CASTOR.

four desirable properties summarized in Section 1. 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 [Oliveira et al. 2013], context-sensitive operations [Inostroza and Storm 2015] and multi-sorted languages [Oliveira and Cook 2012] can be nicely modeled in CASTOR.

3.1 Arith with CASTOR

One nice aspect of the VISITOR pattern is that it decentralizes pattern matching into different methods which are order-insensitive. Taking the best of both worlds, CASTOR combines open case classes with extensible visitors [Hofer and Ostermann 2010; Odersky and Zenger 2005; Oliveira 2009]. To reduce the complexity and verbosity incurred by visitors, CASTOR employs metaprogramming for generating boilerplate code automatically.
Recall the \texttt{ARITH} language shown in Figure 1. With \textsc{Castor}, it is possible to model \texttt{ARITH} in a concise and modular way, as shown in Figure 6. A \textsc{Castor} component is a trait annotated with \@family. Component \texttt{Term} serves as the root of all terms, where an open data type \texttt{Tm} is declared and marked as \@adt. The signature of the small-step evaluator is specified by the inner trait \texttt{Eval1}, where \@default\texttt{(Tm)} denotes that \texttt{Eval1} is an operation on \texttt{Tm} with a default behavior. The output type of \texttt{Eval1} is declared by setting the abstract type \texttt{OTm} as \texttt{Tm}. The default behavior is specified via the otherwise method, which throws a \texttt{NoRuleApplies} exception. \texttt{Nat} and \texttt{Bool} are independent extensions to \texttt{Tm} that are defined separately as two \textsc{Castor} components. \texttt{Nat}, for example, extends the data type \texttt{Tm} with \texttt{TmZero}, \texttt{TmSucc} and \texttt{TmPred} 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 \textsc{Castor} with flexibility in defining operations over \texttt{Tm}. Operations that focus on a fixed subset of variants and have a good default like \texttt{nv} are defined as functions. Ordinary operations like \texttt{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 \texttt{Eval1}, it will be recognized and dispatched to the \texttt{tmPred} method. Then its inner term is pattern matched using several \texttt{case} clauses via Scala’s anonymous function syntax. As these are \texttt{case} clauses, deep patterns and guards can be used. To apply \texttt{Eval1} on the inner term, we call \texttt{this(t)}. To restore the convenience of wildcards for visitors, \textsc{Castor} provides an annotation \@default, which provides a default implementation for all known variants. By annotating \@default and inheriting otherwise from \texttt{Tm}'s \texttt{Eval1}, we only need to override interesting cases.

Definitions from \texttt{Nat} and \texttt{Bool} are easily combined in \texttt{Arith} through Scala’s mixin composition. Data type definitions are merged with a new constructor for the zero test. With top-level patterns split into different methods, \texttt{Eval1} from \texttt{Nat} and \texttt{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, \texttt{Arith}'s \texttt{Eval1} is annotated with \@visit rather than \@default for facilitating totality checking.

\textit{Client Code.} A \textsc{Castor} component can be directly imported in client code. Some tests on the \texttt{Arith} implementation written in \textsc{Castor} are:

\begin{verbatim}
import Arith_
val term = TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
println(eval1(term)) // TmIsZero(TmPred(TmSucc(TmZero)))
println(eval1(eval1(term))) // TmIsZero(TmZero)
println(eval1(eval1(eval1(term)))) // TmTrue
\end{verbatim}

By importing \texttt{Arith}, we are able to construct a term using all the variants including those from \texttt{Nat} and \texttt{Bool}. \textsc{Castor}'s visitors can be used just like normal functions with their lowercase name. Here we use \texttt{eval1} to evaluate the term step by step and the result for each step of evaluation is shown as a comment to the right.

\textit{Discussion.} Here we discuss how \textsc{Castor} addresses the four desirable properties:

- \textbf{Conciseness} By employing Scala’s concise syntax and metaprogramming, \textsc{Castor} greatly simplifies the definition and usage of visitors. In particular, the need of auxiliary visitors for performing deep case analysis is now replaced by pattern matching via \texttt{case} clauses. The concept of visitors is even made transparent to the end user, making the framework more user-friendly.
@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"
  }
}

Fig. 7. Pretty printer of Arith.

- **Totality** The totality of patterns in CASTOR consists of two parts. Top-level patterns are methods, whose totality 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 cannot be statically enforced by Scala, and thus it is not enforced by CASTOR either. That is why we mark totality as a half circle for CASTOR in Table 1. Note, however, that with specialized language support it is possible to enforce that nested always provide a default. This is precisely what Zenger and Odersky [2001] proposal for “Extensible Algebraic Datatypes with Defaults” does.

- **Extensibility** As illustrated by Nat, Bool and Arith, we can extend the data type with new variants and operations, modularly. Such extensibility is enabled by the underlying extensible visitor encoding (see Section 4 for details).

- **Composability** Composability of CASTOR is achieved with the help of 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 in CASTOR

Operations that depend (or even mutually depend) on other operations pose additional challenges for modularity [Oliveira et al. 2013]. However, CASTOR fully supports modular operations with dependencies.
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 7 gives the implementation. 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 another visitor. The dependencies between these three visitors are visualized on the right hand side of Figure 7. 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 like 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 be discussed later in Section 4.

The following code snippet demonstrates the pretty printer:

```scala
import PrintArith._
println(ptmTerm(TmSucc(TmZero))) // "succ 0"
println(ptmTerm(TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero)))))))
// "iszero (if false then true else pred (succ 0))"
```

### 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 case class instances using == thanks to the equals method injected by the case keyword. Still, encoding structural equality manually is interesting because it shows how to pattern match on multiple arguments with Castor.

Castor’s implementation of structural equality for Arith is:

```scala
@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
    }
    def tmPred = t => {
      case TmPred(s) => this(t)(s)
      case _ => false
    }
    def tmTrue = {
      case TmTrue => true
      case _ => false
    }
    def tmFalse = {
      case TmFalse => true
      case _ => false
    }
    def tmIf = (t1,t2,t3) => {
      case TmIf(s1,s2,s3) => this(t1)(s1) && this(t2)(s2) && this(t3)(s3)
    }
  }
}
```


```scala
case _ => false
}
def tmIsZero = t => {
  case TmIsZero(s) => this(t)(s)
  case _ => false
}
```

Equal is a **context-sensitive** visitor [Hofer and Ostermann 2010] whose context is the second term being compared. To capture the context, we instantiate the output type OTm of Equal using a function type Tm => Boolean. Pattern matching on the two terms are done differently: the shape of the first term is revealed by the visit methods while 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 we could further make use of Scala’s pattern matching mechanism for avoiding a case _ => false clause per case:

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

In essence Scala allows overriding the apply method generated by CASTOR. The overridden apply method catches the MatchError exception raised by visit methods for the case that two terms are constructed differently and returns a false. However we prefer to show the longer code using nested case analysis with a default to stick to the core ideas in CASTOR.

Finally, below are some examples that illustrate our implementation:

```scala
import EqArith._
println(equal(TmSucc(TmZero))(TmZero)) // false
println(equal(TmSucc(TmZero))(TmSucc(TmZero))) // true
```

### 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. Our implementation looks like this:

```scala
@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
    def tmTrue = Some(TyBool)
    def tmFalse = tmTrue
  }
```

def tmIf = (t1,t2,t3) => (this(t1),this(t2),this(t3)) match {
  case (Some(TyBool),ty2,ty3) if ty2 == ty3 => ty2
  case _ => None
}
def tmIsZero = t => this(t) match {
  case Some(TyNat) => Some(TyBool)
  case _ => None
}
}

Like Tm, Ty is a data type 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], meaning 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 (a.k.a disjunctive patterns), it is sufficient to define the behavior once and reuse the definition for other variants, as illustrated by tmSucc and tmPred. This kind of reuse is, however, hard to achieve for case clauses because they are not referable.

Here are some tests that illustrate TyArith:

```scala
import TyArith._
println(typeof(TmPred(TmTrue))) // None
println(typeof(TmPred(TmZero))) // Some(TyNat)
```

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 [Hofer and Ostermann 2010; Odersky and Zenger 2005; Oliveira 2009; Zhang and Oliveira 2017] with a focus on better support for pattern matching. Furthermore, CASTOR employs Scalameta3, a modern metaprogramming library for Scala, for generating boilerplate code required by the visitor encoding. In this section we first present the encoding by explaining the generated code for the CASTOR implementation of ARITH shown in Figure 6 and then formalize the code generation and discuss the limitations of CASTOR.

4.1 An Encoding of Extensible Visitors

Recall the CASTOR implementation of ARITH shown in Figure 6. Let us first have a look at the generated code for Term:

```scala
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 => ... /* Unchanged */ }
}
```

3http://scalameta.org/
The visitor encoding presented here is slightly different from the one shown in Figure 2. It turns Odersky and Zenger [2005]'s imperative external visitor encoding into functional [Oliveira 2009]. It uses several Scala-specific features that require some 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 so that we can call v(t) as a shorthand of t.accept(v), where t and v are instances of Tm and TmVisit respectively. In order to pass this as an argument of accept in the implementation of apply, we state that TmVisit is of type TmV using a self-type annotation (_: TmV =>). 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. A corresponding val with lowercase name for Eval1 is generated. This val declaration not only allows the visitor to be used like normal functions inside Term but also facilitates totality checking, as we shall see later.

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

```scala
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 = ... // Unchanged

trait Eval1 extends TmDefault with super.Eval1 { _: TmV => ... /* Unchanged */ }

The constructors of Tm are transformed to case classes/objects that extend Tm [Hofer and Ostermann 2010; Oliveira 2009]. 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.
The document contains a discussion on the syntax and code generation aspects of Pattern Matching in an Open World. It explains the use of Scala's @family and @adts annotations to generate visitor interfaces and datatypes. The text describes how Castor, a tool that automates the generation of visitor interfaces, tries to fix all abstract types to their corresponding visitor interfaces. It also mentions the limitations of Castor, such as the inability to access annotations from parents, which results in unnecessary annotations.

4.2 Formalized Code Generation

We can see that although the extensible visitor encoding is powerful, directly programming with it is very cumbersome. Moreover, the encoding relies on some 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 [Burmako 2017] for generating code. With Scalameta, we are able to modify the parsed source program before type-checking takes place.

Figure 8 describes valid Scala programs that Castor accepts. The uppercase meta-variable ranges over capitalized names. We write $A$ as a shorthand for sequence $A_1 \bullet \ldots \bullet A_n$, where the delimiter $\bullet$ can be with, comma or newline depending on the context. Figure 9 formalizes the code generation. We use semantic brackets ($[\cdot]$) in defining the translation rules and angle brackets notation (<>) for processing a sequence. The translation is quite straightforward that. One can easily see that processing Term and Nat through Figure 6 generates the code shown previously. Here we only briefly discuss some interesting cases. We recognize the base case by checking that it extends nothing. Base cases need some more declarations such as abstract class for datatypes or val declaration for visitors. We also distinguish non-generic, non-argument constructors from others which is translated to a case object rather than a case class.

4.3 Limitations

Castor has some limitations due to the use of meta-programming and the restrictions of 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 nested composition. Lacking of parents’ information also prevents us from automatically composing nested members. Assuming that automatic nested composition is available, Arith shown in Figure 6 can be simplified as:

```scala
@family trait Arith extends Nat with Bool {
  @adts(D) @ops(V) trait F extends F{T[Datatype]Visitor} =
  trait F extends F{T[Datatype]Visitor}
  object F extends F{
    (type DV = DVisitor | D ∈ D ∪ Datatype)
    (object v extends V | V ∈ V ∪ Visitor)
  }
  @adt trait D[X]{Ctr} =
  type DV <: DVisitor
  abstract class D[X]{def accept(v:DV): v.0D[X]}
  [Ctr]
  trait DVisit{_:DV =>
    type 0D[X]
    def apply[X](x:D) = x.accept(this)
  [Ctr]visit
  }
  trait DDefault extends DVisit{_:DV =>
    def otherwise[X]:D[X] => 0D[X]
  [Ctr]default
  }
  @adt trait D[X] extends super[F].D[X]{Ctr} =
  type DV <: DVisitor
  [Ctr]
  trait DVisit extends super[F].DVisit[X]{_:DV => [Ctr]visit}
  trait DDefault extends DVisit with super[F].Default{_:DV => [Ctr]default}
  def C:D = case object C extends D{def accept(v:DV) = v.c}
  def C[X]:(T) => D[X] = case class C[X](x:T) extends D[X]{def accept(v:DV) = v.c(x)}
  def C[X]:(T) => D[X]_visit = def c[X]:(T) => 0D[X]
  def C[X]:(T) => D[X]_default = def c[X] = (x) => otherwise(C(x))
  @a(D) trait V{...} =
    trait V extends DA{_:DV =>...}
    val v : V
  @a(D) trait V extends super[F].V{...} =
    trait V extends DA with super[F].V{_:DV =>...}
  [A] = {A | A ∈ A}
}
```

Fig. 9. Translation.
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.

5 CASE STUDY AND EVALUATION

To give some evidence on the expressivity and effectiveness of CASTOR, we conduct a case study on TAPL [Pierce 2002]. Examples shown in previous sections are directly from or greatly inspired by the TAPL case study. TAPL is a good benchmark for assessing the capabilities of modularization and has been adopted by Zhang and Oliveira [2017]. The reason is that the original implementation duplicates code for features that could be shared. With CASTOR, 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

**Non-modular Implementation.** We took a Scala implementation of TAPL available online⁴ that strictly follows the original OCaml implementation, and hence it is non-modular. We choose the first 10 TAPL languages, namely `arith`, `untyped`, `fulluntyped`, `tyarith`, `simplebool`, `fullsimple`, `fullerror`, `bot`, `rcdsubbot` and `fullsub`, as candidates for refactoring. Each language implementation consists of 4 files: `parser`, `syntax`, `core` and `demo`. These languages cover a lot features including arithmetic, lambda calculus, records, fix points, error handling, subtyping, etc. Features are shared among these

---

⁴ [https://github.com/ilya-klyuchnikov/tapl-scala](https://github.com/ilya-klyuchnikov/tapl-scala)


<table>
<thead>
<tr>
<th>Component</th>
<th>CASTOR</th>
<th>EVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>71</td>
<td>98</td>
</tr>
<tr>
<td>extension</td>
<td>24</td>
<td>34</td>
</tr>
<tr>
<td>floatstring</td>
<td>42</td>
<td>104</td>
</tr>
<tr>
<td>let</td>
<td>48</td>
<td>47</td>
</tr>
<tr>
<td>moreextension</td>
<td>112</td>
<td>106</td>
</tr>
<tr>
<td>nat</td>
<td>85</td>
<td>103</td>
</tr>
<tr>
<td>record</td>
<td>117</td>
<td>198</td>
</tr>
<tr>
<td>top</td>
<td>79</td>
<td>86</td>
</tr>
<tr>
<td>typed</td>
<td>82</td>
<td>138</td>
</tr>
<tr>
<td>varapp</td>
<td>40</td>
<td>65</td>
</tr>
<tr>
<td>variant</td>
<td>136</td>
<td>161</td>
</tr>
<tr>
<td>misc</td>
<td>212</td>
<td>172</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1402</strong></td>
<td><strong>1906</strong></td>
</tr>
</tbody>
</table>

Table 2. SLOC evaluation: CASTOR, EVF and Scala.

10 languages. For example, arith is a sublanguage for fulluntyped, fullsimple and fullsub. However, such featuring sharing is achieved via copying and pasting code, causing problems like:

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

**Refactored Implementation.** We refactor the non-modular implementation using CASTOR. Our refactoring mainly focuses on the syntax and core. Figure 10 gives a simplified high-level overview of the refactored implementation, where the candidate languages are represented as gray boxes and extracted features/sub-languages are represented as white boxes. From Figure 10 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 research questions:

- **RQ1.** Is CASTOR effective in reducing SLOC?
- **RQ2.** How does CASTOR compare to other modularization techniques/frameworks in reducing SLOC?
- **RQ3.** How much performance penalty does CASTOR introduce?

**RQ1.** 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.

**RQ2.** Table 2 also compares CASTOR with EVF [Zhang and Oliveira 2017], 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 over 500 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.
Fig. 11. Performance evaluation of TAPL interpreters.

Fig. 12. Performance evaluation of Arith.

RQ3. To measure the performance, we randomly generate 10,000 terms for each language and calculate the average evaluation time for 10 runs. The ScalaMeter\(^5\) 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 11 compares the execution time in 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. We also compare the performance of the ARITH implementations discussed in Section 2. Figure 12 depicts the results. Without surprise, all modular implementations introduce similar performance penalty.

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.

\(^5\)http://scalameter.github.io/
6 RELATED WORK
In this section we review closely related work that is not discussed in previous sections.

**Polymorphic Variants.** The OCaml programming language supports polymorphic variants [Garri
guie 1998]. Unlike traditional variants, polymorphic variant constructors are defined individually
and are not tied to a particular data type. Garrigue [2000] presents a solution to the EP using
polymorphic variants. To correctly deal with recursive calls, open recursion and explicit fixed-
point operator have to 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
object-oriented dynamic dispatching, reducing the burden of programmers significantly.

**Open Data Types and Open Functions.** To solve the EP, Löh and Hinze [2006] propose to extend
Haskell with open data types and open functions. Different from classic closed data types and
closed functions, they proposed open variants to decentralize the definition of data types and
functions; and a mechanism that reassembles the pieces into a complete definition. To avoid
unanticipated captures caused by classic *first-fit* pattern matching, they propose a *best-fit*
scheme, which rearranges patterns according to their specificness rather than the order they appear. For
example, the wildcard pattern is least specific and hence is put to the end of clauses. However open
data types 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.** Instead of extending Haskell, Data types à la carte (DTC) [Swierstra 2008]
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 [2011] extends DTC with GADTs and automatically
generates boilerplate using Template Haskell [Sheard and Jones 2002]. Oliveira et al. [2015] use list-
of-functors instead of co-products to better simulate object-oriented language features subtyping,
hierarchy 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 [Carette et al.
However, these techniques do not support pattern matching and/or dependencies. This makes it
hard to define operations like small-step semantics discussed in Section 2. Typical workarounds are
defining the operation together with what it depends on or use advanced features like intersection
types and a merge operator [Oliveira et al. 2013; Rendel et al. 2014]. As shown in Section 3 Castor
allows us to implement operations that need nested patterns and/or with dependencies in a simple
and intuitive way.

**Case Classes and Extractors.** Emir et al. [2007] introduced case classes and extractors as complemen-
tary pattern matching mechanisms to the Scala language. We have discussed case classes
and their advantages and disadvantages in Section 2. Extractors are companion objects with user-
declared apply and unapply methods. Emir et al. [2007] compare case classes and extractors with
other four object-oriented 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—totality and composability—are not
addressed. Extractors does not meet these two properties as well.

**Other Approaches to Pattern Matching in Object-Oriented Programming.** There are many attempts
to introduce pattern matching into mainstream OOP languages like Java [Hirzel et al. 2008; Liu
and Myers 2003] and C++ [Solodkyy et al. 2013]. New OOP languages are designed with first-class
pattern matching such as Newspeak [Geller et al. 2010], Fortress [Ryu et al. 2010] or Grace [Homer
et al. 2012]. Yet, how to cooperate pattern matching with the open nature of OOP class hierarchy
while preserving the desirable features is still challenging. Isradisaikul and Myers [2013] use an SMT
solver to reason about the exhaustiveness and redundancy of pattern matching while preserving
extensibility.

**Multimethods.** Multimethods [Chambers 1992; Clifton et al. 2000; Millstein et al. 2004] allow a
series of methods of 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 can be selected
and invoked. With multimethods, shallow patterns can be simulated, making it easy to define
binary methods like structural equality. It is unclear how to do deep patterns with multimethods.
Also, checking the totality is difficult in multimethods.

**Extensible Visitors.** Our work can be viewed as a continuation of work on extensible visitors. The
extensible visitor encoding presented in Section 4 combines ideas from recent work on extensible
visitors in Scala [Hofer and Ostermann 2010; Odersky and Zenger 2005; Oliveira 2009] for better
supporting pattern matching. A common problem of these encodings is that although powerful,
they are too complicated to be used in practice. CASTOR simplifies the encoding by providing code
generation associated with visitors.

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 have presented CASTOR, a Scala metaprogramming library for writing concise, total, extensible
and composable pattern matching code. As discussed in Section 4, the current implementation of
CASTOR has some limitations. Production ready Scala macros are being actively developed and are
supposed to be released in the near future. These limitations might have a chance to be fixed or
relieved by that time. Still, CASTOR is restricted by Scala: we cannot have full language support for
the programming style proposed in this work. One path for future work is to work on languages
that support this programming style in the first place. Another path for future work is to study
support for GADTs [Xi et al. 2003] better. CASTOR already provides some support for GADTs and
can be used to model simple, typed, embedded DSLs with such support. For space reasons we did
not discuss such support in the main body of the paper, but Appendix A shows CASTOR’s support
for GADTs. We would like to see if more realistic embedded DSLs can be modeled in CASTOR in the
future with its support for GADTs.
A MODULAR GENERALIZED ALGEBRAIC DATA TYPES

We have shown how to rule out ill-typed terms using a notion of types and a type-checking algorithm in Section 3.4. A better solution may prevent such terms from being constructed in the first place. The idea is to use generalized algebraic data types (GADTs) [Xi et al. 2003]. GADTs allow not only data types to be parameterized but also well-formedness constraints to be expressed in constructors. This approach is widely used for embedded DSLs, which can then exploit the type system of the host language to type-check the terms of the DSL. This approach is also employed by techniques such as finally tagless [Carette et al. 2009], which provide an encoding of GADTs and provide modularity as well. Nevertheless the encodings employed by finally tagless are based on Church encodings and do not provide support for pattern matching. Unfortunately, this makes it hard to model several operations that require more advanced features such as nested patterns or operations with dependencies. In this section, we present the GADT support in Castor by revisiting the Arith language. We show that, just as finally tagless encodings, modularity is supported; and like GADTs nested pattern matching and/or dependencies are easy to do as well.

A.1 Modular AST and Well-Typed Terms

A generalized term datatype is declared like this:

```scala
@family trait GTerm {
  @adt trait Tm[A]
  ...
}
```

where `Tm` is parameterized by a type parameter `A`. The type parameter `A` is instantiated differently in constructor types using `Int` or `Boolean` for `NAT` and `BOOL`:

```scala
@family trait GNat extends GTerm {
  @adt trait Tm[A] extends super.Tm[A] {
    def TmZero: Tm[Int]
    def TmSucc: Tm[Int] => Tm[Int]
    def TmPred: Tm[Int] => Tm[Int]
  }
  ...
}
```

```scala
@family trait GBool extends GTerm {
  ...
}
```

@adt trait Tm[A] extends super.Tm[A] {
    def TmTrue: Tm[Boolean]
    def TmFalse: Tm[Boolean]
    def TmIf[A]: (Tm[Boolean], Tm[A], Tm[A]) => Tm[A]
}

... 

We further unify the two extended GADTs for Arith:

@family trait GArith extends GNat with GBool {
    @adt trait Tm[A] extends super.GNat.Tm[A] with super.GBool.Tm[A] {
        def TmIsZero: Tm[Int] => Tm[Boolean]
    }
    ...
}

Now, ill-formed terms are statically rejected by the type system, for example:

import GArith._
val t1 = TmIsZero(TmZero) // Type-checked!
val t2 = TmIsZero(TmTrue) // Not type-checked!
TmIsZero expects a term of type Tm[Int] while TmTrue is of type Tm[Boolean], thus causing a compile-time error.

### A.2 Modular Well-Typed Big-Step Evaluator

The type information carried by terms can be utilized in defining the big-step evaluator Eval. Similar to finally tagless, Castor allows us to define well-typed big-step evaluator modularly.

We first give the signature of Eval:

@visit(Tm) trait Eval { type OTm[A] = A } // Inside GTerm

where the output type is consistent with the type parameter of Tm. We then define well-typed big-step evaluator respectively for Nat and Bool:

// Inside GNat
@visit(Tm) trait Eval extends super.Eval {
    def tmZero = 0
    def tmSucc = this(_).+ 1
    def tmPred = this(_).- 1
}

// Inside GBool
@visit(Tm) trait Eval extends super.Eval {
    def tmTrue = true
    def tmFalse = false
    def tmIf[A] = (t1, t2, t3) => if (this(t1)) this(t2) else this(t3)
}

The definition for each case is rather straightforward. The evaluation result can either be Int or Boolean depending on how the term is constructed. Finally, we can unify the two independently developed big-step evaluators for Arith:

// Inside GArith
@visit(Tm) trait Eval extends super[GNat].Eval with super[GBool].Eval {
    def tmIsZero = this(_).== 0
}

As a test, calling eval(t1) returns true as expected.
Imagine implementing the big-step evaluator for \texttt{Arith} without using GADTs. How should we instantiate the output type \texttt{OTm}? We may use a union type (e.g. \texttt{Either[Int,Boolean]}) to accommodate two possible evaluation results (Int and Boolean). However, this definition needs some boilerplate for wrapping and unwrapping around union values. Worse, this definition can no longer be modularly extended when \texttt{Arith} evolves with new primitive terms such as floating numbers because a \textit{different} union type is needed that additionally takes \texttt{Float} into account.

### A.3 Modular Well-Typed Small-Step Evaluator

What distinguishes \texttt{Castor} from finally tagless is in defining the small-step semantics. The need for nested case analysis and the dependency on numeric value checker causes troubles for finally tagless. In contrast, this is unproblematic for GADTs in \texttt{Castor}.

We start with the signature of \texttt{Eval1}:

```scala
// Inside \texttt{GTerm}
@default(Tm) trait Eval1 {
  type OTm[A] = Tm[A]
  def otherwise[A] = (x: Tm[A]) => throw NoRuleApplies
}
```

Different from that in Figure 6, the \texttt{otherwise} method is generic. We now implement the small-step evaluator for \texttt{Nat} and \texttt{Bool}:

```scala
// Inside \texttt{GNat}
def nv[A](t: Tm[A]): Boolean = t match {
  case TmZero => true
  case TmSucc(t1) => nv(t1)
  case _ => false
}
@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))
  }
}
```

```scala
// Inside \texttt{GBool}
@default(Tm) trait Eval1 extends super.Eval1 {
  override def tmIf[A] = {
    case (TmTrue,t2,_) => t2
    case (TmFalse,_,t3) => t3
    case (t1,t2,t3) => TmIf(this(t1),t2,t3)
  }
}
```

The definitions are almost the same as those in Figure 6 except that \texttt{nv} and \texttt{tmIf} become generic.

We are still able to do nested pattern matching and call \texttt{nv} in \texttt{Eval1} of \texttt{Nat}. Again, unifying the two small-step evaluators for \texttt{Arith} is not a problem as well:

```scala
// Inside \texttt{GArith}
@visit(Tm) trait Eval1 extends super[GNat].Eval1 with super[GBool].Eval1 {
  def tmIsZero = {
    case TmZero => TmTrue
    case TmSucc(t) if nv(t) => TmFalse
    case t => TmIsZero(this(t))
  }
}
```
Calling \texttt{eval1(t1)} returns \texttt{True} as expected.