香 港 大 學
THE UNIVERSITY OF HONG KONG

Pattern Matching in an Open World

Weixin Zhang and Bruno C. d. S. Oliveira

GPCE 2018

November 6, 2018

Motivation

FP

```
data Term = Lit Int
          | Add Term Term

eval :: Term -> Int
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
```

OOP

```
trait Term {
  def eval: Int
}
class Lit(n: Int) extends Term {
  def eval = n
}
class Add(t1: Term, t2: Term) extends Term {
  def eval = t1.eval + t2.eval
}
```

- Key observations
  - Algebraic datatypes are **closed** whereas class hierarchies are **open**
  - Conventional pattern matching semantics is based on the **order** of patterns
  - This is in conflict with the open data structures
- Our proposal
  - Top-level patterns should be **order irrelevant**
  - Deep patterns should come with a **default**

Desirable Properties of Open Pattern Matching

- **Conciseness**
  - Patterns should be concise with 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
  - Missing cases should be warned
- **Extensibility**
  - New data variants can be added while existing operations can be reused without modification
- **Composability**
  - Complex patterns can be built from smaller pieces
  - When composing overlapped patterns, redundancies should be warned

Evaluating Pattern Matching Approaches (in Scala)

	Conciseness	Exhaustiveness	Extensibility	Composability
VISITOR pattern	✗	✓	✗	✗
Sealed case class	✓	✓	✗	✗
Open case class	✓	✗	✓	✗
Partial function	✓	✗	✓	✓✗
CASTOR	✓	✓✗	✓	✓

Contributions

- Desirable properties for open pattern matching
- The CASTOR (case class visitor) meta-programming library
- Non-trivial modular operations
- Case study on Types and Programming Languages (TAPL)

Running Example: ARITH

t ::=		*terms:*
	0	*constant zero*
	succ t	*successor*
	pred t	*predecessor*
	true	*constant true*
	false	*constant false*
	if t then t else t	*conditional*
	iszero t	*zero test*

NAT

BOOL

nv ::=		*numeric values:*
	0	*zero value*
	succ nv	*successor value*

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \qquad \frac{}{\texttt{pred}\ 0 \longrightarrow 0}$$

$$\frac{}{\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1}\ \textsc{PredSucc}$$

$$\frac{\to t_1'}{\to \texttt{pred}\ t_1'}\ \textsc{Pred}$$

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

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

The Visitor Pattern

Conciseness	Exhaustiveness	Extensibility	Composability
✗	✓	✗	✗

```
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)
}

trait TmVisit[A] {
  def tmZero: A
  def tmSucc(t: Tm): A
  def tmPred(t: Tm): A
}
```

```
object nv extends TmVisit[Boolean] {
  def tmZero = true
  def tmSucc(t: Tm) = t.accept(nv)
  def tmPred(t: Tm) = false
}

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] {
    def tmZero = TmZero
    def tmSucc(t1: Tm) =
      if (t1.accept(nv)) t1
      else new TmPred(t.accept(eval1))
    def tmPred(t1: Tm) =
      new TmPred(t.accept(eval1))
  })
}
```

Sealed Case Class

Conciseness	Exhaustiveness	Extensibility	Composability
✓	✓	✗	✗

```
sealed abstract class Tm
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 new NoRuleApplies
}
```

Open Case Class

Conciseness	Exhaustiveness	Extensibility	Composability
✓	✗	✓	✗

```
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 = 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
  }
}
```

```
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 {
  case class TmIsZero(t: Tm) extends Tm

  override def eval1(t: Tm): Tm = t match {
    case TmIsZero(TmZero) => TmTrue
    case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
    case TmIsZero(t1) => TmIsZero(eval1(t1))
    case _: TmSucc => super[Nat].eval1(t)
    case _: TmPred => super[Nat].eval1(t)
    case _: TmIf => super[Bool].eval1(t)
    case _ => throw NoRuleApplies
  }
}
```

Partial Function

Conciseness	Exhaustiveness	Extensibility	Composability
✓	✗	✓	✗

```scala
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 = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }

  def eval1: 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
  }
}
```

```scala
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: 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
  }
}

trait Arith extends Nat with Bool {
  case class TmIsZero(t: Tm) extends Tm

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

Castor

CASTOR

Conciseness	Exhaustiveness	Extensibility	Composability
✓	✗	✓	✓

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

@family trait Nat extends Term {
  @adt trait Tm extends super.Tm {
    def TmZero: Tm
    def TmSucc: Tm => Tm
    def TmPred: Tm => Tm
  }

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

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

```scala
@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))
    }
  }
}
```

Client Code

```
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
```

How Castor Addresses the Desirable Properties

- **Conciseness**
  - Scala's concise syntax for patterns and meta-programming
  - Ad-hoc auxiliary visitors are replaced by case clauses
- **Exhaustiveness**
  - The exhaustiveness of top-level patterns is verified when generated code is type checked
  - Deep patterns should come with a default, but this is not enforced
- **Extensibility**
  - The underlying extensible visitor encoding
- **Composability**
  - Scala's mixin composition
  - Overlapped patterns are conflicting methods

Generating Extensible Visitors

- CASTOR employs **Scalameta** in generating code associated with visitors
- The extensible visitor encoding combines ideas from the literature
  [Odersky & Zenger 2005; Hofer & Ostermann 2010; Oliveira 2009; Zhang and Oliveira 2017]

```
@family trait Term {
  @adt trait Tm

  @default(Tm) trait Eval1 {
    type OTm = Tm
    def otherwise = _ =>
      throw NoRuleApplies
  }
}
```


```
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 =>
    type OTm = Tm
    def otherwise = _ => throw NoRuleApplies
  }
  val eval1: Eval1
}
```

Generating Extensible Visitors (Continued)

```
@family trait Nat extends Term {

  @adt trait Tm extends super.Tm {
    def TmZero: Tm
    def TmSucc: Tm => Tm
    def TmPred: Tm => Tm
  }

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

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

CASTOR

```
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) …
  case class TmPred(t: Tm) …

  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 => … }
```

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

Exhaustiveness checking

Limitations

- Unnecessary annotations
  - @adts, @ops
- Boilerplate nested composition

```
@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 = …
  }
}
```

```
@family trait Arith extends Nat with Bool {
  @adt trait Tm {
    def TmIsZero: Tm => Tm
  }
  @visit(Tm) trait Eval1 {
    def tmIsZero = …
  }
}
```

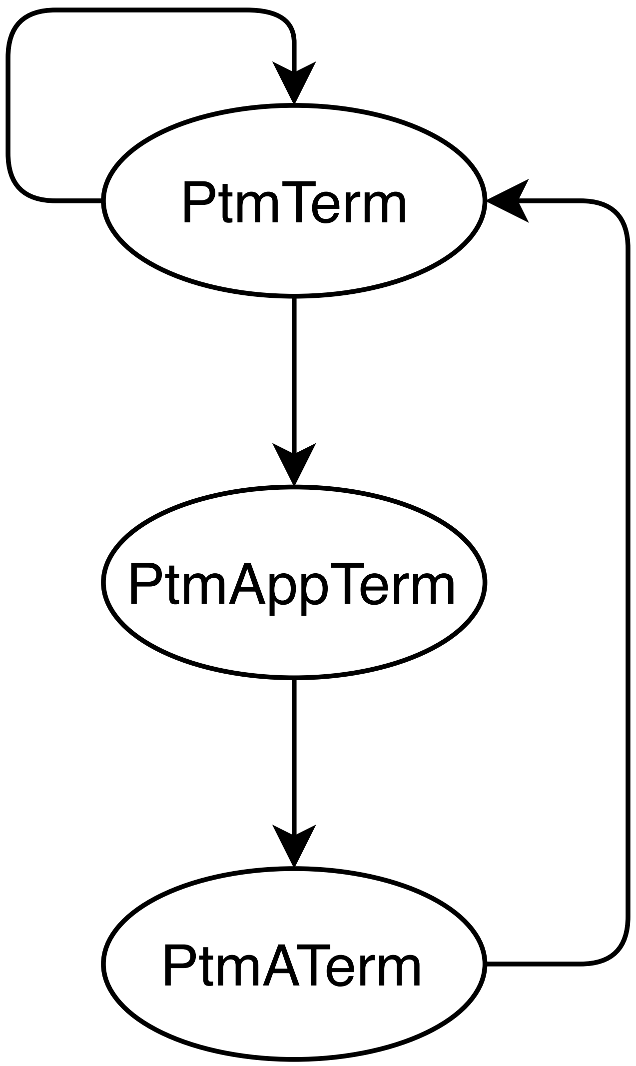
- Imprecise error messages
  - Wrong line number
  - Details of the visitor encoding

Operations with Dependencies

- Printing out parenthesis only when necessary

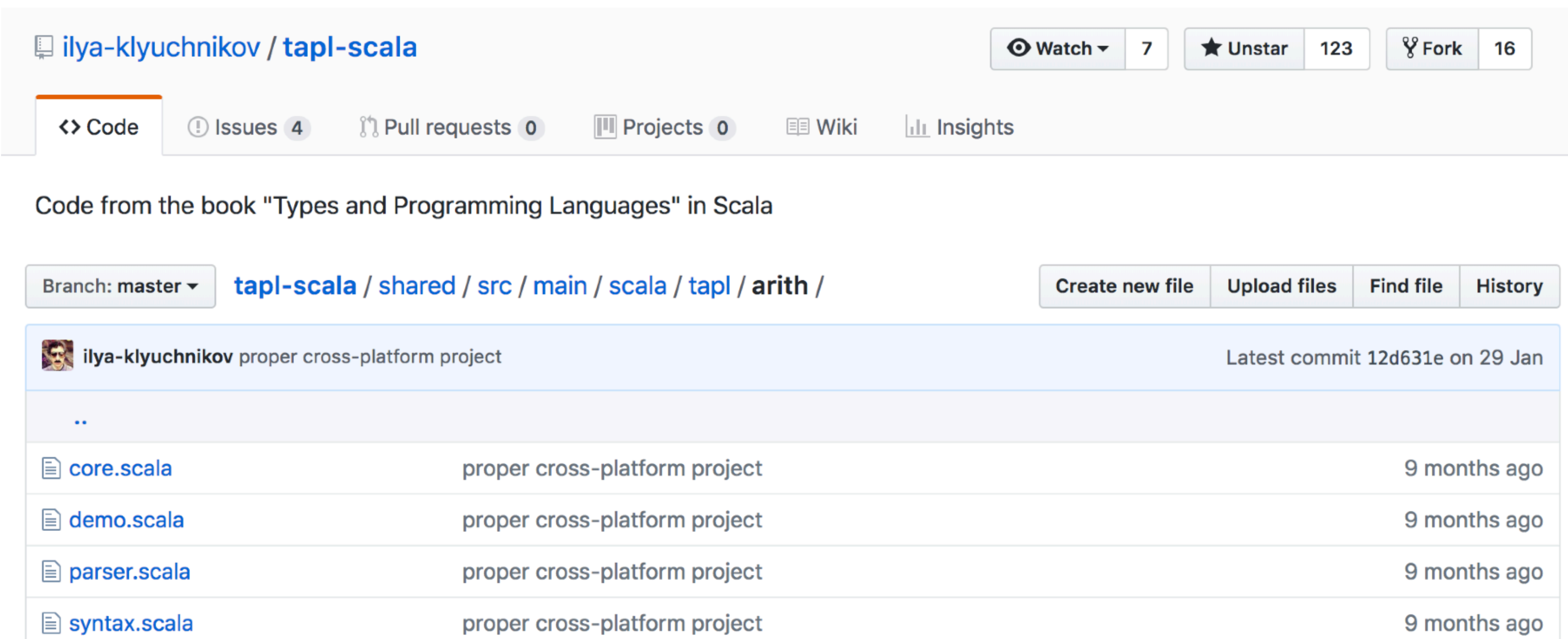
e.g. `TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))`
`"iszero (if false then true else pred (succ 0))"`

```
@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"
  }
}
```



Case Study

- TAPL is a good benchmark for accessing modularity
  - Small-step semantics, feature sharing, complex operations, etc.
- We refactored a non-modular Scala implementation



Problems Caused by Copy-Paste

- Inconsistent definitions

```scala
def ptmTerm(outer: Boolean, ctx: Con
  case TmAbs(x, t2) =>
    val (ctx1, x1) = ctx.pickFreshNa
    val abs = g0("\\" :: x1 :: ".")
    val body = ptmTerm(outer, ctx1,
    g2(abs :/: body)
  case t => ptmAppTerm(outer, ctx, t

}
```

Untyped

```scala
def ptmTerm(outer: Boolean, ctx: Context, t: Term): Document = t match {

  case TmIf(t1, t2, t3) =>
    val ifB = g2("if" :/: ptmTerm(outer, ctx, t1))
    val thenB = g2("then" :/: ptmTerm(outer, ctx, t2))
    val elseB = g2("else" :/: ptmTerm(outer, ctx, t3))
    g0(ifB :/: thenB :/: elseB)
  case TmAbs(x, t2) =>
    val (ctx1, x1) = ctx.pickFreshName(x)
    val abs = g0("lambda" :/: x1 :: ".")
    val body = ptmTerm(outer, ctx1, t2)
    g2(abs :/: body)
```

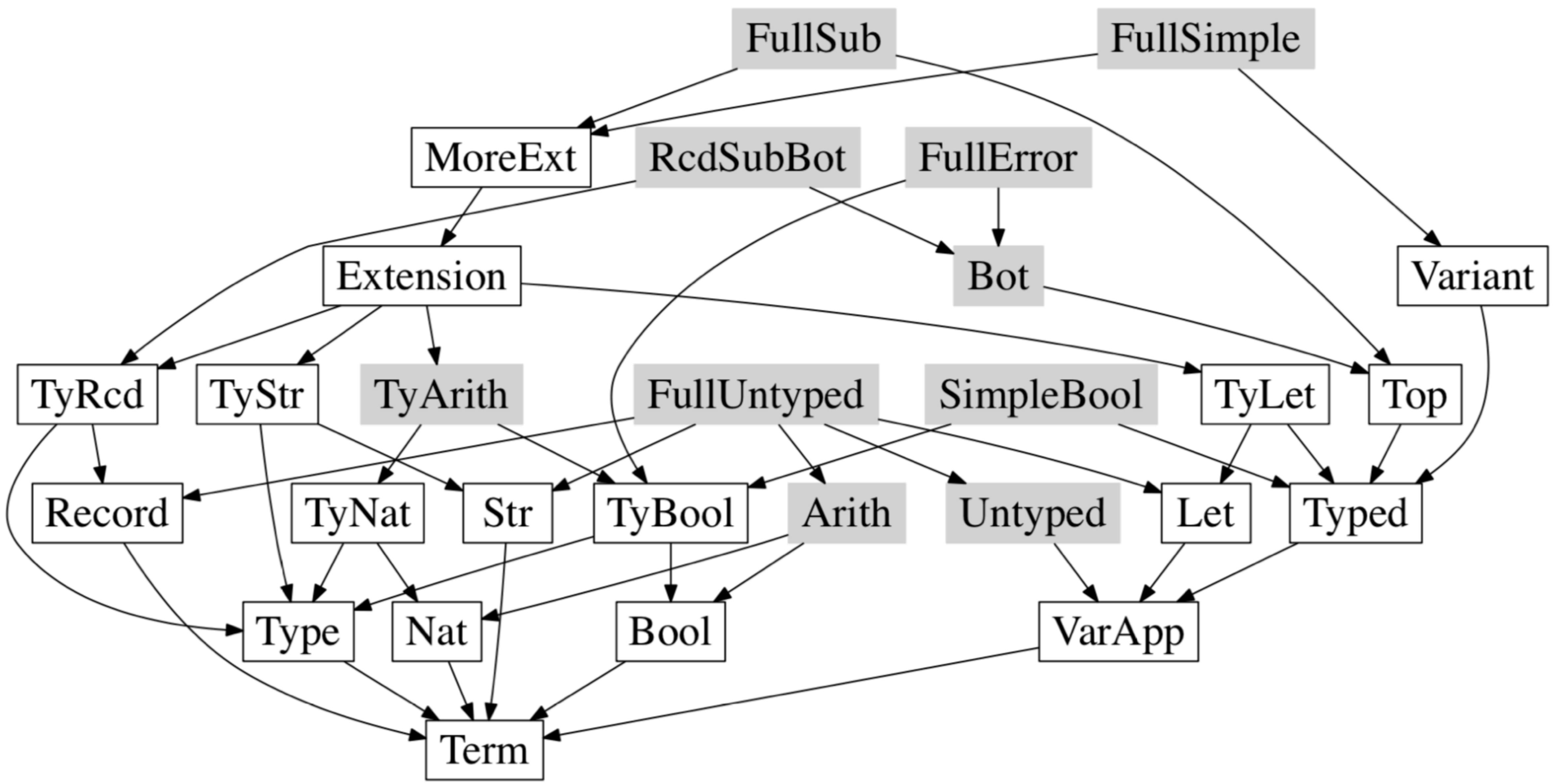
FullUntyped

- Feature leaks

```scala
// (tm ty) reduction - for system F
def typeSubstTop(tyS: Ty, tyT: Ty): Ty =
  typeShift(-1, typeSubst(typeShift(1, tyS), 0, tyT))

// really this is for system F only
private def tytermSubst(tyS: Ty, j: Int, t: Term) =
  tmMap((c, tv) => tv, (j, tyT) => typeSubst(tyS, j, tyT), j, t)

// really this is for system F only
def tyTermSubstTop(tyS: Ty, t: Term): Term =
  termShift(-1, tytermSubst(typeShift(1, tyS), 0, t))

```

FullSimple

An Overview of the Refactored Implementation



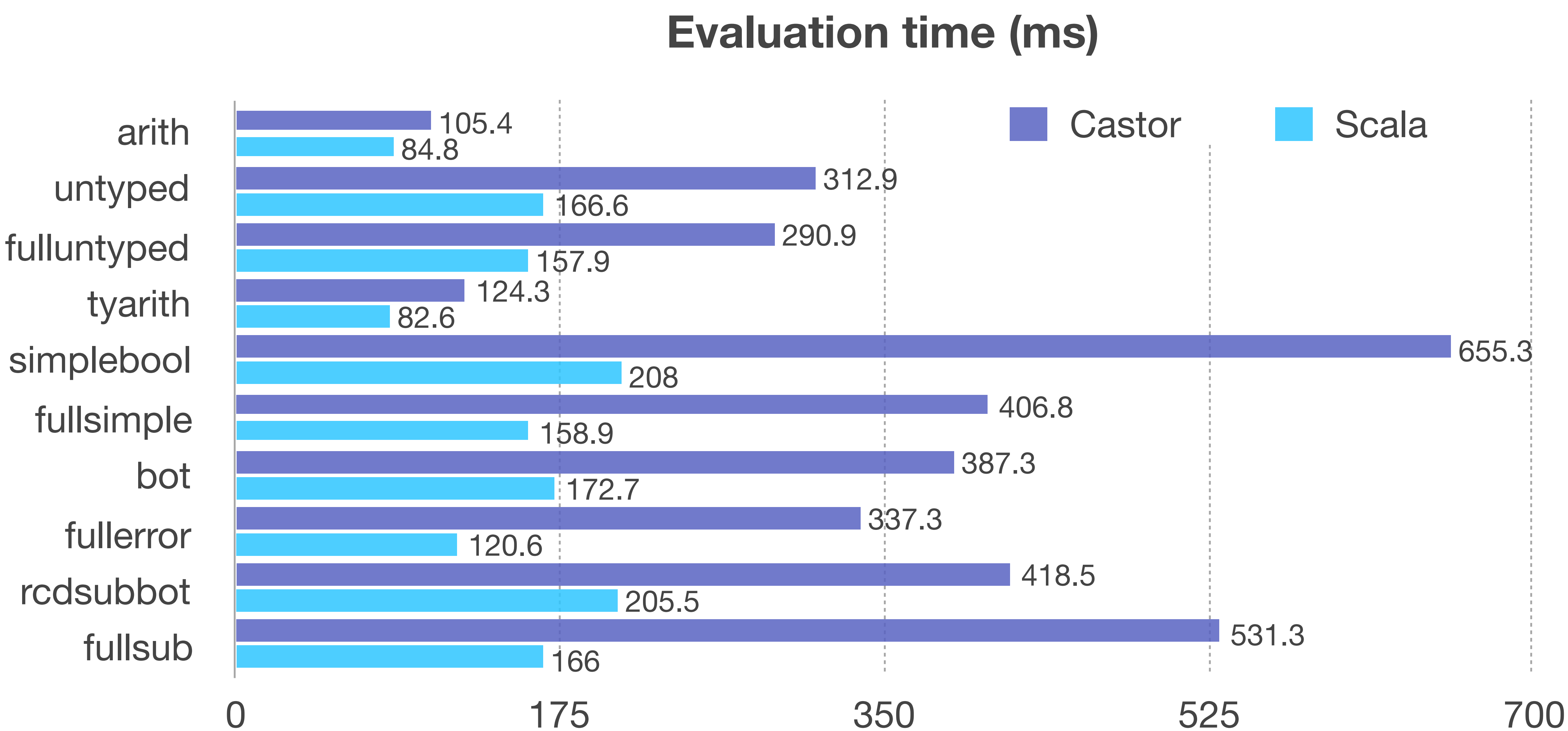
SLOC Comparison

- Is CASTOR effective in reducing SLOC?
- How does CASTOR compare to EVF [Zhang and Oliveira 2017]?

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**

Performance Evaluation

- How much performance overhead does CASTOR incur?



1.2x ~ 3.2x
arith fullsub

More in the Paper

- Examples on non-trivial operations
  - Structural equality
  - Typed arith
- Formalized code generation
- Related and Future work

Conclusions

- Pattern matching in an extensible setting would benefit from an **unordered** semantics for (top-level) patterns
- Nested patterns should come with a **default**, which is often the case in practice
- With this idea applied, CASTOR allows **concise**, **exhaustive**, **extensible** and **composable** pattern matching code

https://github.com/wxzh/Castor

Q & A

- Thanks!