Matching Objects With Patterns

Burak Emir[1], Martin Odersky[1], and John Williams

LAMP-REPORT-2006-006
EPFL, 1015 Lausanne

Abstract. Data in object-oriented programming is organized in a hierarchy of classes. The problem of *object-oriented pattern matching* is how to explore this hierarchy from the outside. This usually involves classifying objects by their run-time type, accessing their members, or determining some other characteristic of a group of objects. In this paper we compare six different pattern matching techniques: object-oriented decomposition, visitors, type-tests/type-casts, typecase, case classes, and extractors. The techniques are compared on nine criteria related to conciseness, maintainability and performance. The paper introduces case classes and extractors as two new pattern-matching methods and shows that their combination works well for all of the established criteria.

1 Introduction

Data in object-oriented programming is organized in a hierarchy of classes. The problem of *object-oriented pattern matching* is how to explore this hierarchy from the outside. This usually involves classifying objects by their run-time type, accessing their members, or determining some other characteristic of a group of objects. Here, we take a very general view of patterns. A pattern is simply some way of characterizing a group of objects and binding local names to objects that match some property in the classification.

A number of functional languages are built on patterns as an essential syntactic constructs. Examples are SML, OCaml or Haskell. In object-oriented languages, patterns are much less common, even though some research exists [20, 21, 31, 17]. Mainstream object-oriented languages propose to do pattern matching through encodings, such as virtual classification methods, visitors, or type-tests and type-casts.

The reason why patterns have so far played a lesser role in object-oriented languages might have to do with the object-oriented principle which states that behavior should be bundled with data and that the only form of differentiation should be through virtual method calls. This principle works well as long as (1) one can plan from the start for all patterns that will arise in an application, and (2) one only needs to decompose one object at a time.

However, these two assumptions do not always hold. The extensive literature on the expression problem [5, 15, 25, 32] has explored many situations where access patterns are constructed a-posteriori, after the interface of the base class is fixed. Furthermore, there are access patterns where the result depends on the identity of several objects.

Consider for instance symbolic manipulation of expressions. We assume a hierarchy of classes, rooted in a base class Expr and containing classes for specific forms of expressions, such as Mul for multiplication operations, Var for variables, and Num for numeric literals. Different forms of expressions have different members: Mul has two members left and right denoting its left and right operand, whereas Num has a member value denoting an integer. A class hiercharchy like this is expressed as follows (we use Scala as programming notation throughout the paper).

```
class Expr
class Num(val value : int) extends Expr
class Var(val name : String) extends Expr
class Mul(val left : Expr, val right : Expr) extends Expr
```

A particular expression would then be constructed as follows

```
new Mul(new Num(21), new Num(2))
```

Let's say we want to write a simplifier for arithmetic expressions. This program should try to apply a set of simplification rules, until no more rewrites are possible. An example simplification rule would make use of the right-neutrality of the number one. That is,

```
new Mul(x, new Num(1))   is replaced with   x .
```

The question is how simplification rules like the one above can be expressed. This is an instance of the object-oriented pattern matching problem, where objects of several variant types connected in possibly recursive data structures need to be classified and decomposed from the outside.

We will review in this paper the six techniques for this task: (1) classical object-oriented decomposition, (2) visitors, (3) type-tests/type-casts, (4) typecase, (5) case classes, and (6) extractors. Of these, the first three are well known in object-oriented languages. The fourth technique, typecase, is well known in the types community, but its extensions to type patterns in Scala is new. The fifth technique, case classes, is specific to Scala. The sixth technique, extractors, is new. It has been proposed independently by John Williams for Scala [29] and by Don Syme under the name "active patterns" for F$^{\#}$ [24]. The basic F$^{\#}$ design is in many ways similar to the Scala design, but Scala's treatment of parametricity is different.

Every technique will be evaluated along nine criteria. The first three criteria are concerned with conciseness of expression:

1. *Conciseness/framework*: How much "boilerplate" code needs to be written to enable classifications?
2. *Conciseness/shallow matches*: How easy is it to express a simple classification on the object's type?
3. *Conciseness/deep matches*: How easy is it to express a deep classification involving several objects?

The next three criteria assess program maintainability and evolution. In big projects, their importance often ranks highest.

4. *Representation independence*: How much of an object's representation needs to be revealed by a pattern match?
5. *Extensibility/variants*: How easy is it to add new data variants after a class hierarchy is fixed?
6. *Extensibility/patterns*: How easy is it to add new patterns after a class hierarchy is fixed? Can new patterns be expressed with the same syntax as existing ones?

Note that all presented schemes allow extensions of a system by new processors that perform pattern matching (one of the two dimensions noted in the expression problem). After all, this is what pattern matching is all about! The last three considered criteria have to do with performance and scalability:

7. *Base performance*: How efficient is a simple classification?
8. *Scalability/breadth*: How does the technique scale if there are many different cases?
9. *Scalability/depth*: How does the technique scale for larger patterns that reach several levels into the object graph? Here it is important that overlaps between several patterns in a classification can be factored out so that they need to be tested only once.

Our evaluation will show that a combination of case classes and extractors can do well in all of the nine criteria.

A difficult aspect of decomposition is its interaction with static typing, in particular type-parametricity. A subclass in a class hierarchy might have either fewer or more type parameters than its base class. This poses challenges for the precise typing of decomposing expressions which have been studied under the label of ”generalized algebraic data-types”, or GADT’s [30, 14]. The paper develops a new algorithm for recovering static type information from patterns in these situations.

Related work Pattern matching in the context of object-oriented programming has been applied to message exchange in distributed systems [16], semistructured data [12] and UI event handling [3].

Moreau, Ringeissen and Vittek [20] translate pattern matching code into existing languages, without requiring extensions. Liu and Myers [17] add a pattern matching construct to Java by means of a backward mode of execution.

Multi-methods [1, 2, 19, 4] are an alternative technique which unifies pattern matching with method dispatch. Multi-methods are particularly suitable for matching on several arguments at the same time. An extension of multi-methods to predicate-dispatch [8, 18] can also access embedded fields of arguments; however it cannot bind such fields to variables, so support for deep patterns is limited.

Views in functional programming languages [26, 22] are conversions from one data type to another that are implicitly applied in pattern matching. They play a role similar to extractors in Scala, in that they permit to abstract from the concrete data-type of the matched objects. However, unlike extractors, views are anonymous and are tied to a particular target data type. Erwig’s active patterns [9] provide views for non-linear patterns with more refined computation rules. Gostanza et al.’s active destructors [13] are closest to extractors; an active destructor corresponds almost exactly to an unapply method in an extractor. However, they do not provide data type injection, which is handled by the corresponding apply method in our design. Also, being tied to traditional algebraic data types, active destructors cannot express inheritance with varying type parameters in the way it is found in GADT’s.

2 Standard Techniques

In this section, we review four standard techniques for object-oriented pattern matching. These are, first, object-oriented decomposition using tests and accessors, second, visitors,

```
// Class hierarchy:
trait Expr {
  def isVar : boolean  = false
  def isNum : boolean = false
  def isMul : boolean  = false
  def value : int      = throw new NoSuchMemberError
  def name : String    = throw new NoSuchMemberError
  def left : Expr      = throw new NoSuchMemberError
  def right : Expr     = throw new NoSuchMemberError
}
class Num(override val value : int) extends Expr {
  override def isNum = true }
class Var(override val name : String) extends Expr {
  override def isVar = true }
class Mul(override val left : Expr, override val right : Expr) extends Expr {
  override def isMul = true }
// Simplification rule:
  if (e.isMul) {
    val r = e.right
    if (r.isNum && r.value == 1) e.left else e
  } else e
```

Fig. 1. Expression simplification using object-oriented decomposition

third, type-tests and type-casts, and fourth, typecase. We explain each technique in terms of the arithmetic simplification example that was outlined in the introduction. Each technique is evaluated using the six criteria for conciseness and maintainability that were developed in the introduction. Performance evaluations are deferred to Section 5.

2.1 Object-Oriented Decomposition

In *classical OO decomposition*, the base class of a class hierarchy contains *test methods* which determine the dynamic class of an object and *accessor methods* which let one refer to members of specific subclasses. Some of these methods are overridden in each subclass. Figure 1 demonstrates this technique with the numeric simplification example.

The base class Expr contains test methods isVar, isNum and isMul, which correspond to the three subclasses of Expr. All test methods return **false** by default. Each subclass re-implements "its" test method to return **true**. The base class also contains one accessor method for every publicly visible field that is defined in some subclass. The default implementation of every access method in the base class throws a NoSuchMemberError exception. Each subclass re-implements the accessors for its own members. Scala makes these re-implementations particularly easy because it allows one to unify a class constructor and an overriding accessor method in one syntactic construct, using the syntax **override val** ... in a class parameter.

Note that in a dynamically typed language like Smalltalk, the base class needs to define only tests, not accessors, because missing accessors are already caught at run-time and are turned into NoSuchMethod messages. So the OO-decomposition pattern becomes considerably more lightweight. That might be the reason why this form of decomposition is more prevalent in dynamically typed languages than in statically typed ones. But even then the technique can be heavy. For instance, Squeak's Object class contains 35 test methods that each inquire whether the receiver is of some (often quite specific) subclass.

Besides its bulk, the object-oriented decomposition technique also suffers from its lack of extensibility. If one adds another subclass of Expr, the base class has to be augmented with new test and accessor methods. Again, dynamically typed languages such as Smalltalk alleviate this problem to some degree using meta-programming facilities where classes can be augmented and extended at run-time.

The second half of Figure 1 shows the code of the simplification rule. The rule inspects the given term stepwise, using the test functions and accessors given in class Expr.

Evaluation: In a statically typed language, the OO decomposition technique demands a high notational overhead for framework construction, because the class-hierarchy has to be augmented by a large number of tests and accessor methods. The matching itself relies on the interplay of many small functions and is therefore often somewhat ad-hoc and verbose. This holds especially for deep patterns. Object-oriented decomposition maintains complete representation independence. Its extensibility characteristics are mixed. It is easy to add new forms of matches using existing tests and accessors. If the underlying language has a concept of open classes or mixin composition, these matches can sometimes even be written using the same method call syntax as primitive matches. On the other hand, adding new subclasses requires a global rewrite of the class-hierarchy.

2.2 Visitors

Visitors [10] are a well-known design pattern to simulate pattern matching using double dispatch. Figure 2 shows the pattern in the context of arithmetic simplification. Because we want to cater for non-exhaustive matches, we use visitors with defaults [31] in the example. The Visitor trait contains for each subclass X of Expr one *case-method* named *caseX*. Every *caseX* method takes an argument of type X and yields a result of type T, the generic type parameter of the Visitor class. In class Visitor every case-method has a default implementation which calls the otherwise method.

The Expr class declares a generic abstract method matchWith, which takes a visitor as argument. Instances of subclasses X implement the method by invoking the corresponding *caseX* method in the visitor object on themselves.

The second half of Figure 2 shows how visitors are used in the simplification rule. The pattern match involves one visitor object for each of the two levels of matching. (The third-level match, testing whether the right-hand operand's value is 1, uses a direct comparison). Each visitor object defines two methods: the *caseX* method corresponding to the matched class, and the otherwise method corresponding to the case where the match fails.

Evaluation: The visitor design pattern causes a relatively high notational overhead for framework construction, because a visitor class has to be defined and matchWith methods

```
// Class hierarchy:
trait Visitor[T] {
  def caseMul(t: Mul): T     = otherwise(t)
  def caseNum(t: Num): T     = otherwise(t)
  def caseVar(t: Var): T     = otherwise(t)
  def otherwise(t: Expr): T  = throw new MatchError(t)
}
trait Expr {
  def matchWith[T](v: Visitor[T]): T }
class Num(val value: int) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseNum(this) }
class Var(val name: String) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseVar(this) }
class Mul(val left: Expr, val right: Expr) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseMul(this) }
// Simplification rule:
  e.matchWith {
    new Visitor[Expr] {
      override def caseMul(m: Mul) =
        m.right.matchWith {
          new Visitor[Expr] {
            override def caseNum(n: Num) =
              if (n.value == 1) m.left else e
            override def otherwise(e: Expr) = e
          }
        }
      override def otherwise(e: Expr) = e
    }
```

Fig. 2. Expression simplification using visitors

have to be provided in all data variants. The pattern matching itself is disciplined but very verbose, especially for deep patterns. Visitors in their standard setting do not maintain representation independence, because case methods correspond one-to-one to data alternatives. However, one could hide data representations using some ad-hoc visitor dispatch implementation in the matchWith methods. Visitors are not extensible, at least not in their standard form presented here. Neither new patterns nor new alternatives can be created without an extensive global change of the visitor framework. Extensible visitors [15] address the problem of adding new alternatives (but not the problem of adding new patterns) at the price of a more complicated framework.

```
// Class hierarchy:
trait Expr
class Num(val value : int) extends Expr
class Var(val name : String) extends Expr
class Mul(val left : Expr, val right : Expr) extends Expr

// Simplification rule:
  if (e.isInstanceOf[Mul]) {
    val m = e.asInstanceOf[Mul]
    val r = m.right
    if (r.isInstanceOf[Num]) {
      val n = r.asInstanceOf[Num]
      if (n.value == 1) m.left else e
    } else e
  } else e
```

Fig. 3. Expression simplification using type-test/type-cast

2.3 Type-Test/Type-Cast

The most direct (some would say: crudest) form of decomposition uses the type-test and type-cast instructions available in Java and many other languages. Figure 3 shows arithmetic simplification using this method. In Scala, the test whether a value x is a non-null instance of some type T is expressed using the pseudo method invocation x.isInstanceOf[T], with T as a type parameter. Analogously, the cast of x to T is expressed as x.asInstanceOf[T]. The long-winded names are chosen intentionally in order to discourage indiscriminate use of these constructs.

Evaluation: Type-tests and type-casts require zero overhead for the class hierarchy. The pattern matching itself is very verbose, for both shallow and deep patterns. In particular, every match appears as both a type-test and a subsequent type-cast. The scheme raises also the issue that type-casts are potentially unsafe because they can raise ClassCastExceptions. Type-tests and type-casts completely expose representation. They have mixed characteristics with respect to extensibility. On the one hand, one can add new variants without changing the framework (because there is nothing to be done in the framework itself). On the other hand, one cannot invent new patterns over existing variants that use the same syntax as the type-tests and type-casts.

2.4 Typecase

The typecase construct accesses run-time type information in much the same way as type-tests and type-casts. It is however more concise and secure. Figure 4 shows the arithmetic simplification example using typecase. In Scala, typecase is an instance of a more general pattern matching expression of the form $expr$ **match** { $cases$ }. Each case is of the form **case** $p \Rightarrow b$; it consists of a pattern p and an expression or list of statements b. There are

```
// Class hierarchy:
trait Expr
class Num(val value : int) extends Expr
class Var(val name : String) extends Expr
class Mul(val left : Expr, val right : Expr) extends Expr

// Simplification rule:
  e match {
    case m : Mul ⇒
      m.right match {
        case n : Num ⇒
          if (n.value == 1) m.left else e
        case _ ⇒ e
      }
    case _ ⇒ e
  }
```

Fig. 4. Expression simplification using typecase

several kinds of patterns in Scala. The typecase construct uses patterns of the form $x : T$ where x is a variable and T is a type. This pattern matches all non-null values whose runtime type is (a subtype of) T. The pattern binds the variable x to the matched object. The other pattern in Figure 4 is the *wildcard pattern* _, which matches any value.

Evaluation: Pattern matching with typecase requires zero overhead for the class hierarchy. The pattern matching itself is concise for shallow patterns but becomes more verbose as patterns grow deeper, because in that case one needs to use nested *match*-expressions. Typecase completely exposes object representation. It has the same characteristics as type-test/typecast with respect to extensibility: adding new variants poses no problems but new patterns require a different syntax.

3 Case Classes

Case classes in Scala provide convenient shorthands for constructing and analyzing data. Figure 5 presents them in the context of arithmetic simplification.

A case class is written like a normal class with a **case** modifier in front. This modifier has several effects. On the one hand, it provides a convenient notation for constructing data without having to write **new**. For instance, assuming the class hierarchy of Figure 5, the expression Mul(Num(42), Var(x)) would be a shorthand for **new** Mul(**new** Num(42), **new** Var(x)). On the other hand, case classes allow pattern matching on their constructor. Such patterns are written exactly like constructor expressions, but are interpreted "in reverse". For instance, the pattern Mul(x, Num(1)) matches all values which are of class Mul, with a right operand of class Num which has a value field equal to 1. If the pattern matches, the variable x is bound the left operand of the given value.

```
// Class hierarchy:
trait Expr
case class Num(value : int) extends Expr
case class Var(name : String) extends Expr
case class Mul(left : Expr, right : Expr) extends Expr

// Simplification rule:

  e match {
    case Mul(x, Num(1)) ⇒ x
    case _ ⇒ e
  }
```

Fig. 5. Expression simplification using case classes

Patterns

A pattern in Scala is constructed from the following elements:

– Variables such as x or right. These match any value, and bind the variable name to the value. The wildcard character _ is used as a shorthand if the value need not be named.
– Type patterns such as x : int or _ : String. These match all values of the given type, and bind the variable name to the value. Type patterns were already introduced in Section 2.4.
– Constant literals such as 1 or ”abc”. A literal matches only itself.
– Named constants such as None or Nil, which refer to immutable values. A named constant matches only the value it refers to.
– Constructor patterns of the form $C(p_1, \ldots, p_n)$, where C is a case class and $p_1, \ldots, p_n$ are patterns. Such a pattern matches all instances of class C which were built from values $v_1, \ldots, v_n$ matching the patterns $p_1, \ldots, p_n$.
  It is not required that the class instance is constructed directly by an invocation $C(v_1, \ldots, v_n)$. It is also possible that the value is an instance of a subclass of C, from where a super-call constructor invoked C's constructor with the given arguments. Another possibility is that the value was constructed through a secondary constructor, which in turn called the primary constructor with arguments $v_1, \ldots, v_n$. Thus, there is considerable flexibility for hiding constructor arguments from pattern matching.
– Variable binding patterns of the form $x@p$ where x is a variable and p is a pattern. Such a pattern matches the same values as p, and in addition binds the variable x to the matched value.

To distinguish variable patterns from named constants, we require that variables start with a lower-case letter whereas constants should start with an upper-case letter or special symbol. There exist ways to circumvent these restrictions: To treat a name starting with a lower-case letter as a constant, one can enclose it in back-quotes, as in **case** ‘x‘ ⇒ ... . To treat a name starting with an upper-case letter as a variable, one can use it in a variable binding pattern, as in **case** X @ _ ⇒ ....

```
case Tuple2(Cons(_,_), Cons(_,_)) ⇒ b1
case Tuple2(Cons(_,_), Nil) ⇒ b2
case Tuple2(Nil, Cons(_,_)) ⇒ b3
case Tuple2(Nil, Nil) ⇒ b4
```

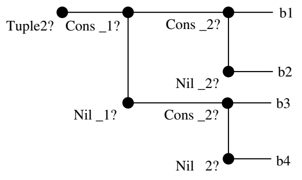

Fig. 6. Optimizing nested patterns

Optimizing Matching Expressions

A pattern match has usually several branches which each associate a pattern with a computation. For instance, a slightly more complete realistic simplification of arithmetic expressions might involve the following match:

```
t match {
  case Mul(Num(1), x) ⇒ simplify(x)
  case Mul(x, Num(1)) ⇒ simplify(x)
  case Mul(Num(0), x) ⇒ Num(0)
  case Mul(x, Num(0)) ⇒ Num(0)
  case _ ⇒ t
```

A possible implementation for this match would be to try patterns one by one. However, this strategy would not be very efficient, because the same type tests would be performed multiple times. Evidently, one needs to test not more than once whether t matches Mul, whether the left operand is a Num, and whether the right operand is a Num. The literature on pattern matching algebraic data types discusses identification and removal of superfluous tests [27, 11]. In a Java-like language, we additionally deal with subtyping [7].

The principle is shown in Fig. 6. After preprocessing, a group of nested patterns is expressed as a decision tree. A successful test leads to the right branch, where as a failing one proceeds down. If there is no down path, backtracking becomes necessary until we can move down again. If backtracking does not yield a down branch, the whole match expression fails with a MatchError.

A vertically connected line in the decision tree marks type tests on the same value (the *selector*). This can be implemented using type-test and type-case. However, a linear sequence of type tests could be inefficient: in matches with n cases, on average $n/2$ cases might fail. For this reason, we attach integer tags to case class and translates tests on the same selector to a lookup-switch. After having switched on a tag, only a constant number of type tests (typically one) is performed on the selector. We review this decision in the performance evaluation.

Examples of Case Classes

Case classes are ubiquitous in Scala's libraries. They express lists, streams, messages, symbols, documents, and XML data, to name just a few examples. Two groups of case classes are referred to in the following section. First, there are classes representing optional values:

```
trait Option[+T]
case class Some[T](value : T) extends Option[T]
case object None extends Option[Nothing]
```

Trait Option[T] represents optional values of type T. The subclass Some[T] represents a value which is present whereas the sub-object None represents a value which is absent. The '$+$' in the type parameter of Option indicates that optional values are covariant: if S is a subtype of T, then Option$[S]$ is a subtype of Option$[T]$. The type of None is Option[Nothing], where Nothing is the bottom in Scala's type hierarchy. Because of covariance, None thus conforms to every option type.

For the purposes of pattern matching, None is treated as a named constant, just as any other object. The **case** modifier for the object only changes some standard method implementations for None, as is explained in Section 4. A typical pattern match on an optional value would be written as follows.

```
v match {
  case Some(x) ⇒ "do something with x"
  case None ⇒ "handle missing value"
}
```

Option types are recommended in Scala as a safer alternative to **null**. Unlike the case with **null**, it is not possible to forget the test for the None case of an optional type.

Tuples are another group of standard case classes in Scala. All tuple classes are of the form:

```
case class Tuplei[T1, ..., Ti](_1 : T1, ..., _i : Ti)
```

There's also an abbreviated syntax using curly braces: $\{T_1, ..., T_i\}$ means the same as the tuple type Tuple$i[T_1, ..., T_i]$ and analogous abbreviations exist for expressions and patterns.

Evaluation: Pattern matching with case classes requires no notational overhead for the class hierarchy. The matching code is concise for shallow as well as for nested patterns. Case classes expose object representation. They have mixed characteristics with respect to extensibility. Adding new variants is straightforward. However, it is not possible to define new patterns, since patterns are in a one to one correspondence with case classes. This shortcoming is eliminated when case classes are paired with extractors.

4 Extractors

An extractor provides a way for defining a pattern without a case class. A simple example is the following object Twice which enables patterns of even numbers:

```
object Twice {
  def apply(x :Int) = x∗2
  def unapply(z :Int) = if(z%2==0) Some(z/2) else None
}
```

This object defines an apply function, which provides a new way to write integers: Twice(x) is now an alias for $x * 2$. Scala uniformly treats objects with apply methods as functions, inserting the call to apply implicitly. Thus, Twice(x) is really a shorthand for Twice.apply(x).

The unapply method in Twice reverses the construction in a pattern match. It tests its integer argument z. If z is even, it returns Some(z/2). If it is odd, it returns None. The unapply method is implicitly applied in a pattern match, as in the following example, which prints "42 is two times 21":

```
val x = Twice(21)
x match {
  case Twice(y) ⇒ Console.println(x+" is two times "+y)
  case _ ⇒ Console.println("x is odd") }
```

In this example, apply is called an *injection*, because it takes an argument and yields an element of a given type. unapply is called an *extraction*, because it extracts parts of the given type. Injections and extractions are often grouped together in one object, because then one can use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes. However, it is also possible to define an extraction in an object without a corresponding injection. The object itself is often called an *extractor*, independently of the fact whether it has an apply method or not.

It may be desirable to write injections and extractions that satisfy the equality F.unapply(F.apply(x)) == Some(x), but we do not require any such condition on user-defined methods. One is free to write extractions that have no associated injection or that can handle a wider range of data types.

Patterns referring to extractors look just like patterns referring to case classes, but they are implemented differently. Matching against an extractor pattern like Twice(x) involves a call to Twice.unapply(x), followed by a test of the resulting optional value. The code in the preceding example would thus be expanded as follows:

```
val x = Twice.apply(21)   // x = 42
Twice.unapply(x) match {
  case Some(y) ⇒ Console.println(x+" is two times "+y)
  case None ⇒ Console.println("x is odd")
}
```

Extractor patterns can also be defined with numbers of arguments different from one. A nullary pattern corresponds to an unapply method returning a boolean. A pattern with more than one element corresponds to an unapply method returning an optional tuple.

Pattern matching in Scala is loosely typed, in the sense that the type of a pattern does not restrict the set of legal types of the corresponding selector value. The same principle applies to extractor patterns. For instance, it would be possible to match a value of Scala's root type Any with the pattern Twice(y). In that case, the call to Twice.unapply(x) is preceded by a type test whether the argument x has type int. If x is not an int, the pattern match would fail without executing the unapply method of Twice. This choice is convenient, because it avoids many type tests in unapply methods which would otherwise be necessary. It is also crucial for a good treatment of parameterized class hierarchies, as will be explained in Section 6.

Representation Independence

Unlike case-classes, extractors can be used to hide data representations. As an example consider the following trait of complex numbers, implemented by case class Cart, which represents numbers by Cartesian coordinates.

```
trait Complex
case class Cart(re : double, im : double) extends Complex
```

Complex numbers can be constructed and decomposed using the syntax Cart(r, i). The following injector/extractor object provides an alternative access with polar coordinates:

```
object Polar {
  def apply(mod : double, arg : double): Complex =
    new Cart(mod ∗ Math.cos(arg), mod ∗ Math.sin(arg))

  def unapply(z : Complex): Option[{double, double}] = z match {
    case Cart(re, im) ⇒ Some{Math.sqrt(re∗re + im∗im), Math.atan(re/im)}
  }
}
```

With this definition, a client can now alternatively use polar coordinates such as Polar(m, e) in value construction and pattern matching.

Arithmetic Simplification Revisited

Figure 7 shows the arithmetic simplification example using extractors. The simplification rule is exactly the same as in Figure 5. But instead of case classes, we now define normal classes with one injector/extractor object per each class. The injections are not strictly necessary for this example; their purpose is to let one write constructors in the same way as for case classes.

Even though the class hierarchy is the same for extractors and case classes, there is an important difference regarding program evolution. A library interface might expose only the objects Num, Var, and Mul, but not the corresponding classes. That way, one can replace or modify any or all of the classes representing arithmetic expressions without affecting client code.

Note that every X.unapply extraction method takes an argument of the alternative type X, not the common type Term. This is possible because an implicit type test gets added when matching on a term. However, a programmer may choose to provide a type test himself:

```
  def unapply(x : Term) = x match {
    case m :Mul ⇒ Some {m.left, m.right}
    case _ ⇒ None
}
```

This removes the target type from the interface, more effectively hiding the underlying representation.

```
// Class hierarchy:
trait Term
class Num(val value: int) extends Term
class Var(val name: String) extends Term
class Mul(val left: Term, val right: Term) extends Term

object Num {
  def apply(value: int) = new Num(value)
  def unapply(n: Num) = Some(n.value)
}
object Var {
  def apply(name: String) = new Var(name)
  def unapply(v: Var) = Some(v.name)
}
object Mul {
  def apply(left: Term, right: Term) = new Mul(left, right)
  def unapply(m: Mul) = Some {m.left, m.right}
}
// Simplification rule:
  e match {
    case Mul(x, Num(1)) ⇒ x
    case _ ⇒ e
  }
```

Fig. 7. Expression simplification using extractors

Evaluation: Extractors require a relatively high notational overhead for framework construction, because extractor objects have to be defined alongside classes. The pattern matching itself is as concise as for case-classes, for both shallow and deep patterns. Extractors can maintain complete representation independence. They allow easy extensions by both new variants and new patterns.

Case Classes and Extractors

For the purposes of type-checking, a case class can be seen as syntactic sugar for a normal class together with an injector/extractor object. This is exemplified in Figure 8, where a syntactic desugaring of the following case class is shown:

```
case class Mul(left: Expr, right: Expr) extends Expr
```

Given a class C, the expansion adds accessor methods for all constructor parameters to C. It also provides specialized implementations of the methods equals, hashCode and toString inherited from class Object. Furthermore, the expansion defines an object with the same name as the class (Scala defines different name spaces for types and terms; so it is legal to use the same name for an object and a class). The object contains an injection method apply and an extraction method unapply. The injection method serves as a factory; it makes

```
class Mul(_left: Expr, _right: Expr) extends Expr {
  // Accessors for constructor arguments
  def left = _left
  def right = _right

  // Standard methods
  override def equals(other: Any) = other match {
    case m: Mul ⇒ left.equals(m.left) && right.equals(m.right)
    case _ ⇒ false
  }
  override def hashCode = hash(this.getClass, left.hashCode, right.hashCode)
  override def toString = "Mul("+left+", "+right+")"
}
object Mul {
  def apply(left: Expr, right: Expr) = new Mul(left, right)
  def unapply(m: Mul) = Some{m.left, m.right}
}
```

Fig. 8. Expansion of case class Mul

it possible to create objects of class C writing simply $C(\ldots)$ without a preceding **new**. The extraction method reverses the construction process. Given an argument of class C, it returns a tuple of all constructor parameters, wrapped in a Some.

However, in the current Scala implementation case classes are left unexpanded, so the above description is only conceptual. The current Scala implementation also compiles pattern matching over case classes into more efficient code than pattern matching using extractors. One reason for this is that different case classes are known not to overlap, i.e. given two patterns $C(\ldots)$ and $D(\ldots)$ where C and D are different case classes, we know that at most one of the patterns can match. The same cannot be assured for different extractors. Hence, case classes allow better factoring of multiple deep patterns.

5 Performance Evaluation

In this section, we measure relative performance of the presented approaches, using three micro-benchmarks. All benchmarks presented here were carried out on a Pentium 4 machine running Ubuntu GNU/Linux operating system using the HotSpot server VM and Sun's JDK 1.5 and the Scala distribution v2.3.1. They can be run on any runtime environment supported by the Scala compiler and are available on the first author's website. The BASE benchmark establishes how the techniques perform for a single pattern. The DEPTH benchmark shows how factoring out common cases affects performance. Finally, the BREADTH benchmarks tests. Since typecase is equivalent cast after translation, we do not include it in the benchmarks.

BASE Performance

We assess base performance by running the arithmetic simplification rule from the introduction. This benchmark measures execution time of $2 * 10^7$ successful matches, in milliseconds. The simplification is not applied recursively.

The results are given below. They are graphically represented in the left half of Fig. 9. We use the abbreviations oo for object-oriented decomposition, vis for visitor, cast for test-and-cast, ccls for case classes, ext for extractors returning tuples. Finally, ext+ shows extractors in a modified example where classes extend product interfaces, such that the extraction can return the same object and avoid constructing a tuple.

Discussion: No difference is observed between the object-oriented, test-and-cast and case class approaches. The visitor and the extractor approaches suffer from having to create new objects. In ext+, we diminish this penalty by making the data classes implement the Product2 interface and returning the same object instead of a tuple. In future work, we intend to reduce the overhead of extractors further by optimizing away the objects required for the use of option types.

The DEPTH Benchmark

When several patterns are tested side-by-side, a lot of time can be saved by factoring out common tests in nested patterns. If this is done by hand, the resulting code becomes hard to read and hard to maintain.

This benchmark measures execution time of 10^5 applications of several arithmetic simplification rules that are applied side-by-side and recursively. The results are graphed in the right side of Fig. 9.

Discussion: The ooNaive column shows that a readable, semantically equivalent program with redundant type tests can be 6 times slower than the hand-optimized oo version. But cast and ccls improve on both. Again, vis, ext and ext+ suffer from object construction. After factoring out common tests and removing unnecessary object constructions, ext+ is only twice as fast as ext and on a par with vis. We expect further improvements by optimizing option types.

The BREADTH Benchmark

Finally, we study how performance of a pattern match is related to the number of cases in a matching expression. For a fixed number n, the BREADTH benchmark defines n generated subclass variants and a matching expression that covers all cases. Applying the match 25000 times on each term of a list of 500 randomly generated terms yields the result (the terms and the order are the same for all approaches). The data is shown in Fig. 10 using two scales.

Discussion: Chained if-statements used by oodecomp fail $n/2$ on average. We also see that ooabstract that dispatching to a virtual method in an abstract class is faster than to one in an interface. The visitor approach vis is predictably unaffected by the number of cases, because it uses double-dispatch. Surprisingly, cast performs on a par with vis. It seems that HotSpot

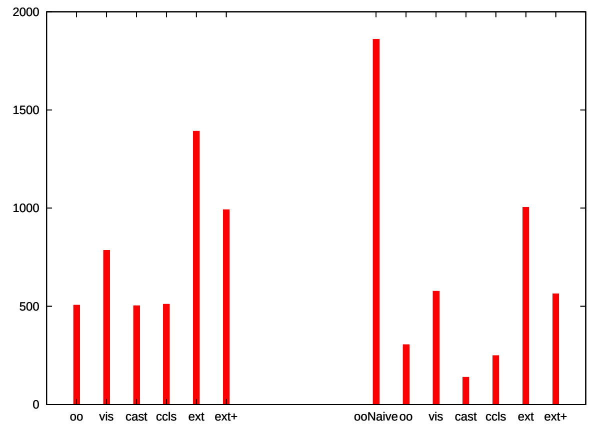

Fig. 9. Results on BASE and DEPTH benchmarks, in ms

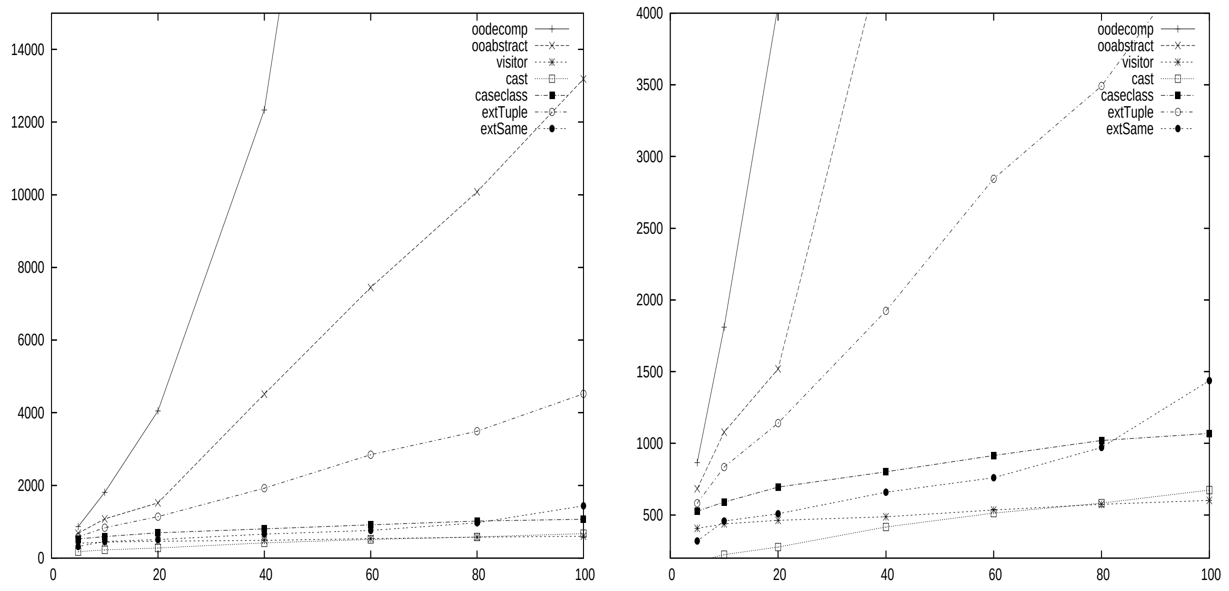

Fig. 10. Diagrams for BREADTH benchmark

recognizes sequences of type-tests as an idiom and translates it efficiently. Unaware of this VM optimization, we tried to achieve the same using integer tags caseclass on which we switch before doing the instanceOf. Extractions returning new tuples as results extTuple are affected negatively, but still outperform object-oriented decomposition. Finally, saving on object constructions in extSame achieves performance similar to caseclass.

Summary

The results show that the HotSpot VM is good at optimizing the output resulting from translation of pattern matching. While providing for much more readable code, case classes and unapply methods in long-running computations have performance that is not inferior to the performance of standard techniques. Hotspot optimizes sequences of type tests, which shows that we can reconsider our implementation strategy to take advantage of this fact.

6 Parametricity

Up to now, we have studied only hierarchies of monomorphic classes. The problem becomes more interesting once we consider classes with type parameters. An example is the typed evaluator for lambda expressions given in Figure 11.

There is an abstract base trait Term with subclasses Var for variables, Num for numbers, Lam for lambda abstractions, App for function applications, and Suc for a predefined successor function. The abstract base trait is now parameterized with the type of the term in question. The parameters of subclasses vary. For instance Var is itself generic with the same type parameter as Term, whereas Num is a Term of int, and Lam is a Term of b $\Rightarrow$ c where both b and c are type parameters of Lam.

The challenge is now how to write – in a statically type-safe way – an evaluation function that maps a term of type Term[a] and an environment to a value of type a. Similar questions have been explored in the context of "generalized algebraic data types" (GADT's) in functional languages such as Haskell [28] and Omega [23]. Kennedy and Russo [14] have introduced techniques to simulate GADT's in an extension of C# using visitors and equational constraints on parameters. We show here how GADT's can be simulated using typecase as the decomposition technique. This provides a new perspective on the essence of GADTs by characterizing them as a framework for exploiting type-overlaps. It goes beyond previous work by also providing a way to write updateable polymorphic functions. Such functions are used in several forms in denotational and operational semantics, for instance they can implement stores or environments.

Figure 11 shows an evaluation function eval which uses typecase for pattern matching its term argument t. The first observation from studying this function is that we need to generalize our previous concept of a typed pattern. Given a term of type Term[a] and pattern of form f: App[...], what type arguments should be provided? In fact, looking at a term's static type we can determine only the second type argument of App (which must be equal to a), but not the first one. The first type argument needs to be a fresh, completely undetermined type constant. We express this by extending the syntax in a type pattern:

A type pattern can now consist of types and *type variables*. As for normal patterns, we have the convention that type variables start with a lower-case letter whereas references to

```
//Class hierarchy
trait Term[a]
class Var[a]    (val name : String)                       extends Term[a]
class Num       (val value : int)                         extends Term[int]
class Lam[b, c] (val x : Var[b], val e : Term[c])         extends Term[b ⇒ c]
class App[b, c] (val f : Term[b ⇒ c], val e : Term[b])    extends Term[c]
class Suc       ()                                        extends Term[int ⇒ int]

// Environments :
abstract class Env {
  def apply[a](v : Var[a]): a

  def extend[a](v : Var[a], x : a) = new Env {
    def apply[b](w : Var[b]): b = w match {
      case _ : v.type ⇒ x          // v eq w, hence  a = b
      case _ ⇒ Env.this.apply(w)
}}}

object empty extends Env {
  def apply[a](x : Var[a]): a = throw new Error(”not found : ”+x.name) }

// Evaluation :
def eval[a](t : Term[a], env : Env): a = t match {
  case v : Var[b]     ⇒ env(v)                                   // a = b
  case n : Num        ⇒ n.value                                  // a = int
  case i : Suc        ⇒ { y : int ⇒ y + 1 }                      // a = int ⇒ int
  case f : Lam[b, c]  ⇒ { y : b ⇒ eval(f.e, env.extend(f.x, y)) } // a = b ⇒ c
  case a : App[b, c]  ⇒ eval(a.f, env)(eval(a.e, env))           // a = c
}
```

Fig. 11. Typed evaluation of simply-typed lambda calculus

existing types should start with an upper-case letter. (The primitive types int, char, boolean, etc are excepted from this rule; they are treated as type references, not variables).

Scala currently does not keep run-time type information beyond the top-level class. That is, it uses the same erasure module for generics as Java 1.5. Therefore, all type arguments in a pattern must be type variables. Normally, a type variable represents a fresh, unknown type, much like the type variable of an opened existential type. We enforce that the scope of such type variables does not escape a pattern matching clause. For instance, the following would be illegal:

```
def headOfAny(x : Any) = x match {
  case xs : List[a] ⇒ xs.head  // error : type variable ‘a’ escapes its scope as
}                              // part of the type of ‘xs.head’
```

The problem above can be cured by ascribing to the right-hand side of the case clause a weaker type, which does not mention the type variable. Example:

```
def headOfAny(x : Any): Any = x match {
  case xs : List[a] ⇒ xs.head  // OK, xs.head is inferred to have type ‘Any’, the
}                                    // explicitly given return type of 'headOfAny'
```

In the examples above, type variables in patterns were treated as fresh type constants. However, there are cases where the Scala type system is able to infer that a pattern-bound type variable is an alias for an existing type. An example is the first case in the eval function in Figure 11.

```
def eval[a](t : Term[a], env : Env): a = t match {
  case v : Var[b]     ⇒ env(v) ...
```

Here, the term t of type Term[a] is matched against the pattern v : Var[b]. From the class hierarchy, we know that Var extends Term with the same type argument, so we can deduce that b must be a type alias for a. It is essential to do so, because the right-hand side of the pattern has type b, whereas the expected result type of the eval function is a. Aliased type variables are also not subject to the scoping rules of fresh type variables, because they can always be replaced by their alias.

A symmetric situation is found in the next case of the eval function:

```
  case n : Num ⇒ n
```

Here, the type system deduces that the type parameter a of eval is an alias of int. It must be, because class Num extends Term[int], so if a was any other type but int, the Num pattern could not have matched the value t, which is of type Term[a]. Because a is now considered to be an alias of int, the right-hand side of the case can be shown to conform to eval's result type.

Why is such a reasoning sound? Here is the crucial point: the fact that a pattern matched a value tells us something about the type variables in the types of both. Specifically, it tells us that there is a non-null value which has both the static type of the selector and the static type of the pattern. In other words, the two types must *overlap*. Of course, in a concrete program run, the pattern might not match the selector value, so any deductions we can draw from type overlaps must be restricted to the pattern-matching case in question.

We now formalize this reasoning in the following algorithm **overlap-aliases**. Given two types t_1, t_2 which are known to overlap, the algorithm yields a set $\mathcal{E}$ of equations of the form $a = t$ where a is a type variable in t_1 or t_2 and t is a type.

Algorithm: overlap-aliases

The algorithm consists of two phases. In the first phase, a set of type equalities is computed. In the second phase, these equalities are rewritten to solved form, with only type variables on the left-hand side. We consider the following subset of Scala types:

1. Type variables or parameters, a.
2. Class types of form $p.C[\bar{t}]$. Here, p is a *path*, i.e. an immutable reference to some object, C names a class which is a member of the object denoted by p, and $\bar{t}$ is a (possibly empty) list of type arguments for C.
3. Singleton types of form $p.\mathbf{type}$ where p is a path. This type denotes the set of values consisting just of the object denoted by p.

Every type t has a set of basetypes denoted $\mathbf{basetypes}(t)$. This is the smallest set of types which includes t itself, and which satisfies the following closure conditions:

- if t is a type variable with upper bound u, $\mathbf{basetypes}(t) \subseteq \mathbf{basetypes}(u)$,
- if t is a singleton type $p.\mathbf{type}$, where p has type u, $\mathbf{basetypes}(t) \subseteq \mathbf{basetypes}(u)$,
- if t is a class type $p.C[\overline{u}]$, $\mathbf{basetypes}(t)$ includes all types in the transitive supertype relation of t [6].

The class extension rules of Scala ensure that the set of basetypes of a type is always finite. Furthermore, it is guaranteed that if $p.C[\overline{t}]$ and $q.C[\overline{u}]$ are both in the basetypes of some type t', then the two prefix paths are the same and corresponding type arguments are also the same, i.e. $p = q$ and $\overline{t} = \overline{u}$.

This property underlies the first phase of the algorithm, which computes an initial set of type equalities $\mathcal{E}$:

```
for − all $t$ of form $p.C[\overline{t}] \in \mathbf{basetypes}(t_1)$
    for − all $u$ of form $q.D[\overline{u}] \in \mathbf{basetypes}(t_2)$
        if $C = D$
            $\mathcal{E} \;:=\; \mathcal{E} \cup \{t = u\}$
```

The second phase repeatedly rewrites equalities in $\mathcal{E}$ with the following rules, until no more rules can be applied.

$$\begin{array}{rcll}
p.C[\overline{t}] = q.C[\overline{u}] & \longrightarrow & \{p = q\} \cup \{\overline{t} = \overline{u}\} & \\
p = q & \longrightarrow & t = u & \textbf{if } p : t, q : u \\
t = a & \longrightarrow & a = t & \textbf{if } t \text{ is not a type variable}
\end{array}$$

Note that intermediate results of the rewriting can be path equalities as well as type equalities. A path equality $p = q$ is subsequently eliminated by rewriting it to a type equality between the types of the two paths p and q. □

Returning to the type-safe evaluation example, consider the first clause in function eval. The type of the selector is Term[a], the type of the pattern is Var[b]. The basetypes of these two types have both an element with Term as the class; for Var[b] the basetype is Term[b], whereas for Term[a] it is Term[a] itself. Hence, the algorithm yields the equation Term[b] = Term[a] and by propagation b = a.

Now consider the second clause in function eval, where the type of the pattern is Num. A basetype of Num is Term[int], hence **overlap-aliases**(Num, Term[a]) yields the equation Term[int] = Term[a], and by propagation a = int.

As a third example, consider the final clause of eval, where the type of the pattern is App[b, c]. This type has Term[c] as a basetype, hence the invocation **overlap-aliases**(App[b, c], Term[a]) yields the equation Term[c] = Term[a], and by propagation c = a. By contrast, the variable b in the pattern rests unbound; that is, it constitutes a fresh type constant.

In each case, the overlap of the selector type and the pattern type gives us the correct constraints to be able to type-check the corresponding case clause. Hence, the type-safe evaluation function needs no type-cast other than the ones implied by the decomposing pattern matches.

Polymorphic updateable functions

The evaluator in question uses environments as functions which map lambda-bound variables to their types. In fact we believe it is the first type-safe evaluator to do so. Previous type-safe evaluators written in Haskell [28], Omega [23] and extended C# [14] used lambda expressions with DeBrujn numbers and represented environments as tuples rather than functions.

In Figure 11, environments are modeled by a class Env with an abstract polymorphic apply method. Since functions are represented in Scala as objects with apply methods, instances of this class are equivalent to polymorphic functions of type ∀a.Var[a] ⇒ a. Environments are built from an object empty representing an empty environment and a method extend which extends an environment by a new variable/value pair. Every environment has the form

empty.extend(v_1, x_1). ... .extend(v_n, x_n)

for $n \geq 0$, where each v_i is a variable of type Var$[T_i]$ and each x_i is a value of type T_i.

The empty object is easy to define; its apply method throws an exception every time it is called. The implementation of the extend method is more difficult, because it has to maintain the universal polymorphism of environments. Consider an extension env.extend(v, x), where v has type Var[a] and x has type a. What should the apply method of this extension be? The type of this method is ∀b.Var[b] ⇒ b. The idea is that apply compares its argument w (of type Var[b]) to the variable v. If the two are the same, the value to return is x. Otherwise the method delegates its task by calling the apply method of the outer environment env with the same argument. The first case is represented by the following case clause:

```
case _: v.type ⇒ x .
```

This clause matches a selector of type Var[b] against the singleton type v.**type**. The latter has Var[a] as a basetype, where a is the type parameter of the enclosing extend method. Hence, **overlap-aliases**(v.**type**, Var[b]) yields Var[a] = Var[b] and by propagation a = b. Therefore, the case clause's right hand side x of type a is compatible with the apply method's declared result type b. In other words, type-overlap together with singleton types lets us express the idea that if two references are the same, their types must be the same as well.

A pattern match with a singleton type $p.\mathbf{type}$ is implemented by comparing the selector value with the path p. The pattern matches if the two are equal. The comparison operation to be used for this test is reference equality (expressed in Scala as eq). If we had used user-definable equality instead (which is expressed in Scala as $==$ and which corresponds to Java's equals), the type system would become unsound. To see this, consider a definition of equals in some class which equates members of different classes. In that case, a succeeding pattern match does no longer imply that the selector type must overlap with the pattern type.

Parametric case-classes and extractors

Type overlaps also apply to the other two pattern matching constructs of Scala, case-classes and extractors. The techniques are essentially the same. A class constructor pattern $C(p_1, ..., p_m)$ for a class C with type parameters $a_1, \ldots, a_n$ is first treated as if it was a type pattern _: $C[a_1, \ldots a_n]$. Once that pattern is typed and aliases for the type variables

$a_1, \ldots, a_n$ are computed using algorithm **overlap-aliases**, the types of the component patterns $(p_1, ..., p_m)$ are computed recursively. Similarly, if the pattern $C(p_1, ..., p_n)$ refers to a extractor of form

```
object C {
  def unapply[a_1, ..., a_n](x : T) ...
  ...
} ,
```

it is treated as if it was the type pattern $\_ : T$. Note that T would normally contain type variables $a_1, \ldots, a_n$.

As an example, here is another version of the evaluation function of simply-typed lambda calculus, which assumes either a hierarchy of case-classes or extractors for every alternative (the formulation of eval is the same in each case).

```
def eval[a](t : Term[a], env : Env): a = t match {
  case v @ Var(name)    ⇒ env(v)
  case Num(value)       ⇒ value
  case Suc              ⇒ { y : int ⇒ y + 1 }
  case Lam(x : Var[b], e) ⇒ { y : b ⇒ eval(e, env.extend(x, y)) }
  case App(f, e)        ⇒ eval(f, env)(eval(e, env))
}
```

	oodecomp	visitor	test/cast	typecase	caseclass	extractor
Conciseness						
framework	–	–	+	+	+	–
shallow matches	o	–	–	+	+	+
deep matches	–	–	–	o	+	+
Maintainability						
representation independence	+	o	–	–	–	+
extensibility/variants	–	–	+	+	+	+
extensibility/patterns	+	–	–	–	–	+
Performance						
base case	+	o	+	+	+	–
scalability/breath	–	+	+	+	+	–
scalability/depth	–	o	+	+	+	–

Table 1. Evaluation summary

7 Conclusion

We described and evaluated six techniques for object-oriented pattern matching along nine criteria. The evaluations are summarized in Table 1. The table classifies each technique for each criterion in three broad bands. We should emphasize that this is more a snapshot than a definitive judgment of the different techniques. All evaluations come from a single language on a single platform, with two closely related implementations. Conciseness might vary for languages with a syntax different from Scala. Performance comparisons might be different on other platforms, in particular if there is no JIT compiler.

However, the evaluations can serve for validating Scala's constructs for pattern matching. They show that case classes and extractors together perform well in all of the criteria. That is, every criterion is satisfied by either case-classes or extractors, or both. What is more, case-classes and extractors work well together. One can conceptualize a case class as syntactic sugar for a normal class with an injector/extractor object, which is implemented in a particularly efficient way. One can also switch between case classes and extractors without affecting pattern-matching client code. The typecase construct plays also an important role as the type-theoretic foundation of pattern matching in the presence of parametricity. Extractor matching generally involves implicit pattern matches with type patterns. Typecase is thus useful as the basic machinery on which the higher-level constructs of case classes and extractors are built.

Acknowledgments We thank Craig Chambers, William Cook, Christian Plesner Hansen, and Urs Hölzle for discussions about previous pattern matching constructs, which led to the extractor design described here. We thank Don Syme for discussing the fine points of active patterns. Andrew Kennedy provided feedback on a previous version of the paper.

References

1. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Lisp and Functional Programming*, pages 182–192, June 1992.
2. Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Springer LNCS*, pages 33–56, 1992.
3. Brian Chin and Todd Millstein. Responders: Language Support for Interactive Applications. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2006.
4. Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, May 2006.
5. William Cook. Object-oriented programming versus abstract data types. In *REX Workshop on Foundations of Object-oriented Languages*, volume 489 of *Springer LNCS*, 1990.
6. Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *Proc. of Mathematical Foundations for Computer Science (MFCS)*, 2006.
7. Burak Emir. Translating pattern matching in a java-like language. In *Proc. of Third International Kyrgyz Conference on Electronics and Computer*, 2005.
8. Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: unified theory of dispatch. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Springer LNCS*, pages 186–211, 1998.
9. Martin Erwig. Active patterns. In *8th Int. Workshop on Implementation of Functional Languages*, volume 1268 of LNCS, pages 21–40, 1996.
10. Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1995.
11. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proc. of International Conference on Functional Programming*, pages 26–37, 2001.
12. Vladimir Gapeyev and Benjamin C. Pierce. Regular Object Types. In *proc:ecoop*, 2003.
13. Pedro Palao Gostanza, Ricardo Pena, and Manuel Manuel Nunez. A new look at pattern matching in abstract data types. In *Proc. of International Conference on Functional Programming*, 1996.

14. Andrew Kennedy and Claudio Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, 2005.
15. Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. of European Conference on Object-Oriented Programming (ECOOP), Springer LNCS 1445*, 1998.
16. Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: Object-oriented Pattern Matching for Evolvable Distributed Systems. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, 2003.
17. Jed Liu and Andrew C. Myers. Jmatch: Iterable Abstract Pattern Matching for Java. In *Proc 5th Internnational Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 110–127, 2003.
18. Todd Millstein. Practical predicate dispatch. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 245–364, 2004.
19. Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):836–889, September 2004.
20. Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *In G. Hedin, ed., 12th Conference on Compiler Construction, volume 2622 of LNCS*, pages 61–76, 2003.
21. Martin Oderksy and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. of Principles of Programming Languages (POPL)*, 1997.
22. Chris Okasaki. Views for standard ml. In *In SIGPLAN Workshop on ML, pages 14-23*, 1998.
23. Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Proc. GPCE*, October 2004.
24. Don Syme. `http://blogs.msdn.com/dsyme/archive/2006/08/16/activepatterns.aspx`, August 2006.
25. Mads Torgersen. The expression problem revisited. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 3086, 2004.
26. Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. of Principles of Programming Languages (POPL)*, 1987.
27. Philip Wadler. *Pattern Matching*, chapter 4. Prentice Hall, 1987.
28. Stephanie Weirich. A statically type-safe typechecker for haskell. unpublished manuscript, Dagstuhl seminar, September 2004.
29. John Williams. `http://article.gmane.org/gmane.comp.lang.scala/1993`, April 2006.
30. Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proc. of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.
31. Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc ICFP*, 2001.
32. Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Workshop on Foundations of Object-Oriented Languages*, 2005.