Seminar Program Analysis and Transformation
University of Applied Sciences Rapperswil

Pattern Matching in Scala

Michael Rüegg, mrueegg@hsr.ch

December 18, 2009

Abstract. Pattern matching on strings is widely known and well supported by a lot of programming languages. But there are more possibilities with pattern matching than just to match strings. Functional programming languages traditionally have a very strong support for generic pattern matching. Scala, a programming language deeply influenced by functional programming, provides rich assistance for pattern matching with objects through its concepts of case classes and extractors. This paper shows the theory behind these concepts and how they are applied in real-world examples. In the last part of this paper pattern matching in other languages like Haskell, Erlang, F#, XSLT and Prolog is presented and compared to Scala's implementation.

1 Introduction

Most people associate the term pattern matching with matching strings with the help of regular expressions known from languages like Python or Perl. But there is a lot more that can be achieved with pattern matching. Generally speaking, pattern matching is a technique for assigning names to things and decomposing data structures and objects with a known structure into its underlying parts. Pattern matching has a strong background in functional programming languages. Scala's pattern matching is greatly influenced by these.

Scala is a statically and strongly typed multi-paradigm programming language which name stands for "scalable language". It has its roots at the École Polytechnique Fédérale de Lausanne (EPFL) where it has been designed by a team around Prof. Martin Odersky. Scala integrates features of both object-oriented and functional programming and generates bytecode that runs on the Java Virtual Machine (JVM).

The aim of this paper is to show how pattern matching is used in the Scala programming language. Furthermore it compares its implementation with the ones of various other famous programming languages. It is structured as follows: The rest of this section shows an example of pattern matching in Scala and how nice mathematical function descriptions can be translated into code with its help as well as a short analysis of how pattern matching is used in other object-oriented languages. The main part of this paper is contained in section 2 where pattern matching in Scala is introduced with its two main concepts case classes and extractors which are explained in detail with various examples. Section 3 presents pattern matching in other programming languages like Haskell, Erlang, F#, XSLT and Prolog and their commonalities and differences to Scala. Finally section 4 tries to outline the strengths and weaknesses of Scala's implementation of pattern matching and section 5 concludes.

1.1 Motivation

Pattern matching has always been a very important concept in the functional programming world. The reason for its wide application gets more obvious through an example. The following Scala code computes the Fibonacci numbers recursively with the help of pattern matching:

```
def fib(n: Int) : Int = n match {
  case 0 => 0
  case 1 => 1
  case _ => fib(n-1) + fib(n-2)
}
```

This pattern matching code has a very similar structure to the recursive mathematical definition of the Fibonacci sequence:
$f_n = f_{n-1} + f_{n-2}, f_0 = 0, f_1 = 1$
As an analogy to mathematics Richard and Lhotak [1] say that pattern matching is akin to giving an equation to a calculator and letting it solve for the variables instead of solving it by the mathematician itself.
Although it might seem that pattern matching is only some kind of syntactical sugar for an enhanced switch statement, e.g., known from C, pattern matching is much more powerful. In general it is not limited to only test constants. It is also used for matching objects with patterns which are structured and contain variables themselves. Furthermore these patterns can contain guards for addi-

tional matching rules, they can be nested and there even exists algorithms and implementations to support the programmer with checks as Maranget discusses in [2]. These checks can assure exhaustiveness, i.e., that no cases are missed, and can help to avoid useless checks (i.e., cases that are impossible to become true). In this paper all these features will be examined very detailed.

1.2 Pattern matching in object-oriented programming languages

While pattern matching is a fundamental paradigm in functional programming languages, it is not common in object-oriented programming languages so far. Pattern matching in functional programming languages is based on the concept of algebraic data types (ADT[1]). One of the reasons that pattern matching has not been applied more in object-oriented programming might come from the fact that there are a lot of arguments against ADT's like, e.g., as Emir writes in his thesis [4], their lacking extensibility and representation independence. Another reason is the often stated paradigm mismatch between the object-oriented motto of grouping data and operations together whereas pattern matching is more useful when these are separated.

In the past, several projects tried to bring pattern matching into the object-oriented world. JMatch [5], a Java language extension, supports iterable abstract pattern matching and has ML-style deep pattern matching [6]. TOM [7] provides pattern matching for languages like C, Java and Eiffel and is designed to manipulate tree structures and XML documents [8]. Another language extension for Java is HydroJ, which is targeted to message exchange in distributed systems and uses semi-structured messages and pattern matching for handling these to gain robustness against changes in the underlying data formats [9]. There is also a project similar to JMatch for C++ called Prop [10]. Prop offers ML-style algebraic data types and pattern matching for supporting the development of interpreters, compilers and language translation and transformation tools [11].

Scala now combines functional pattern matching and ADT's in a new way with object-oriented paradigms as a fundamental principle. Its concept of case classes turns ADT's into real object-oriented classes and with extractors Scala offers a possibility to define new patterns without the need to always provide case classes. Both concepts and their application in pattern matching are described in detail in the next section.

[1] A new type in functional programming languages is defined through the help of one or more type constructors. Such a type is known as an algebraic data type. In Haskell one can create an algebraic data type for shapes like this [3]: `data Shape = Circle Float | Rect Float Float`

2 Pattern Matching in Scala

This section describes how pattern matching in Scala works. To accomplish this the basic syntax of pattern matching will be shown first and then its two main use cases: case classes and extractors. Furthermore an overview is given about the various applications of pattern matching in Scala's internals.

2.1 The pattern matching syntax

Basically a pattern match in Scala has the following syntax [12]:

```
selector match { case pattern => expression }
```

selector is the expression a *pattern* is matched against. The patterns are separated from the expressions through an arrow symbol ⇒. The cases are evaluated in order they appear in the pattern match. Therefore, if the preceding case is broader in scope than the following one, the latter will never be reached which causes an `unreachable code` compile error.

Similar to the famous *default* case in conventional switch statements there is a "catch all" clause denoted with the underscore _. In case no pattern matches a selector, the match statement will throw a `MatchError`. Appendix A shows the pattern syntax in very detail as it is described in the Scala language specification [13].

2.2 Case Classes

As Odersky et al. state in [12], with the help of case classes Scala allows pattern matching on objects without a lot of boilerplate code. A class is said to be a *case class* when its class definition is prefixed with `case`. Listing 1 [14] shows a class hierarchy for a binary search tree built with case classes.

```
abstract class Tree[+T]
case object Leaf extends Tree
case class Node[T](elem: T, left: Tree[T],
                   right: Tree[T]) extends Tree[T]
```

Listing 1: *Class hierarchy for binary search trees*

The hierarchy has an abstract base class `Tree` with two subclasses as the building blocks of the tree. `Leaf` is a case object and not a case class because there is only one instance of it, so a *Singleton* as described in [15] perfectly fits here. `Node` consists of an element `elem` and two subtrees `left` and `right`.

Case classes provide several useful features out of the box. First, all constructor parameters become immutable (as it is described in the Scala language specification [13], a `val` prefix is implicitly added unless the parameter already carries a `val` or `var` attribute) and accessors for the arguments are generated. Second, the Scala compiler automatically provides implementations of `equals`, `hashCode`

and `toString`[2]. Third, a companion object[3] with a method named `apply` and an extractor based on the constructor fields are declared.

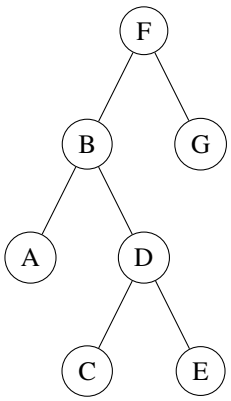

Figure 1: *Binary search tree for case class example*

As an example for the application of case classes one can build the binary search tree of figure 1 with the help of the defined case classes in listing 1 as follows:

```
val t = Node("F", Node("B", Node("A", Leaf, Leaf),
                            Node("D",
                               Node("C", Leaf, Leaf),
                               Node("E", Leaf, Leaf))),
                      Node("G", Leaf, Leaf))
```

Consider that there is no need to specify `new` for the object creation of case classes. This is implicitly achieved through the method `apply`. This method plays the role of a factory method. It will be further discussed in section 2.8 together with its counterpart `unapply`.

Case classes are used heavily in Scala. One very important application of case classes are optional values. They are the recommended and safer way to express and replace `null` in Java. Listing 2 shows how they are defined.

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

Listing 2: *Optional values with case classes*

An optional value in Scala is represented by the type `Option`. This type can be of two forms: Either it has the form `Some(x)` where *x* is the actual value or it is the `None` object (notice again the *Singleton*). Latter stands for a missing value. As an example of their application and to show that no checks for `null` are necessary anymore, see listing 3. Compare this solution to the inconvenient and error-prone checks that would have been necessary in a language like Java [12].

[2]With Scala version 2.8 a copy method will also be provided which allows to generate a new instance of a case class identical to an already existing one except for a few different fields (see [16]).

[3]A companion object of a class is the singleton that has the same as the class.

```
def annualRise(salary: Option[BigDecimal])
                                    = salary match {
  case Some(salary) => "$" + (salary * 1.1)
  case None         => "?"
}

val salaries = Map("Alice" -> BigDecimal("5700"),
                   "Bob"   -> BigDecimal("4500"))
annualRise(salaries get "Bob") // => $4950
annualRise(salaries get "Carol") // => ?
```

Listing 3: *Example with optional values*

With the help of the optional type `Option[BigDecimal]` the method `annualRise` clearly states that it is also able to handle a missing `salary` value (i.e., `None`). The use of optional types tends to a better and more meaningful contract of the method. Even more important is the fact that one also gains type safety at compile time. `annualRise` expects as a parameter `Option[BigDecimal]`. In case a variable of type `BigDecimal` is passed to it, the compiler will complain [12]. Notice that, as with `Map` in this example, a lot of data structure methods in Scala produce optional values.

2.3 Constructor Patterns

Scala not only provides pattern matching with constants as seen in the Fibonacci example, it also has constructor patterns. Listing 4 [14] shows an implementation of an in-order tree traversal that uses constructor patterns and the case classes described in listing 1.

```
def inOrder[T](t: Tree[T]): List[T] = t match {
  case Leaf => List()
  case Node(e, l, r) =>
    inOrder(l):::List(e):::inOrder(r)
}
//Result: List(A,B,C,D,E,F,G) with input on p. 2
```

Listing 4: *In-order tree traversal with constructor patterns*

Constructor patterns let one define so called *deep matches*. As an example consider listing 5 which presents the use of constructor patterns in Swing event handling code. One can react to events in Scala through adding so called *handlers* to the `reaction` property of a Swing component. These handlers are functions which use pattern matching to react according to the generated event. When the pattern in the handler for table events is evaluated, then not only will be checked that the object is a member of the named case class (i.e., `TableUpdated`), but also if the contents of the object match against the supplied patterns. In the presented example the argument `column` is verified to be equal 1.

```
// Case class definition for table update events
case class TableUpdated(override val source:Table,
                        range: Range, column: Int)
           extends TableChange(source)

// GUI table event handling code
```

```
val salaryTable = new Table(200, 100) {
  reactions += {
    case TableUpdated(salaryTable, rows, 1) =>
      // handle table update event in column 1
    case TableUpdated(salaryTable, rows, cols)
      if cols % 2 == 0 =>
      // ... only on even column numbers
  }
}
```

Listing 5: *Deep matches with Swing event handling*

As one can see from the second case, Scala also allows to specify conditions after the pattern expression. These conditions are called *pattern guards*. With the given condition only table updates for even column numbers would be handled. Note that the Scala language specification [13] says that a guard expression is only evaluated in case the pattern matches the selector.

2.4 Typed Patterns

Through the help of typed patterns it is possible to avoid type tests and type casts. A typed pattern has the form *x: T* and consists of a pattern variable *x* and a type pattern *T* [13]. An application area of typed patterns can be found in Scala's exception handling:

```
try {
  file = new FileReader(filePath)
} catch {
  case e: FileNotFoundException =>
    // Handle missing file
  case e: IOException =>
    // Handle other I/O error
  case e => e.printStackTrace()
} finally { // cleanup resources }
```

In this example the variable binding is done for `e`. Note that compared to Java only *one* catch clause for handling several types of exceptions is needed.

2.5 Sequence Patterns

In Scala it is also possible to match against sequence types like lists and arrays. For matching sequences Scala provides a special syntax which helps to deal with list and array elements in a way that is very common in the functional programming world. The basic pattern looks like this:

```
case x :: xs
```

x is bound to the first element of the list (*head*), then comes Scala's cons operator *::* [12] followed by *xs* which matches the remainder of the list (*tail*) and - as in functional programming - it is common to refer to this with an ending *s* (for *sequence*). This schema allows a convenient implementation of recursive list processing. Listing 6 shows an example of list processing for a method that computes the product of a number sequence.

```
def fac(ns: List[Int]) : Int = ns match {
  case List()  => 1
  case n :: ns => n * fac(ns)
}
```

Listing 6: *Sequence patterns with list processing*

In this example the pattern `n :: ns`[4] permits the recursive definition of factorials to be defined as common whereas `List()` is the base case of the recursion which distinguishes the empty list.

2.6 Wildcards and Variable Binding

Wildcards in pattern matching are useful to denote a "catch-all case". A wildcard is expressed as an underscore _ and matches *any* object. Variable binding allows to associate a variable name to any pattern. The syntax consists of the variable name, an at sign `@` followed by the pattern. Listing 7 shows these two concepts and highlights another application of pattern matching in Scala: pattern matching with XML. As a possible XML input document consider the book list XML in appendix B.

```
def processAuthor(node: scala.xml.Node): String =
  node match {
    case <author>{authorname @ _*}</author> =>
      authorname.text
    case _ =>
      // something else
  }
}
```

Listing 7: *Pattern matching with XML*

`processBook` takes a `Node` as an argument and matches it against two patterns. The first one shows that a pattern definition can contain XML tags. `_*` is known as a *sequence wildcard*. In this context the sequence wildcard simply means to match any sequence of XML nodes [12]. As one can think the Kleene star also matches the empty sequence. Therefore this pattern matches both <*author*> elements with arbitrary XML content (as the book of Bertrand Meyer in the mentioned example in appendix B with its nested XML nodes for the first and last name) and the empty tag (<*author/*>). Consider the application of the variable binding here which is used to return the name of the book author as a value of the method. The second pattern is a wildcard pattern and matches everything else.

2.7 Safe Pattern Matching with Sealed Classes

Scala offers special support for safe and robust pattern matching. Assume that one has implemented an expression evaluator with a hierarchy of case classes as shown in Listing 8.

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Div(n: Expr, d: Expr) extends Expr

def evaluate(e: Expr): Int = e match {
  case Num(n)    => n
  case Div(n, d) => evaluate(n) / evaluate(d)
}
```

Listing 8: *Expression evaluator*

[4] Note that this is a special case of an infix operation pattern. It is treated as `::(n, ns)` which uses the class `scala.::` as a pattern constructor.

Now imagine one would like to expand the expression evaluator with another expression type *Mult*:

```
case class Mult(f: Expr, f: Expr) extends Expr
```

It is possible that one might forget to adapt the pattern matching code and to handle the new expression type with an appropriate pattern in the match expression. This is especially true when there are many places in the code base where these expressions are used in pattern matching. The missing patterns would lead to a `MatchError` at runtime when one calls the method `evaluate` with an expression argument of the new type.
Scala provides a way to detect missing patterns in match expressions at compile-time. This can be achieved by defining the superclass of the case classes as *sealed*:

```
sealed abstract class Expr
```

With the added keyword `sealed` in front of the superclass definition, the Scala compiler will now generate a warning with a message like *"Match is not exhaustive! missing combination Mult"*. The keyword tells the compiler that the only subclasses of the superclass `Expr` will be defined in the same source file because it is not allowed to directly inherit from a sealed superclass outside of its containing file[5].
With sealed case classes Scala provides a safety net which is not that easily to achieve with standard object-oriented techniques like type-tests. One could also say that with pattern matching on objects in combination with sealed case classes there is no need for a default case anymore. This is because one doesn't have to deal with unknown defaults when the compiler does the hard work. Note also that the option type presented in section 2.2 was sealed too.

2.8 Extractors

So far the pattern matching on objects in this paper was entirely based on case classes. Scala provides a complementary and often more flexible way to do pattern matching with a concept called *extractor*. An extractor in Scala is defined in the language specification [13] as an object which has a method `unapply` or `unapplySeq` that matches the pattern. With an extractor one can create patterns without the need to provide case classes.

2.8.1 Example of use

As an introductory example, listing 9 shows an extractor object for 10-digit ISBN numbers.

```
object ISBN {
  def apply(cntryCode: String, publCode: String,
            titleCode: String, chkDigit: String) =
    cntryCode + "-" + publCode + "-" +
      titleCode + "-" + chkDigit

  def unapply(s: String): Option[(String,String,
                                  String,String)] = {
    val parts = s split "-"
    if (parts.length == 4)
      Some(parts(0), parts(1), parts(2), parts(3))
    else None
  }
}
```

Listing 9: *ISBN number extractor object*

Even though it is not necessary for an extractor, `ISBN` defines a method `apply` as a shorthand for object creation similar as seen before with case classes. An object creation like `ISBN(s1,s2,s3,s4)` gets translated by the Scala compiler into `ISBN.apply(s1,s2,s3,s4)`. Through `apply`, the extractor object can be used like a *function object*. `apply` is called an *injection* because it takes arguments and returns an element of the involved type [4].
As mentioned before, the method `unapply` is the important property of an extractor. This method is what makes `ISBN` an extractor object. `unapply` can be looked at as an *inverse* function object. Mathematically one can define it as a function f in a object x which takes exactly one argument, matches a pattern $x(p_1, .., p_n)$ with n parts and the following applies [13]:

$$f(x) = \begin{cases} Boolean & \text{if} \quad n = 0; \\ Option[T] & \text{if} \quad n = 1; \\ Option[(T_1, .., T_n)] & \text{if} \quad n > 1. \end{cases}$$

`unapply` is called *extraction* [4] because it extracts parts of the type. As it reverses the construction in a pattern match, it gets called implicitly when the extractor object is used in a pattern match:

```
val isbn = "0-201-10194-7"
isbn match {
  case ISBN("0", "201", "10194", "7") =>
    // "Red Dragon Book" by Aho et al.
}
```

As the pattern match involves a pattern related to an extractor object, the preceding code leads to the following method invocation:

```
ISBN.unapply("0-201-10194-7")
```

This method invocation then returns with the given ISBN number in `isbn`:

```
Some("0", "201", "10194", "7")
```

In case the ISBN number would not match, `None` would be returned. This fact is reflected in the return type of the method `ISBN.unapply` which is an `Option` type. Note that `ISBN` offers the duality

```
ISBN.unapply(ISBN.apply(isbn)) ==
    Some(cntry, publ, title, chkDigit)
ISBN.unapply(isbn) match {
  case Some(cntry, publ, title, chkDigit) =>
    ISBN.apply(cntry, publ, title, chkDigit)
}
```

which is not necessary, but is considered as a good design principle in Scala [12].

[5] But note that subclasses of sealed classes can be inherited from everywhere [13].

2.8.2 Variable argument extractors

As stated in the mathematical definition of `unapply`, Scala also supports extractors with a variable number of arguments. Let's suppose the definition of the extractor object `ISBN` (10-digit number, 4 string parts) now should also support its successor standard with 13-digit ISBN numbers (5 string parts). To achieve this, `unapplySeq` could be used to return a sequence of strings:

```
def apply(parts: String*): String  =
  parts.mkString("-")

def unapplySeq(whole: String):
    Option[Seq[String]] = { // validation omitted
  Some(whole.split("-"))
}
```

Now it is also possible to use the wildcard sequence pattern `_*` to match only against a part of the whole ISBN number:

```
isbn match {
  case ISBN("0", "201", _*) =>
    // English Book by Addison-Wesley Longman
}
```

2.8.3 Representation Independence

Case classes have a fixed relationship between the selector object (i.e., data representation) and the constructor pattern. Consider the following case of a pattern matching implementation for temperature scales with case classes:

```
trait Temp
case class Fahrenheit(d: Double) extends Temp
```

Whenever one uses `case Fahrenheit(...)` and the match succeeds, it is immediately clear that the object in the selector statement is of type `Fahrenheit`. This is what makes cases classes expose their internal representation.

In the example shown before with the extractor object `ISBN`, the representation of the ISBN number was a string compared to if one would have used case classes. The underlying data representation could be changed (e.g., from ISBN 10-digit numbers to 13-digit ones) without affecting client code. Emir calls this property extractors have compared to case classes in his thesis [4] *representation independence*.

Assume now that the writer of the temperature scale implementation would like to allow clients to use additional scales than Fahrenheit (e.g., Celsius) that they should be able to use in pattern matching code. Instead of providing another case class for every supported temperature scale one could implement extractor objects like `Celsius` as seen in listing 10.

```
object Celsius {
  def apply(d: Double): Temp =
    Fahrenheit(d * (9.0/5.0) + 32)

  def unapply(t:Temp):Option[(Double)] = t match {
    case Fahrenheit(d) => Some((d-32) * (5.0/9.0))
  }
}
```

Listing 10: *Extractor object for Celsius temperature*

The implicit conversion between the different data types (i.e., temperature scales) allows clients to use the types they would like in their pattern matching code. As an example the method `isBoilingPointOfWater` shows the use of the temperature scale Celsius:

```
def isBoilingPointOfWater(t: Temp) = t match {
  case Celsius(100.0) => true
  case _              => false
}
```

The library provider can now introduce new data types or change the underlying representation of the existing ones without breaking the client code as long as the extractors will be adapted accordingly. Extractors and their support of different data types and the implicit conversion between them are related to Walder's concept of *views* [17].

Although extractors have this advantage in loose coupling that comes with representation independence, it is not possible with them to benefit from safe pattern matching the compiler offers with case classes as described in section 2.7. Therefore notice that the method `isBoilingPointOfWater` in the code example before had to provide a default case.

2.8.4 Regular Expressions

An important application of extractors in Scala are regular expressions. A regular expression for parsing Swiss IBAN numbers[6] can be defined as follows:

```
val swissIBAN = """CH(\d{2})\s?(\d{4}\s?\d)
                   (\d{3}\s?\d{4}\s?\d{4}\s?\d)""".r
```

This code example uses Scala's convenient way to define regular expressions with an appended `.r`. The method `r` in the class `scala.runtime.RichString` returns an instance of class `scala.util.Regex`. In Scala every regular expression defines an extractor [12]. This can be used to match the groups of the IBAN with the help of the regular expression:

```
val swissIBAN(chkDigit, bankCode, accNumber) =
  "CH93 0076 2011 6238 5295 7"
```

What happens here is that the `unapplySeq` method of class `Regex` is invoked which returns a list of strings in case the pattern matches and `None` otherwise. Scala even lets one use these extractors in *for* expressions:

```
val in = """Transfer $1000000 from
 CH9300762011623852957 to CH9301762011123852444"""

for (swissIBAN(k,b,c) <- swissIBAN findAllIn in)
  println("checkDigit k:" + k + ", bankCode B:" +
          b + ", accountNumber C:" + c)
```

2.9 Parser combinators

Pattern matching in Scala's internals is ubiquitous. As illustrated in this paper so far pattern matching

[6] Definition of Switzerland IBAN format: CHkk BBBB BCCC CCCC CCCC C, k=control digit, B=bank clearing, C=account No., spaces between elements are optional (see [18]).

is used among others in Scala's event and exception handling, in processing of XML, etc. But there is another place where pattern matching is also very valuable: In the implementation of parsers with Scala's concept of *parser combinators*. In short parser combinators form an *internal domain specific language* (internal DSL as Fowler's definition in [19]) which is used to build a very strong relationship between a grammar of a language in *Extended Backus-Naur Form* (EBNF) and the corresponding parsing code. Whereas case classes can be used for a representation of an *abstract syntax tree* (AST) and pattern matching for its interpretation (i.e., evaluation), pattern matching is also used in parser combinators to build that AST.

Consider again the example in listing 8 where case classes are used for a very simple expression evaluator. The next logical step in building a small calculator language (in fact also a DSL) would be a parser that takes a string with an expression and transforms it into an AST representing that expression. An example of how this can be accomplished is shown in the following code extract:

```
def term: Parser[Expr] =
  (factor ~ "/" ~ factor) ^^ {
    case lhs~div~rhs => Div(lhs, rhs)
  } | factor
```

The definition in this code extract represents a production of a grammar for arithmetic expressions which could be defined as `term ::= '(' factor '/' factor ')' | factor`. This production states that a term is a factor which can further be divided by other factors [12]. Note that `~` is used to express sequential composition. `^^` transforms the result of the parser and hands it over to a function that uses pattern matching to build the corresponding object of the case class `Div`.

As one can see pattern matching is very useful to decompose the results of the parsing process. The relationship between pattern matching in parser combinators and case classes as the building blocks of AST's is very fundamental. Of course parser combinators are much more powerful than presented here. More information can be found, e.g., in an article series of Ted Neward [20] where he shows a fully-fledged example of building a calculator language similar to the extract shown here.

2.10 Implementation of Pattern Matching

The purpose of this section is not to give an exhaustive overview how pattern matching is implemented in Scala. This can be found in very detail in Emirs thesis [4]. Rather its intend is to show the underlying implementation of a pattern match with an example to strengthen the understanding of pattern matching in Scala.

To show which underlying language constructs are used when one uses pattern matching listing 11 shows the internal implementation[7] of the in-order tree traversal method presented in section 2.3.

```
def inOrder(t: Tree): List = {
  val temp8: Tree = t;

  if (Leaf.==(temp8))
    scala.this.Nil
  else if (temp8.$isInstanceOf[Node]()) {
    val temp10: Node = temp8.$asInstanceOf[Node]();
    val temp11: java.lang.Object = temp10.elem();
    val temp12: Tree = temp10.left();
    val e: java.lang.Object = temp11;
    val l: Tree = temp12;
    val x$2: List = inOrder(l);
    val x$1: List = scala.List.apply(e);
    inOrder(temp10.right()).:::(x$1)}.:::(x$2)
  else
    throw new MatchError(temp8)
};
```

Listing 11: *Internal implementation of the in-order tree traversal method*

As one can see Scala uses type tests through the method `isInstanceOf` and type casts with `asInstanceOf` internally to support pattern matching on objects. Furthermore it's shown here that in case no pattern matches Scala would throw a `MatchError`.

3 Related Work

SNOBOL4 was one of the first programming languages that supported pattern matching as a built-in language construct. As Griswold et al. describe in [22] SNOBOL4 allows it to build complex patterns through alternation and concatenation. According to Scott [23] ML's pattern matching differs from SNOBOL4 in its integration with static typing and type interference. It greatly influenced pattern matching in other functional programming languages like OCaml, Erlang and Haskell. With its concept of *constructors* it offers convenient pattern matching on composite types like trees. Furthermore, ML implementations have compile-time checks for verifying if a pattern match can succeed or even if it's sure that it will fail [23].

The following sections will give an overview how pattern matching is supported as well as differences and commonalities to Scala for the following programming languages: Haskell, Erlang, F#, XSLT and Prolog.

3.1 Haskell

Pattern matching in Haskell [24] is well supported. The Fibonacci sequence (see introduction example in Scala in section 1.1) can be defined in Haskell as follows:

```
fib  :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

[7] Generated with the *scalac* compiler option *-print* which basically prints the program with all Scala-specific features removed (see [21] for more information). For easier reading certain details in this output have been omitted by the author.

Haskell supports a special kind of pattern called *integer pattern*. Integer patterns take the form `n+k`, where `n` stands for an integer variable and `k>0` is a constant. As an example [3] consider the following function `pred` which uses an integer pattern to map any positive integer to its predecessor.

```
pred      :: Int -> Int
pred (n+1) = n
```

As elementary for functional programming languages in general, Haskell also provides well supported pattern matching with lists. Similar to Scala, in Haskell one specifies the empty list with *[]* and *x:xs* implies any list with at least one element, where *x* is bound to the first element and *xs* to the rest of the list. Consider the following function `take` which returns the first *n* elements of a list:

```
take           :: (Num t) => t -> [a] -> [a]
take 0   _      = []
take _   []     = []
take n   (x:xs) = x : take (n-1) xs
```

As seen in the previous example, Haskell also allows *wildcards* in pattern matching which - as in Scala - match everything. There is a further commonality in Haskell to Scala with the concept of *guards*. Guards are often used in pattern matching because they provide an elegant way of expressing certain conditions that must apply for a successful match. The following function [3] uses a guard to test if its argument is not equal to 0:

```
isZero           :: (Num t) => t -> Bool
isZero 0          = True
isZero n | n /= 0 = False
```

`isZero` is an example of a function that uses patterns which are *disjoint* because it does not matter in which order they are matched. Notice that this would not be the case if the guard would be omitted. This principle is a best practice that should be followed whenever possible.

There is a related concept to Scala's case classes in Haskell called *constructors*. Consider the following recursive data type definition for a binary tree similar to the one shown in section 2.2:

```
data Tree t = Node t (Tree t) (Tree t) | Leaf
```

In this example `Tree` is a type constructor whereas `Node` and `Leaf` are data constructors. Note that the type definition is parametrized through `t`. In Haskell one can do pattern matching with these constructors. The following example shows how the depth of a binary tree can be calculated with the help of pattern matching:

```
depth             :: (Tree t) -> Int
depth Leaf         = 0
depth (Node v l r) = max (depth l) (depth r) + 1
```

Haskell also has a special kind of patterns called *lazy patterns*. A lazy pattern is defined with the syntax form ∼pat. Lazy patterns have a property called *irrefutability*. This means that a match against a value *val* is always successful, regardless of how *pat* is defined [25]. In the case of the tree depth example before if the pattern would be defined like `~(Node v l r)` then, e.g., `v` would only be bound to a matching value if it is used on the right-hand-side of the definition. This is in contrast to "normal" patterns where the binding is done *before* during the match process. The most common application of lazy patterns is with recursively infinite data structures [25].

3.2 Erlang

Pattern matching is at the core of Erlang [26]. Consider the following example [27] which shows a nice application of pattern matching in a function that produces the area of certain geometric figures:

```
-module(geometry).
-export([area/1]).

area({rectangle, Width, Ht}) -> Width * Ht;
area({square, X})            -> X * X;
area({circle, R})            -> 3.14159 * R * R;
```

This is by far not all support for pattern matching in Erlang. Erlang provides wildcards, pattern matching with sequences, case statements similar to Scala, pattern guards, etc. Instead of presenting all these features and their language syntax, the rest of this section concentrates on the probably most prominent application of pattern matching in Erlang: The *actor model*. The `receive` statement in Erlang's actor model uses pattern matching for deciding which messages in the mailbox of a process should be handled next. The syntax looks as follows:

```
receive
  pattern1 [when guard1] ->
      actions1;
  pattern2 [when guard2] ->
      actions2;
  ...
  patternN [when guard3] ->
      actionsN
end.
```

If a pattern matches a message and the optional guard succeeds, the message will be taken out of the mailbox and the corresponding action will be evaluated. This procedure will be repeated for all the messages that are contained in the mailbox. In case no message matches the defined patterns, then the process waits for a suitable message (i.e., is suspended). As an example, Listing 12 shows an actor implementation for converting the length units mile into kilometer and vice versa.

```
kmToMilesConverter() ->
 receive
   {toMiles, km} ->
     io:format("~p km is ~p miles",[km, km/1.61]),
     kmToMilesConverter();
   {toKM, miles} ->
     io:format("~p miles is ~p kilometer",
     [miles, 1.61*miles]),
     kmToMilesConverter();
 end.
```

Listing 12: *Length unit converter in Erlang*

Erlang chooses which messages it should process based on the tuple format which is either `toMiles`, indicating the convertion to mile unit and its only parameter `km`, or `toKM` used to convert the argument `miles` into the unit kilometer. Notice the recursion here which is used to process the next incoming messages. As a side note it should be mentioned that Scala has a very similar implementation with its own actor concept, which is also greatly based on pattern matching.

3.3 F#

F# [28] is a very new multi-paradigm programming language for the .NET platform. It is similar to Scala in that it's statically typed, combines functional programming with object-oriented techniques and tries to fit on a very stable and proven platform. This section outlines the pattern matching F# offers and its similarities to Scala's implementation.

3.3.1 Pattern Matching

F# has rich support for pattern matching. It provides pattern matching with all its standard types like tuples, record types and discriminated unions. There is also a pattern matching construct similar to Scala's extractors which is discussed in detail in the next section.

As an example of F#'s basic pattern matching syntax the following function `fib` shows how the Fibonacci sequence is calculated:

```
let rec fib n =
  match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> fib(n - 1) + fib(n - 2)
```

The syntax for the pattern matching is similar to the one of Haskell. Of course it is also possible to pattern match over F# union types. Listing 13 shows a union type for a binary tree structure with a customizable type `t` and a function for evaluating the tree with in-order traversal [29]. Compare this example with the solution for Scala given in section 2.3.

```
type BinaryTree<'t> =
  | Leaf
  | Node of 't * BinaryTree<'t> * BinaryTree<'t>

let rec inorder t =
  match t with
  | Leaf          -> []
  | Node (v,l,r) -> inorder(l) @ [v] @ inorder(r)
```

Listing 13: *Binary tree and in-order traversal in F#*

The discriminated union type `BinaryTree` has the following two *tags* as they are called by Syme et al. in [30]: `Leaf` and `Node`. Notice the similarity between these and Scala's case classes.

3.3.2 Active Patterns

There is a concept in F# related to Scala's extractors with the objective to make pattern matching more extensible. F#'s solution is called *active patterns*. According to its inventors Don Syme et al., active patterns try to address the problems pattern matching on concrete types causes [30]. With active patterns one can provide different views to the underlying data types. They allow to hide the internal representation with the goal to only publish these views as Petricek states in [31].

Active patterns have the form `(|id|...|id|{_|}?)` [30] whereas the brackets are commonly known as "bananas". The code in listing 14 [30] shows the use of active patterns with complex numbers. A complex number can be viewed either as a number in the cartesian or in the polar form. With active patterns one wants to hide the internal representation of these types.

```
let mkCart (x,y) = Complex.mkRect(x,y)
let mkPolar (r,th) = Complex.mkPolar(r,th)

let (|Cart|) (c:complex) = (c.RealPart,
                            c.ImaginaryPart)
let (|Polar|) (c:complex) = (c.Magnitude, c.Phase)

let add c1 c2 =
    match c1, c2 with
    | Cart (ar, ai), Cart(br, bi) ->
      mkCart(ar*br - ai*bi, ai*br + bi*ar)
let mul c1 c2 =
    match c1, c2 with
    | Polar(r1, th1),Polar (r2, th2) ->
      mkPolar(r1*r2, th1+th2)
```

Listing 14: *Complex numbers as F# active patterns*

The definition of the active pattern `|Cart|` binds the name `Cart` to a value of the type `complex -> float * float` which means that everywhere this pattern occurs it matches complex numbers and returns two float values [30]. As with Scala's extractors one could change the underlying data representation (in this example the format of the complex number between cartesian and polar) without affecting the client's code base.

3.4 XSLT

XSL Transformations (XSLT) [32] uses pattern matching in the application of template rules to transform XML documents into other XML documents. Pattern matching is very important here because different data needs to be processed (e.g., formatted) differently and the types of data are distinguished by pattern matching. A template rule consists of a template pattern and a template body. The following example shows the basic syntax of a template rule in XSLT:

```
<xsl:template match="pattern">
  <!-- template body -->
</xsl:template>
```

Template patterns in XSLT are denoted with the XML Path Language (XPath) [33] syntax. These template patterns are matched against nodes (e.g., elements, attributes or text) in the source tree and if the match succeeds, the template body is instantiated and generates a part of the result tree as Kay

describes in [32]. As an analogy one can compare XSLT template rules to pure functions in the tradition of functional programming languages because they define functions which map fragments of the input tree to fragments of the output tree free of side effects [34].
The most important matching facility XPath provides is the concept of *location path*. Location paths look a lot like the well known Unix file system paths. As an example imagine an XML document which is represented by a source tree as shown in figure 1. To address node *E* one could use the XPath expression `/F/B/D/E`. XPath also allows to select all descendants of a node including itself with the double slash notion. `//D` would therefore result in the node set `D,C,E`.
XPath offers a concept similar to pattern guards in Scala called *predicates*. Predicates are especially useful when a XPath expression matches more than one node. In that case one can further narrow the resulting node set with a predicate. Assume one wants to process all economic books in the XML book example found in appendix B with a XSLT template that prints the title of the matched books in form of a HTML list:

```
<xsl:template match="//book[@genre='economics']">
  <li>
    <xsl:value-of select="title" />
  </li>
</xsl:template>
```

The predicate is denoted inside of square brackets. The @ symbol is used here to refer to the attributes of the matched book nodes. This predicate only returns nodes that have an attribute `genre` with the value *economics*. It is important to note that the matched set of nodes defines the context for further pattern matches inside the template rule. This is the case here with the output of the book titles. The pattern `title` refers to this context and returns only the title nodes of the node set defined by the template pattern.
XSLT has a set of integrated default template rules which can be overridden by self-defined templates. The most important one is shown next which guarantees that all nodes in the source tree are processed [35]:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

The asterisk `*` is a XPath wildcard and matches all nodes. XPath also allows to match several types of nodes with the help of its union operator `|`. This is highlighted here with the combination of the slash `/` which denotes the root node of the source tree. The template body doesn't process the node set directly. Instead it delegates the processing with the help of the XSLT instruction `<xsl:apply-templates>` to any other template rule that has a corresponding pattern. The partitioning of the transformation process into several template rules with well-defined responsibilities is a very common approach in XSLT.
The shown pattern matching facilities of XSLT are only a tiny subset of the full capabilities of the language, considerably with the new XSLT 2.0 and XPath 2.0 standards. The interested reader will find more information in Michael Kay's book "XSLT 2.0 and XPath 2.0 Programmer's Reference" [36].

3.5 Prolog

Prolog [37] makes use of pattern matching in the process of finding solutions to given queries. Consider the following list of facts in Prolog about a part of the relationships between the members of the British royal family:

```
son(charles, elizabeth).
son(andrew, elizabeth).
son(william, charles).
son(henry, charles).
daughter(anne, elizabeth).
daughter(beatrice, andrew).
```

To find all sons of Charles one could query Prolog's fact database like follows:

```
?- son(X, charles).
```

The Prolog interpreter now scans the facts top down and tries to bind a value to the variable `X` so that the query pattern `(X, charles)` is matched. This process is called *unification*. Prolog then answers with `X = william ->`, which denotes that `william` is the first match. The arrow symbolizes the backtracking part of Prolog [38]. If one hits the symbol `;` Prolog unbinds the variable `william` from `X` and continues with the search as described and comes up with another matching solution (`henry`). After pressing `;` again there are no more values to match anymore and the interpreter prints `no`. The continues unbinding and searching for additional solutions is Prolog's *backtracking* process.
Consider the more complex example with the rules given in listing 15 (similar to Hoffmann's implementation in [39]). The first two lines define that if `X` is a son or a daughter of `Y` then `X` is a child of `Y`. The following two rules define under what circumstances `X` is a descendant of `Y`: Either `X` is a child of `Y` or `X` is a child of `Z` and `Z` is a descendant of `Y` through recursion.

```
child(X,Y) :- son(X,Y).
child(X,Y) :- daughter(X,Y).

descendant(X,Y) :- child(X,Y).
descendant(X,Y) :- child(X,Z),
                   descendant(Z,Y).
```

Listing 15: *Rules for relationships between family members in Prolog*

If one states a query, Prolog tries to find a match for the pattern as described before through unification and backtracking. As an example the following query results in `true`:

```
?- descendant(anne, elizabeth).
```

Prolog also supports the concept of *wildcards*. If one uses a wildcard as in the following example, Prolog will not bind any variables. This is useful, e.g., if one is only interested if Elizabeth has any sons at all and not which ones in particular:

```
?- son(_, elizabeth).
true; true; false.
```

In this example the Prolog interpreter returns two times `true` because Elizabeth has two sons. After the last backtracking Prolog returns `false`, indicating that there are no more matches.
Prolog and logic programming languages in general offer - compared to the languages discussed so far - an advanced version of pattern matching which is called *two-way matching* [25]. Consider the following example:

```
?- f(1,X,3) = f(Y,2,Z).
```

The Prolog interpreter returns `X=2,Y=1,Z=3`. Prolog tries to bind values to variables with its unification process from left to right and from right to left, therefore is the name two-way matching.

4 Discussion

As seen in the last section Scala's pattern matching functionality is greatly influenced by other languages like, e.g., Haskell. Scala's case classes are a convenient way of expressing different types of objects and to use them in pattern matching code. Through the automatically provided methods (e.g., `hashCode`) the developer can get rid of a lot of boilerplate code if one compares this to the amount of code that would be necessary for the same functionality in Java, e.g., with Java Beans.
Through Scala's compiler support with sealed base classes one can gain a lot of comfort and safety. Traditional object-oriented languages like Java lack this kind of support and need to provide a default case in their matching code. Concerning the performance of pattern matching with case classes and extractors compared to traditional object-oriented techniques like type-tests / type-casts, object-oriented decomposition or the application of the visitor pattern, investigations by Emir [4] have shown that Scala's pattern matching mechanism is not inferior compared to these. Nevertheless note that - according to Emir - unlike case classes there is a performance penalty to pay with extractors when using them heavily in pattern matching code.
The argument that pattern matching in Scala with case classes breaks encapsulation that was often stated by the OOP community and also accepted as a fact in Emir's thesis [4] as a fundamental flaw can't last anymore with the introduction of extractors. As was described in section 2.8.3 concerning representation independence with the example of calculations with Celsius and Fahrenheit degrees, one can just provide another view for new data types without affecting clients pattern matching code. For a detailed discussion about encapsulation and pattern matching in general one might follow Prof. Odersky's post "In Defense of Pattern Matching" and the corresponding discussion thread [40].
To summarize, when using Scala's pattern matching functionality programmers can gain safety through compiler support, less and better readable code without losing speed. Even though its usefulness there is one shortcoming in Scala's implementation of pattern matching: Its syntax still looks a little bit verbose compared to, e.g., Haskell or F#.

5 Conclusions

This paper gave an overview over the pattern matching facilities in Scala. In the first half of the paper Scala's two basic concepts for pattern matching, case classes and extractors, have been discussed. Various examples and applications of pattern matching in Scala's internals - as, e.g., Scala's event and exception handling - showed its underlying principles. With Scala's built-in support for parsers through parser combinators the paper discussed the usefulness of case classes and pattern matching in the implementation of DSL's.
In the second half of the paper pattern matching in various prominent programming languages have been compared to Scala's implementation. Pattern matching is a very fundamental concept in both functional and logic programming languages and it was shown that Scala succeeded a lot of its concepts from its predecessors like, e.g., Erlang and Haskell.
The last part of the paper discussed Scala's implementation of pattern matching. Case classes offer a lot of automatically provided functionality for free and therefore one can get rid of a lot of boilerplate code easily. But as there are fundamental problems in terms of encapsulation with case classes it was stated that pattern matching in Scala gains its real power through the use of extractors.

A Pattern syntax in Scala

```
Pattern    ::= Pattern1 { '|' Pattern1 }
Pattern1   ::= varid ':' TypePat
             | '_' ':' TypePat
             | Pattern2
Pattern2   ::= varid ['@' Pattern3]
             | Pattern3
Pattern3   ::= SimplePat
             | SimplePat
               { id [nl] SimplePat }
SimplePat  ::= '_'
             | varid
             | Literal
             | StableId
             | StableId '(' [Patterns [',']] ')'
             | StableId '(' [Patterns ','']
                        [varid '@'] '_' '*' ')'
             | '(' [Patterns [',']] ')'
             | XmlPattern
Patterns   ::= Pattern [',' Patterns]
             | '_' *
```

B XML Book List example

```
<?xml version="1.0"?>
<catalog>
  <book id="ka03" genre="philosophy">
    <author>Immanuel Kant</author>
    <title>Critique of Pure Reason</title>
    <publisher>Dover Publications</publisher>
    <pub_year>2003</pub_year>
  </book>
  <book id="sm08" genre="economics">
    <author>Adam Smith</author>
    <title>Wealth of Nations</title>
    <publisher>Dolphin Books</publisher>
    <pub_year>2008</pub_year>
  </book>
  <book id="asu85" genre="computer science">
    <author>Aho, Sethi and Ullman</author>
    <title>
      Compilers: Principles, Techniques,
      and Tools
    </title>
    <publisher>Addison-Wesley</publisher>
    <pub_year>1985</pub_year>
  </book>
  <book id="me00" genre="computer science">
    <author>
      <first_name>Bertrand</first_name>
      <last_name>Meyer</last_name>
    </author>
    <title>
      Object-oriented
      Software Construction
    </title>
    <publisher>Prentice Hall</publisher>
    <pub_year>2000</pub_year>
  </book>
</catalog>
```

References

[1] Adam Richard and Ondrej Lhotak. OOMatch: Pattern Matching as Dispatch in Java. Technical report, University of Waterloo, 2007.

[2] Luc Maranget. Warnings for pattern matching. In *Journal of Functional Programming*, 2007.

[3] Graham Hutton. *Programing in Haskell*. Cambridge University Press, 2007.

[4] Burak Emir. *Object-Oriented Pattern Matching*. PhD thesis, ETH Lausanne, 2007.

[5] Andrew C. Myers and Jed Liu. JMatch - Pattern Matching and Interruptible Iterators for Java. World Wide Web, http://www.cs.cornell.edu/Projects/jmatch/, 2009.

[6] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *Proc. of the 5th Int. Symposium of Pratical Aspects of Declarative Languages (PADL)*, pages 110–127, 2003.

[7] Jean-Christophe Bach et al. Tom. World Wide Web, http://tom.loria.fr/, 2009.

[8] Pierre-Etienne Moreau, Christoph Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *Proc. of Compiler Construction (CC)*, volume 2622 of LNCS, pages 61–76, 2003.

[9] 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.

[10] Allen Leung. Prop: A C++ based Pattern Matching Language. World Wide Web, http://www.cs.nyu.edu/leunga/prop.html, 2009.

[11] Allen Leung. Prop: A C++-based Pattern Matching Language. World Wide Web, http://www.cs.nyu.edu/leunga/papers/research/prop.ps, 1996.

[12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Inc., 2008.

[13] Martin Odersky. The Scala Language Specification, Version 2.7. World Wide Web, http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2009.

[14] Martin Odersky. The Scala Experience - Programming With Functional Objects. World Wide Web, http://lamp.epfl.ch/~odersky/talks/pppj07.pdf, 2007. OOPSLA 2007 Tutorial.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., 1995.

[16] Lukas Rytz. Named and Default Arguments in Scala 2.8. World Wide Web, http://www.scala-lang.org/sites/default/files/sids/rytz/Thu,%202009-05-28,%2008:32/named-args.pdf, 2009.

[17] Philip Walder. Views: A way of pattern matching to cohabit with data abstraction. Technical report, Programming Research Group, Oxford University, UK and Programming Methodology Group, Chalmers University, Sweden, 1986.

[18] Swiss Interbank Clearing AG. Basisinformationen IBAN. World Wide Web, http://www.six-interbank-clearing.com/de/dl_tkicch_basisinfoiban0302.pdf, 2002.

[19] Martin Fowler. Mf bliki: Domain specific languages. World Wide Web, http://www.martinfowler.com/bliki/DomainSpecificLanguage.html, 2009.

[20] Ted Neward. The busy Java developer's guide to Scala: Building a calculator, Part 1-3. World Wide Web, http://www.ibm.com/developerworks/java/library/j-scala08268.html, 2008.

[21] Martin Odersky et al. Manpage of scalac. World Wide Web, http://www.scala-lang.org/docu/files/tools/scalac.html, 2007.

[22] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, Inc., 1971.

[23] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2005.

[24] Simon Peyton Jones et al. Haskell 98 Language and Libraries - The Revised Report. World Wide Web, http://haskell.org/onlinereport, 2002.

[25] Paul Hudack, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell 98. World Wide Web, http://www.haskell.org/tutorial/haskell-98-tutorial.pdf.

[26] Jonas Barklund and Robert Virding. Erlang 4.7.3 Reference Manual. World Wide Web, http://www.erlang.org/download/erl_spec47.ps.gz, 1999.

[27] Joe Armstrong. The Pragmatic Bookshelf: What's all this fuss about Erlang? World Wide Web, http://www.pragmaticprogrammer.com/articles/erlang.html, 2007.

[28] Microsoft Corporation. The F# 1.9.7 Draft Language Specification. World Wide Web, http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf, 2009.

[29] Robert Pickering. Beyond Foundations of F# - Active Patterns. World Wide Web, http://www.infoq.com/articles/Beyond-Foundations-FSharp, 2007.

[30] Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *Proc. of ICFP 2007*, 2007.

[31] Tomas Petricek. F# language overview. World Wide Web, http://tomasp.net/articles/fsharp-i-introduction/article.pdf, 2007.

[32] Michael Kay. XSL Transformations (XSLT) Version 2.0. World Wide Web, http://www.w3.org/TR/xslt20/, 2007.

[33] Anders Berglund et al. XML Path Language (XPath) 2.0. World Wide Web, http://www.w3.org/TR/xpath20/, 2007.

[34] Michael Kay. What kind of language is XSLT? World Wide Web, http://www.ibm.com/developerworks/xml/library/x-xslt/, 2005.

[35] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, Inc., 2004.

[36] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Wrox, 2008.

[37] J.P.E. Hodgson. Prolog: The ISO Standard Documents. World Wide Web, http://pauillac.inria.fr/~deransar/prolog/docs.html, 1999.

[38] Dennis Merritt. Prolog Under the Hood: An Honest Look. World Wide Web, http://www.amzi.com/articles/prolog_under_the_hood.htm, 1992.

[39] Dirk W. Hoffmann. *Theoretische Informatik*. Hanser, 2009.

[40] Martin Odersky. In Defense of Pattern Matching. World Wide Web, http://www.artima.com/weblogs/viewpost.jsp?thread=166742, 2006.