Matchete: Paths through the Pattern Matching Jungle

Martin Hirzel[1], Nathaniel Nystrom[1], Bard Bloom[1], and Jan Vitek[1,2]

[1] IBM Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
{hirzel,nystrom,bardb,jvitek}@us.ibm.com
[2] Purdue University, Dpt. of Computer Science, West Lafayette, IN 47907, USA

Abstract. Pattern matching is a programming language feature for selecting a handler based on the structure of data while binding names to sub-structures. By combining selection and binding, pattern matching facilitates many common tasks such as date normalization, red-black tree manipulation, conversion of XML documents, or decoding TCP/IP packets. Matchete is a language extension to Java that unifies different approaches to pattern matching: regular expressions, structured term patterns, XPath, and bit-level patterns. Matchete naturally allows nesting of these different patterns to form composite patterns. We present the Matchete syntax and describe a prototype implementation.

Keywords: Pattern matching, regular expressions, XPath, binary data formats, Java.

1 Introduction

Recognizing patterns in data is a recurrent problem in computer science. Many programming languages and systems provide syntax for pattern matching. Functional programming languages emphasize matching over data types, and support defining functions as sequences of cases over the structure of their parameters. String-oriented languages such as AWK or Perl come with builtin support for pattern matching with a powerful regular expression language. XML processing systems often support extracting sub-structures from a document. Finally, some languages support matching of bit-level data to extract patterns in network packets or binary data streams. While pattern matching constructs differ in terms of syntax, data types, type safety, and expressive power, they share the common characteristic of being able to conditionally deconstruct input data and bind variables to portions of their input. This paper is a step towards a unified pattern matching construct for the Java programming language. Our experimental compiler, called Matchete, supports the different flavors of pattern matching mentioned above as well as user-defined patterns.

Below is a simple example that illustrates the expressive power of Matchete. The method `findAge()` expects a list of strings containing a name and an age, encoded as a sequence of letters followed by a seqence of digits. It traverses the list until it finds a value matching its `name` argument. If found, it converts the associated age to an integer and returns it. This function showcases a number of features of Matchete. The `match` statement extracts the value from a `List` object. A nested pattern specifies a regular expression and at the same time performs string comparison against the value of `name`. Its `age` field is implicitly converted from a `String` to a primitive `int`.

P. Hudak and D.S. Warren (Eds.): PADL 2008, LNCS 4902, pp. 150–166, 2008.

```
int findAge(String name, List l) {
  match(l) {
    cons~(/([a-zA-Z]+) ([0-9]+)/(name, int age), _): return age;
    cons~(_, List tail): return findAge(name, tail);
  }
  return -1;
}
```

Matchete's contribution is a seamless and expressive integration of the major flavors of pattern matching with a straightforward syntax and semantics. This should be contrasted with many recent efforts that focus on a particular pattern matching style, for example, functional-style patterns in an object-oriented language [5,19,20]. In Matchete, functional-style term patterns, Perl-style regular expressions, XPath expressions, and Erlang-style bit-level patterns can contain one another, and use the same small set of primitive patterns at the leaves. Matchete is a minimal extension to Java, adding only one new statement, one new declaration, and one new kind of expression to the base language. We have implemented a fully functional prototype and have validated the applicability of Matchete by a number of small case studies. The Matchete prototype compiler performs no optimizations, we leave this to future work.

2 Related Work

Structured term pattern matching is a central feature of functional programming languages. In languages such as ML [22] or Haskell [17], instances of algebraic data types can be constructed and deconstructed using the same constructors. The simplicity and elegance of the approach is tied to having a relatively simple data model in which the definition of a data type suffices to automatically define constructors that can be inverted to deconstructors. Object oriented languages introduce abstract data types, and even where constructors could be automatically inverted, this would violate encapsulation and interact poorly with inheritance. A number of recent works have investigated extensions to object-oriented languages that allow pattern matching over abstract data types. Views are implicit coercions between data types that are applied during pattern matching [27]. Active patterns [26] for F# generalize views to functions that deconstruct values into an option type—either `Some` values if the input value can be deconstructed or `None`. Active patterns can be used just like regular structural pattern matching on data types. Scala's extractors [5] are a form of active patterns for objects. PLT Scheme's match form allows adding new pattern matching macros, which can be used to support other kinds of pattern matching by supplying an expansion to the primitive matching forms [29]. Tom is a preprocessor that adds structured term pattern matching to Java, C, and Eiffel [23]. OOMatch [25] and JMatch [20] are Java extensions. OOMatch allows pattern declaration in method parameters and resembles Matchete in its treatment of extractors. JMatch provides invertible methods and constructors, which serve to deconstruct values during pattern matching, and also support iteration and logic programming.

String pattern matching is a central feature of text-processing languages such as SNOBOL [12] and Perl [28]. While the Java standard library provides an API for Perl-style regular expressions, which are familiar to many programmers, this API can be

awkward to use and leads to code that is considerably more verbose than an equivalent Perl program. Matchete addresses this shortcoming by integrating Perl regular expressions directly in the language.

Bit-level data manipulation has traditionally been the domain of low-level languages such as C. Some recent work takes a type-based approach for parsing bit-level data [1,4,7]. The Erlang programming language, on the other hand, allows specifying bit-level patterns directly. Erlang's patterns are widely used for network protocols, and are optimized [14]. Matchete follows the Erlang approach.

XML pattern matching comes in two flavors: XPath expressions and semi-structured terms. XPath expressions are paths through the tree representation of an XML document that specify sets of nodes [3]. XPath is the primary matching mechanism of XSLT, and Matchete supports XPath directly. Several recent languages treat XML as semi-structured terms [2,8,15,16,18,19,21]. These languages support patterns similar to structured term patterns in functional languages, in some cases augmented by Kleene closure over sibling tree nodes. Matchete also supports structured term patterns.

What sets Matchete apart from this previous work is that it integrates XPath and structured term matching with each other and with Perl-style regular expressions and bit-level patterns.

3 The Matchete Language

Matchete extends Java with a `match` statement with the following syntax:

MatchStatement ::= `match (` *Expression* `) {` *MatchClause** `}`
MatchClause ::= *MatchPattern* `:` *Statement*

A `match` requires an *Expression* (the subject of the match) and zero or more *MatchClause*s. Each *MatchClause* has the form *MatchPattern* : *Statement*, where the *MatchPattern* guards the execution of the *Statement* (or handler), and may make some bindings available for the handler. The syntax deliberately resembles that of the Java `switch` statement, with three important differences: there is no need to write `case` before each clause, each handler consists of a single statement (which may be a block), and the `break` keyword is not used to prevent fall-through. This last difference is motivated by software engineering concerns (it is a common mistake to forget a `break`), and by the need to provide a well-defined scope for variables bound by patterns.

A common beginner's exercise in functional languages is to write a recursive function `mult` that multiplies the elements of a list. List multiplication has a simple recursive definition: multiply the first element with the result of multiplying the rest of the list. For example, `mult([2, 3]) = 2*mult([3]) = 2*3*mult([])`. The last factor, `mult([])`, requires a base case: for an empty list the function returns 1, the multiplicative identity. Of course, if any number in the list is zero, the entire product will be zero and the rest of the list need not be evaluated.

Fig. 1 shows the Matchete definition of `mult`. The match statement matches the parameter `ls` against two clauses. The first clause handles the case when the value at the head of the list is zero. The second clause extracts the head of the list, `h`, and the tail of the list, `t`, and multiplies `h` by the result of recursively calling `mult` on `t`. If the list

```
1  int mult(IntList ls) {
2    match (ls) {
3      cons~(0, _): return 0;
4      cons~(int h, IntList t): return h * mult(t);
5    }
6    return 1;
7  }
```

Fig. 1. List multiply in Matchete

is empty, neither clause matches, and `mult` returns 1. The method `cons~()` is a user-defined deconstructor of the `IntList` class that extracts the head and the tail of a list.

This example illustrates four kinds of patterns: wildcard (_ matches anything), values (0 matches the integer zero), binders (`int h` matches any integer and binds it to `h`), and deconstructor patterns (`cons~()` matches composite data, extracts its parts, and delegates said parts to nested patterns).

3.1 Evaluation Order

Matchete defines a deterministic order of evaluation for patterns. A match statement evaluates match clauses sequentially in textual order until a clause succeeds. Each clause evaluates patterns sequentially in textual order until either a pattern fails, or control reaches the handler statement.

Each pattern, whether simple or composite, operates on a *subject*. The expression on which the match statement operates becomes the subject for the outermost pattern of each match clause. Composite patterns provide subjects for their children (nested patterns) to match on. Consider the following clause where the outer `cons~` supplies its two nested patterns with subjects:

```
cons~(1, cons~(int x, IntList y)): print("1::"+x+"::"+y);
```

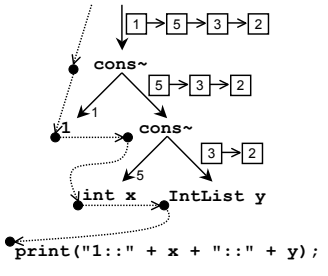

Fig. 2. Evaluation order

Each pattern also has a unique *successor*. The successor of a pattern is the next pattern to run if the match has been successful through the given pattern. In other words, each pattern determines whether its successor runs, or whether to branch to the next clause, if any. The successor of the last pattern in a match clause is the handler.

Fig. 2 illustrates match statement evaluation order: composite patterns have solid edges to child patterns (nesting edges), and each pattern has a dotted edge to its successor (branching edges). Subjects flow along nesting edges, bindings flow along successor edges. Nesting edges are labeled with subjects flowing from parent to child patterns. For example, if the match statement operates on the list `[1,5,3,2]`, that list becomes the subject of the outermost pattern. Each parent pattern extracts subjects for its children to match on. In this case, the number `1` flows to the left child, and the sublist `[5,3,2]` flows to the right child. Successor edges

chain all patterns in the nesting tree in a preorder depth-first traversal. This means that pattern matches are attempted in textual order, left-to-right. If a pattern succeeds, it branches to its successor, otherwise, it branches out of the current match clause. The successor of the last pattern is the handler, in this case, the `print` statement.

3.2 Type Conversions

Because there can be a mismatch between the type of the data under consideration and the most convenient type for manipulating that data in the handler, Matchete provides support for automatic type conversions. For example, Java programs may store boxed `Integer` values, yet in order to perform arithmetic, these must be unboxed. Matchete will do the unboxing for the programmer as part of pattern matching. Likewise, XML DOM trees contain text nodes, which the handler may want to manipulate as strings. Pattern matching also makes a branching decision, and it is often convenient to consider the type in that decision. For example, catch clauses in Java try/catch statements perform a limited form of pattern matching, converting the subject (an exception object) to the type of the catch clause (a subclass of `Throwable`) if the appropriate subclass relationship holds.

Table 1. Type conversions during pattern matching. `Object`, `String`, and `Integer` are defined in package `java.lang`, while `Node` and `NodeList` are defined in package `org.w3c.dom`.

Subject	Target type	Constraints
`obj`	→ `RefType`	`obj instanceof RefType`
`obj`	→ `Node`	`obj instanceof NodeList` and `obj.getLength()==1`
`obj`	→ `String`	`obj instanceof Node` and `obj.getNodeValue()!=null`
`obj`	→ `PrimitiveType`	`obj` is a boxed object assignment convertible to PrimitiveType
`obj`	→ `PrimitiveType`	`obj instanceof String` and T`.parse`T`(obj)` succeeds
`prim`	→ `String`	(always succeeds, using the `toString` method of the box type)
`prim`	→ `PrimitiveType`	`prim` assignment convertible to `PrimitiveType`

Matchete augments Java's type conversions and promotions [9, Chapter 5] with so-called *matching conversions*, defined as a relation between values and types. Table 1 gives these conversion rules. For example, if the subject of a match is a reference with static type `Object`, say the string literal `"42"`, it can be converted to a reference target of type `String` provided that the dynamic `instanceof` check succeeds. In some cases, the conversion may involve multiple steps to get from the subject to the target. For example, `NodeList` → `Node` → `String` → `int` starts from the result of an XPath query, and converts it to an `int` if all the constraints along the way are satisfied. In general, Matchete attempts conversions in the order in which Table 1 enumerates them, and the conversion succeeds if it reaches the target type.

3.3 Primitive Patterns

Matchete has three kinds of primitive patterns which can be used at the leaves of a match clause.

Wildcard Patterns ::= `_`
The wildcard pattern, written `_`, is a catch-all that always matches. Formally, every occurence of the wildcard pattern is distinct and can match against any Java value.

Value Patterns ::= *Expression*
It is often convenient to check whether a subject has a particular value. In general, value patterns match against arbitrary Java expressions. A value pattern match first checks whether the type of the subject can be converted to the type of the expression. If so, it checks for equality using `==` (for primitive types or null) or `equals()` (for reference types). If both the conversion and the comparison succeed, the value pattern branches to its successor, otherwise it fails.

Binder Patterns ::= *Modifiers Type Identifier Dimensions*$^?$
Composite patterns extract parts of the subject and bind them to names, so that the handler statement can refer to them directly. This binding is performed by binder patterns. For example, the `mult` function uses the binder pattern `int x` to bind the head of the list to a new local variable `x` for the handler. In general, a binder pattern match succeeds if the type of the subject can be converted to the type of the expression. Binder patterns look like variable declarations, with modifiers (e.g., `final`), a type, an identifier, and an optional dimensions for array types. The binding is visible in all successors in the same match clause.

3.4 Composite Patterns

A composite pattern is one that has nested sub-patterns, which may themselves be composite or primitive. Each composite pattern, regardless of kind, first decides whether or not to invoke the nested patterns, and if yes, supplies them with subjects. If the composite pattern and everything nested inside of it succeed, it invokes its successor. For example, the root node of Fig. 2 is a composite pattern. It first checks that its subject is a non-empty list. If so, it extracts parts of the list, and supplies them as subjects to its children. If all succeed, the handler runs.

Deconstructor Patterns ::= *Identifier* `~ (` *PatternList* `)`
Deconstructor patterns allow Matchete to match structured terms of user-defined data types and thus program in a style reminiscent of functional programming. One notable difference is that deconstructor patterns invoke user-defined methods that decouple data extraction from the implementation of the data type, preserving encapsulation. A pattern list is simply a comma-separated list of match patterns:

$$\mathit{PatternList} ::= \mathit{MatchPattern}\ \big(\texttt{,}\ \mathit{MatchPattern}\big)^* \mid \mathit{Empty}$$

Semantically, the deconstructor pattern first checks whether the subject has a deconstructor method with the given identifier, for example, `cons` in Fig. 1. If yes, it calls `subject.method ()`. The method either returns the subjects for nested patterns, or reports a failure. If there was no failure and the number of subjects matches the length of the *PatternList*, the deconstructor pattern branches to the first nested pattern. Matching proceeds as usual following the rules from Section 3.1.

```
1 class IntList {
2   private int head; private InList tail;
3   public IntList(int h, IntList t){ head = h; tail = t; }
4   public cons~(int h, IntList t){ h = head; t = tail; }
5 }
```

Fig. 3. List declaration with deconstructor

Deconstructor methods have syntax and semantics that differ from normal Java methods. Syntactically, deconstructors are denoted by the presence of a tilde between their name and their argument list. They have no declared return type.

Declaration ::+= ... | *Deconstructor*
Deconstructor ::= *Modifiers Identifier* ~ (*ParameterList*) *ThrowsClause*$^{?}$ *Block*

Semantically, the arguments of a deconstructor are *out* parameters which must be assigned to in the body. The deconstructor can use a `fail;` statement to exit early and report failure. Fig. 3 is an example where class `IntList` has two private fields, a constructor, and a deconstructor (Line 4). In this case, the deconstructor is the inverse of the constructor, it assigns the fields into output parameters for use as subjects in nested matches. The current version of Matchete has no notion of exhaustive matches or unreachable clauses. Considering that deconstructors are user-defined this seems difficult.

Array Patterns ::= *ArrayType* { *PatternList* }
Array patterns are a special case of deconstructor patterns for Java arrays. For example, the pattern `int[]{1, x, int y}` matches an array of three elements if the first element is 1 and the second element has the same value as variable `x`, and binds the third element to a fresh variable `y`. The syntax of array patterns resembles that of array constructors. In general, an array pattern first checks whether the subject is an array of the appropriate length, then invokes nested patterns on extracted elements. Matching proceeds as usual following the rules from Section 3.1.

Regular Expression Patterns ::= / *RegExpLiteral* / (*PatternList*)
Perl excels at extracting data from plain text. This can be attributed to the tight integration of regular expression pattern matching in the syntax. Regular expressions are a fundamental concept from language theory that has phenomenal practical success, because they concisely describe string matches that can be implemented efficiently. For instance, consider the regular expression pattern `/([0-9]+)\.([0-9]+)/ (_, int frac)`. The slashes delimit a regular expression that matches a sequence of digits followed by a decimal point followed by more digits. Parentheses, `(...)`, inside the regular expression capture groups of characters to extract. The parentheses on the right contain a list of nested patterns, which operate on the groups captured by the regular expression on the left. On success, this pattern binds `frac` to the digits after the decimal1 point.

Regular expression patterns first convert the subject to a string, then match it as specified by the `java.util.regex` package. If this succeeds and produces the correct number of results, the pattern invokes its nested patterns, providing the results as subjects. Matching proceeds as usual following the rules from Section 3.1.

XPath Patterns ::= < *XPathLiteral* > (*PatternList*)
XML is a widely-used data interchange format, and XPath is a pattern matching mechanism on the tree representation of an XML document. XPath is widely used because it facilitates common data manipulation tasks through a simple tree query language. An XPath query specifies a path in the tree of XML nodes in a fashion similar to how file name paths in most operating systems specify paths in the tree of directories. The subject of the match is a node, and the result is a set of nodes. Matchete supports XPath patterns. For example, `<bibliography/article> (NodeList nodes)` extracts the set of all `article` grandchildren that are children of `bibliography` children of the subject.

A Matchete XPath pattern converts the subject to a `Node` or `InputSource`, then queries it as specified by the `javax.xml.xpath` package. If this throws any exception, it catches that exception and the match fails, otherwise, it supplies the resulting `NodeList` as the subject to nested patterns. Matching proceeds as usual following the rules from Section 3.1.

Bit-Level Patterns ::= `[[` (`0` | `1` | `(` *MatchPattern* `:` *Expression* `)`)* `]]`

When communicating with low-level network and hardware interfaces, programs need to manipulate data at the level of raw bits. Writing code that does that by hand with shifts and masks is time-consuming and error-prone. However, this problem resembles other typical pattern tasks, in that it requires branching (depending on tag bits) and binding (extracting sequences of payload bits and storing them in variables). Matchete supports matching at the bit-level using patterns such as

```
[[ (0xdeadbeef : 32) 10 (byte x: 6) (int y: x) (int z: 24 - x) ]]
```

This example extracts the first 32 bits, converts them to an integer, and matches them against the the nested integer value pattern `0xdeadbeef`. On success, it matches literal bits `1` and `0`, then a group of 6 bits, which it binds using the nested binder pattern `byte x`. Next, it extracts a group of `x` bits into `y`, and a group of $24 - \texttt{x}$ bits into `z`.

Bit-level patterns consist of two kinds of sub-patterns: literal `0` or `1` for matching individual bits, and groups in parentheses for matching subsequences of bits. Each group has the syntax `(` *MatchPattern* `:` *Expression* `)`, where the expression specifies a bit width, the number of bits to use as the subject of the nested pattern. The width expression can be any Java expression producing an `int`, including literals, variables, or arithmetic expressions. The subject for the nested pattern is the group of bits converted to the smallest primitive type that will hold it.

Besides patterns for bit-level deconstruction, Matchete also has expressions for bit-level construction, whose syntax is similar:

PrimaryExpression ::+= ... | *BitLevelExpression*
BitLevelExpression ::= `[[` (`0` | `1` | `(` *Expression* `:` *Expression* `)`)* `]]`

Parameterized Patterns ::= *Identifier* `(` *Expression* `)` `~` `(` *PatternList* `)`
Parameterized patterns are Matchete's extension mechanism: they allow users to implement new kinds of patterns for use in match statements. For example, the following code uses a parameterized pattern where the parameter is a regular expression string, instead of using the built-in RegExp syntax:

```
1 matchete.Extractor re = myLibrary.RegExp();
2 match(subject) {
3   re("([a-zA-Z]+) ([0-9]+)")~(name, int age): handler(age);
4 }
```

Line 1 creates an extractor and stores it in variable `re`. Matchete implements Line 3 by first making the call

```
re.extract(subject, "([a-zA-Z]+) ([0-9]+)")
```

That call returns an *extraction* (a tuple of subjects), and Matchete matches the extraction against the nested patterns (in this case, the value pattern `name` and the binder pattern `int age`). If all succeeds, Matchete executes the handler.

To support an additional pattern matching mechanism in Matchete, the user needs to implement two matchete library interfaces:

```
interface Extractor{ Extraction extract(Object subject, Object pattern); }
interface Extraction{ int size(); Object get(int i); }
```

In the earlier example, `myLibrary.RegExp` was a class that implements the Extractor interface, and `"([a-zA-Z]+) ([0-9]+)"` was passed in as the `pattern` parameter. In general, the pattern parameter can be anything, such as an SQL query, a fileglob, a boolean predicate, or a scanf format. A parameterized pattern in a match statement leads to an extractor call, which returns an extraction, and Matchete matches the extraction against nested patterns. Matching proceeds as usual following the rules from Section 3.1.

3.5 Deconstructors, Extractors, and Parameterized Patterns

Matchete's deconstructor patterns and parameterized patterns are similar in that both invoke a user-defined pattern matching method. For deconstructors, that method is an instance method of the subject. For parameterized patterns, that method is a method of a separate extractor object. Deconstructors are part of the design of a data type, whereas parameterized patterns serve to wrap a pattern matching library that operates on existing types such as strings.

Other languages supported user-defined pattern matching methods before Matchete. The Scala language supports extractors, which are objects with a user-defined `unapply` method [5]. For example, if the name `cons` refers to an extractor object, then the pattern `cons(...)` calls `cons.unapply(subject)` and uses the return values in nested patterns. The F# language supports active patterns, which are first-class functions [26]. They work similarly to Scala extractors, but furthermore, can take additional input parameters, similar to parameterized patterns in Matchete.

Matchete gives a slightly different design point than Scala and F#. It relies less on functional language features, and integrates with more other flavors of pattern matching.

3.6 Summary

Table 2 summarizes Matchete's syntax. The additions are limited to a new kind of statement (*MatchStatement*), one new kind of declaration (*Deconstructor*), and one new kind of expression (*BitLevelExpression*). The syntax and semantics of these are described

Table 2. Matchete syntax. Literals are set in monospace fonts, non-terminals in italics. The syntax for RegExpLiteral and XPathLiteral is checked by `java.util.regex` and `javax.xml.xpath`.

Nonterminal		Parsing Expression
Statement	::+=	... | *MatchStatement*
Declaration	::+=	... | *Deconstructor*
PrimaryExpression	::+=	... | *BitLevelExpression*
MatchStatement	::=	`match` `(` *Expression* `)` `{` *MatchClause** `}`
MatchClause	::=	*MatchPattern* `:` *Statement*
MatchPattern	::=	*WildcardPattern* | *BitLevelPattern* | *ArrayPattern*
	|	*BinderPattern* | *DeconstructorPattern* | *ParameterizedPattern*
	|	*RegExpPattern* | *XPathPattern* | *ValuePattern*
ArrayPattern	::=	*ArrayType* `{` *PatternList* `}`
BinderPattern	::=	*Modifiers Type Identifier Dimensions*$^{?}$
BitLevelExpression	::=	`[[` (`0` | `1` | `(` *Expression* `:` *Expression* `)`)* `]]`
BitLevelPattern	::=	`[[` (`0` | `1` | `(` *MatchPattern* `:` *Expression* `)`)* `]]`
DeconstructorPattern	::=	*Identifier* `~` `(` *PatternList* `)`
Deconstructor	::=	*Modifiers Identifier* `~` `(` *ParameterList* `)` *ThrowsClause*$^{?}$ *Block*
ParameterizedPattern	::=	*Identifier* `(` *Expression* `)` `~` `(` *PatternList* `)`
PatternList	::=	*MatchPattern* (`,` *MatchPattern*)* | *Empty*
RegExpPattern	::=	`/` *RegExpLiteral* `/` `(` *PatternList* `)`
ValuePattern	::=	*Expression*
WildcardPattern	::=	`_`
XPathPattern	::=	`<` *XPathLiteral* `>` `(` *PatternList* `)`

earlier in this section. Composite patterns are those where *MatchPattern* occurs in the right hand side of the grammar rule. By using the general *MatchPattern* non-terminal instead of any specific kind of pattern, all formalisms are pluggable into each other at the syntactic level. Pluggability at the semantic level is accomplished by the evaluation order rules from Section 3.1 and the type conversion rules from Section 3.2.

4 Examples

Red–black trees. Fig. 4 shows part of a Matchete implementation of red–black trees [13,24]. The `balance` method uses the `T~` deconstructor to disassemble a node into its components, and then reassembles black interior nodes so that the data structure invariant is maintained: each red node must have two black children.

TCP/IP packet headers. Fig. 5 shows bit-level patterns used to recognize TCP/IP packets. The packet header contains the length of the header itself in 32-bit words (Line 5), and the length of the entire packet in bytes (Line 7). The header consists of a fixed 5-word (20-byte) section and an optional variable-length `options` field (Line 18). The `options` length is computed by subtracting the fixed 5-word length from the header length field. If the header length is less than 5 words, the packet is malformed and the pattern match will fail. Similarly, the length of the packet *payload* (Line 19) is a function of the extracted length and header length.

```
1  class Node {
2    static final int R = 0, B = 1;
3
4    int color;
5    Node left, right;
6    int value;
7
8    Node(int c, Node l, int v, Node r) {
9      color = c; left = l; value = v; right = r;
10   }
11
12   T~(int c, Node l, int v, Node r) {
13     c = color; l = left; v = value; r = right;
14   }
15
16   Node balance() {
17     match (this) {
18       T~(B,T~(R,T~(R,Node a,int x,Node b),int y,Node c),int z,Node d):
19         return new Node(R, new Node(B,a,x,b), y, new Node(B,c,z,d));
20       T~(B,T~(R,Node a,int x,T~(R,Node b,int y,Node c)),int z,Node d):
21         return new Node(R, new Node(B,a,x,b), y, new Node(B,c,z,d));
22       T~(B,Node a,int x,T~(R,T~(R,Node b,int y,Node c)),int z,Node d):
23         return new Node(R, new Node(B,a,x,b), y, new Node(B,c,z,d));
24       T~(B,Node a,int x,T~(R,Node b,int y,T~(R,Node c,int z,Node d))):
25         return new Node(R, new Node(B,a,x,b), y, new Node(B,c,z,d));
26     }
27     return this;
28   }
29 }
```

Fig. 4. Red–black tree balancing

5 The Matchete Compiler

We implemented a compiler that translates Matchete source code into Java source code. The result can then be compiled with a regular Java compiler to Java bytecode, and executed on a Java virtual machine together with the Matchete runtime library.

5.1 Background: Rats! and xtc

The Matchete parser is generated by *Rats!*, a packrat parser generator [11]. *Rats!* has a module system for grammars, which permits Matchete to reuse and extend the Java grammar without copy-and-paste. Instead, the Matchete grammar simply includes the Java grammar as a module, changes it with rule modifications, and adds new rules only for new language features. As *Rats!* is scannerless (i.e., it does not separate lexing from parsing), Matchete needs to recognize new tokens only in match clauses without perturbing the Java syntax.

```
1  class IPDumper {
2    void dumpPacket(byte[] b) {
3      match (b) {
4        [[ (4: 4) /* version 4 */
5           (byte headerLength: 4)
6           (int TOS: 8)
7           (int length: 16)
8           (int identification: 16)
9           (byte evil: 1)
10          (byte doNotFragment: 1)
11          (byte moreFragmentsFollow: 1)
12          (int fragmentOffset: 13)
13          (int ttl: 8)
14          (int protocol: 8)
15          (int headerChecksum: 16)
16          (byte[] srcAddr: 32)
17          (byte[] dstAddr: 32)
18          (byte[] options: ((headerLength-5)*32))
19          (byte[] payload: (length-headerLength*4)*8) ]]: {
20         System.out.println("Source␣address:␣" + dotted(srcAddr));
21         System.out.println("Destination␣address:␣" + dotted(dstAddr));
22       }
23       _: System.out.println("bad␣header");
24     }
25   }
26   String dotted(byte[] a) {
27     return a[0] + "." + a[1] + "." + a[2] + "." + a[3];
28   }
29 }
```

Fig. 5. TCP/IP packet header parsing

Matchete uses libraries from the xtc eXTensible C toolkit [10], which includes semantic analyzers for Java and C. Analyzers are visitors that traverse abstract syntax trees with dynamic dispatch, which permits the Matchete compiler to reuse and extend the Java analyzer without copy-and-paste. Instead, the Matchete semantic analyzer is simply a subclass of the Java analyzer, and defines additional `visit` methods for the new grammar productions. xtc also includes support for synchronized traversal of symbol tables. This permits Matchete to populate the symbol table during semantic analysis, then automatically push and pop the same scopes for the same nodes during code generation. One feature of xtc that was particularly helpful in writing the Matchete compiler is the support for concrete syntax, which generates abstract syntax tree snippets from parameterized stencils. This facilitated generation of the boilerplate code required to use, for example, the `java.util.regexp` and `javax.xml.xpath` APIs.

5.2 Type Checking

The Matchete compiler statically checks the semantic rules of Java as specified in the Java Language Specification [9]. This includes checking Java code nested inside of new Matchete constructs, such as value patterns, handler statements, deconstructor bodies, and width expressions of bit-level patterns. Type checking of regular Java code is facilitated by Matchete's language design. For example, binder patterns declare the type of the bound variable, which gets used for type-checking the handler statement.

5.3 Translation

The Matchete compiler is a prototype that demonstrates the existence of a straightforward translation from Matchete to Java. It performs no optimizations, we leave that to future work.

```
1  static int mult(IntList ls) {
2    boolean matchIsDone=false;
3    if (matchete.Runtime.hasDeconstructor(ls , "cons")) {
4      final Object[] subject1= matchete.Runtime.deconstruct(ls, "cons");
5      if (null!=subject1 && 2==subject1.length) {
6        final Object subject2=subject1[0];
7        if (matchete.Runtime.convertible(subject2, Integer.TYPE)
8            && 0==matchete.Runtime.toInt(subject2)) {
9          matchIsDone=true;
10         return 0;
11       }
12     }
13   }
14   if (!matchIsDone && matchete.Runtime.hasDeconstructor(ls, "cons")) {
15      final Object[] subject3=matchete.Runtime.deconstruct(ls, "cons");
16      if (null!=subject3 && 2==subject3.length) {
17        final Object subject4=subject3[0];
18        if (matchete.Runtime.convertible(subject4, Integer.TYPE )) {
19          int h=matchete.Runtime.toInt(subject4);
20          final Object subject5=subject3[1];
21          if (null==subject5 || subject5 instanceof IntList) {
22            IntList t=(IntList) subject5;
23            matchIsDone=true;
24            return h * mult(t);
25          }
26        }
27     }
28   }
29   return 1;
30 }
```

Fig. 6. Code generated for example from Fig. 1

After parsing and type checking, the Matchete compiler has a Matchete abstract syntax tree (AST) with type annotations and a symbol table. Next, it transforms this AST into a Java AST. Finally, it turns the Java AST into Java source code. The resulting Java code calls the `matchete.Runtime` library for services implementing common tasks, in particular, dynamic type conversions. As illustrated in Fig. 2, a match consists of a sequence of patterns, each one determining whether its successor runs. The successor order follows a preorder depth-first traversal of the AST. During code generation, each pattern AST node turns into an if-statement around the code generated for its successor. To generate this structure, the code generator simply traverses patterns in reverse order, plugging each pattern into its predecessor as it creates them.

Fig. 6 shows the code generated for the Matchete code in Fig. 1. The outermost if-statements (Lines 3–13 and 14–28) correspond to the match clauses in Lines 3 and 4 of Fig. 1. They communicate with each other using a synthesized boolean variable `matchIsDone`. This code illustrates the translation of deconstructor patterns (Lines 3–5 and 15–17), value patterns (Lines 7–8), the wildcard pattern (no `if` clause required, cf. Fig. 1), and binder patterns (Lines 18–19 and 21–22). The result of deconstructor patterns get used by multiple children, not just the immediate successor (Lines 17 and 20). Deconstructor patterns are currently implemented with reflection, but we intend to use static types to invoke deconstructors directly. The scope of bindings from binder patterns extends over all their successors. For example, variable `h` declared on Line 19 is in scope for the handler in Line 24. Note that code generation for value and binder patterns requires type analysis: for example, Line 7 checks whether *subject2* is convertible to the type of a value expression. In this case, the value is 0, so the type is obviously `int`. But in general, value patterns can contain arbitrary Java expressions, including calls and arithmetic, so finding their type requires a type checking phase in the compiler.

A deconstructor `p~(`$\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n$`)` translates to a method `p` with no formal parameters. The deconstructor parameters are translated to local variables of the generated method, and the bindings in the deconstructor body are translated to assignments to these variables. The method returns an `Object` array initialized with the deconstructor parameters. A `fail;` statement compiles to `return null;`.

As another example of how the compiler translates composite patterns, consider this XPath pattern for bibliography data: `<.//author/text()>(NodeList authors)`. Fig. 7 shows the Java code that the Matchete compiler generates for this pattern. It

```
1  XPath evaluator = XPathFactory.newInstance().newXPath();
2  try {
3    final Object nodeList = (NodeList) evaluator.evaluate(
4         ".//author/text()",(Node)subject, XPathConstants.NODESET);
5    if (null == nodeList || nodeList instanceof NodeList) {
6      NodeList authors = (NodeList) nodeList;
7      /* successor code */
8    }
9  } catch (XPathExpressionException e) { /* do nothing */ }
```

Fig. 7. Code generated for an XML query

delegates the actual evaluation of the XPath expression to standard Java libraries. Unlike other pattern types, XPath patterns detect match failure by catching an exception. The exception prevents successor patterns from executing, and the empty `catch` block allows control to reach the next match clause, or the end of the match statement if this pattern was in the last match clause.

6 Discussion

One tradeoff we faced when we designed Matchete was how tightly to integrate each kind of pattern matching mechanism. Both regular expressions and XPaths are examples of loosely integrated pattern matching mechanisms. They are integrated, since they can nest and be nested in patterns of other kinds. But they could be more tightly integrated. For example, the regular expression `/([0-9]+) \.([0-9]+)/ (int x, int y)` specifies groups twice: once in the regular expression on the left, and then again in the nested binder patterns on the right. A tight integration would combine the two, so that programmers do not have to rely on counting and ordering to correlate them. The advantage of loose integration is that it does not alter the familiar syntax of the existing matching mechanism, and it allows the implementation to reuse feature-rich optimized libraries.

An example of tight integration is bit-level patterns in Matchete. A syntactic argument for tight integration is that when the different matching mechanisms resemble each other, programmers can amortize their learning curve. On the other hand, tight integration puts the full burden of the implementation on the host language vendor.

At the other end of the spectrum is no integration. For example, Matchete does not directly support file name patterns as used in shells or makefiles. Matchete focuses on covering the most important kinds of matches: on typed structured data (deconstructor), on bit data (bit-level), on semistructured data (XPath), and on text (RegExp). But it leaves out variations of these kind of matches, such as file name patterns. Instead, it provides an extension mechanism (parameterized patterns).

Matchete has to strike a balance between static and dynamic typing. The arguments for and against either are a subject of hot debate and beyond the scope of this paper. But no language design can avoid decisions on this. Matchete's decisions are summarized in Section 3.2 and Table 1. Pattern matching in Matchete is mostly dynamically typed. This felt natural, since patterns are often used to overcome a data representation mismatch. However, it reduces optimization opportunities, which is why Matchete adds hints that can allow compilers to determine types statically in many cases. One advantage of demonstrating pattern matching with little reliance on types is that it is more applicable to dynamically typed host languages—Matchete features could easily be transferred from Java to a scripting language. Note that Matchete pattern matches are strongly typed: when they would violate types, patterns quietly fail instead.

The combination of dynamic typing and source-to-source translation raises concerns about ease of debugging Matchete code. When a match statement does not work as the programmer expected, they need to pinpoint the defect. Matchete uses functionality provided with xtc to inject SMAPs [6] into class files, which allow Java debuggers such as Eclipse or Jdb to work at the level of the original Matchete source code.

7 Conclusions

This paper has introduced Matchete, an extension to the Java programming language with a pattern matching construct that integrates data structure deconstruction, string and bit-level manipulation, and XML queries. Our experience with Matchete suggests that the ability to mix and match different kinds of pattern expressions is powerful and leads to compact and elegant code. The prototype compiler is adequate as a proof of concept, but we are actively working on optimizing the generated code.

Acknowledgements

Robert Grimm and the anonymous reviewers gave helpful feedback on the writing. Mukund Raghavachari, Igor Peshansky, and John Field participated in many of the early design discussions. Robert Grimm supplied the *Rats!* and xtc infrastructure, and Byeongcheol "BK" Lee contributed source-level debugging support to it.

References

1. Back, G.: DataScript: A specification and scripting language for binary data. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, Springer, Heidelberg (2002)
2. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: International Conference on Functional Programming (ICFP) (2003)
3. Clark, J., DeRose, S.: XML path language (XPath) version 1.0. W3C recommendation, W3C (November 1999), http://www.w3.org/TR/1999/REC-xpath-19991116
4. Diatchki, I.S., Jones, M.P., Leslie, R.: High-level views on low-level representations. In: International Conference on Functional Programming (ICFP) (2005)
5. Emir, B., Odersky, M., Williams, J.: Matching objects with patterns. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, Springer, Heidelberg (2007)
6. Field, R.: JSR 45: Debugging Support for Other Languages, http://jcp.org
7. Fisher, K., Gruber, R.: PADS: A domain-specific language for processing ad hoc data. In: Programming Language Design and Implementation (PLDI) (2005)
8. Gapeyev, V., Pierce, B.C.: Regular object types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
10. Grimm, R.: xtc (eXTensible C), http://cs.nyu.edu/rgrimm/xtc/
11. Grimm, R.: Better extensibility through modular syntax. In: Programming Language Design and Implementation (PLDI) (2006)
12. Griswold, R., Poage, J.F., Polonsky, I.P.: The Snobol 4 Programming Language. Prentice-Hall, Englewood Cliffs (1971)
13. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Foundations of Computer Science (FOCS) (1978)
14. Gustafsson, P., Sagonas, K.: Efficient manipulation of binary data using pattern matching. Journal on Functional Programming (JFP) (2006)
15. Harren, M., Raghavachari, M., Shmueli, O., Burke, M.G., Bordawekar, R., Pechtchanski, I., Sarkar, V.: XJ: Facilitating XML processing in Java. In: International World Wide Web Conferences (WWW) (2005)

16. Hosoya, H., Pierce, B.: XDuce: A statically typed XML processing language. Transactions on Internet Technology (TOIT) (2001)
17. Hudak, P., Peyton Jones, S.L., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J.H., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R.B., Nikhil, R.S., Partain, W., Peterson, J.: Report on the programming language Haskell, a non-strict, purely functional language. SIGPLAN Notices 27(5), R1–R164 (1992)
18. Kirkegaard, C., Møller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Transactions on Software Engineering (TSE) (2004)
19. Lee, K., LaMarca, A., Chambers, C.: HydroJ: Object-oriented pattern matching for evolvable distributed systems. In: Object-Oriented Programming, Systems, Languages and Applciations (OOPSLA) (2003)
20. Liu, J., Myers, A.C.: JMatch: Iterable abstract pattern matching for Java. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, Springer, Heidelberg (2002)
21. Meijer, E., Beckman, B., Bierman, G.M.: LINQ: Reconciling objects, relations, and XML in the .NET framework. In: SIGMOD Industrial Track (2006)
22. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
23. Moreau, P.-E., Ringeissen, C., Vittek, M.: A pattern matching compiler for multiple target languages. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, Springer, Heidelberg (2003)
24. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
25. Richard, A.J.: OOMatch: Pattern matching as dispatch in Java. Master's thesis, University of Waterloo (2007)
26. Syme, D., Neverov, G., Margetson, J.: Extensible pattern matching via a lightweight language extension. In: International Conference on Functional Programming (ICFP) (2007)
27. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Principles of Programming Languages (POPL) (1987)
28. Wall, L.: Programming Perl. O'Reilly (1990)
29. Wright, A.K.: Pattern matching for Scheme. The match special form is part of PLT Scheme's MzLib library (1996)