Software Quality Improvement via Pattern Matching

Radu Kopetz, Pierre-Etienne Moreau

Software quality improvement via pattern matching

Radu Kopetz and Pierre-Etienne Moreau

INRIA & LORIA
{Radu.Kopetz, Pierre-Etienne.Moreau}@loria.fr

Abstract. Nested *if-then-else* statements is the most common programming schema in applications like data transformation or data analysis. In most cases, these can be replaced by higher level pattern matching constructs, rendering the code more readable and easier to maintain. We present a tool that integrates strong semantically grounded pattern matching features in JAVA via a plug-in for the Eclipse platform.

1 Introduction

Software systems tend to be bigger and more complex every day. Consequently, software maintenance grows in importance, stressing the need for code that is more readable and easier to maintain. In domains like simplifications of formulae or query optimizations, an important percent of the code is concerned with deciding if we are in a case or another, decisions that are usually implemented using sequences of nested *if-then-else* statements. E-commerce applications are also perfect examples where data retrieval and data analysis play a central role. In many cases they use persistence APIs, such as JPA [3], to map a database on an object-relational model. Then, the analysis and the transformation (to adapt an online catalog to different devices for examples) are performed in JAVA, using a combination of *getters* and *if-then-else* statements.

Nesting *if*-s, along with *else* statements and negations renders the code quite illegible. The use of pattern matching, a well known feature that exists in functional programming languages, is an interesting alternative to improve the quality and reduce the maintenance cost of software. In this paper we propose a seamless integration of advanced pattern matching features in new or already existing JAVA projects, using a plug-in for the Eclipse platform [2].

2 Matching Java objects

Tom language. TOM[1] [1] is an extension of JAVA which adds support for algebraic data-types and pattern matching. An important construct is `%match`, which is parameterized by a list of objects, and contains a list of rules. The left-hand side of the rules are pattern matching conditions (build upon JAVA class

[1] http://tom.loria.fr/

names and variables), and the right-hand side are JAVA statements. Like standard `switch/case` construct, patterns are evaluated from top to bottom, firing each action (*i.e.* right-hand side) whose corresponding left-hand side *matches* the list of objects given as arguments.

For instance, suppose that we have a hierarchy of classes composed of `Account`, from which inherit `CCAccount` (credit card account) and `SAccount` (savings account), each with a field `owner` of type `Owner`. Given two objects `s1` and `s2`, if they are of type `CCAccount`, respectively `SAccount`, and have the same owner's name we want to print this name. If they are both of type `CCAccount`, just print the text `"CCAccount"`:

```
%match(s1,s2) {
  CCAccount(Owner(name)),SAccount(Owner(name)) -> { print(name); }
  CCAccount(_),CCAccount(_) -> { print("CCAccount"); }
}
```

In the above example, `name` is a variable. Using the same variable more than once in the left-hand side of a rule is called *non-linearity*, and denotes that the same value is expected. The `_` is an anonymous variable that stands for anything. The equivalent JAVA code would be:

```
if (s1 instanceof CCAccount) {
  if (s2 instanceof SAccount) {
    Owner o1=((CCAccount)s1).getOwner();
    Owner o2=((SAccount)s2).getOwner();
    if (o1 != null && o2 != null) {
      if ((o1.getName()).equals(o2.getName())) {
        print(o1.getName());
      }
    }
  } else { if (s2 instanceof CCAccount) { print("CCAccount"); } }
}
```

Besides matching simple objects, TOM can also match lists of objects. For instance, given a list of accounts (`List<Account> list`), the following code prints all the names of the credit card accounts' owners:

```
%match(list) {
 AccountList(X*,CCAccount(Owner(name)),Y*) -> { print(name); }
}
```

`AccountList` is a variadic list operator, the variables suffixed by `*` are instantiated with lists (possibly empty), and can be used in the action part: here `X*` is instantiated with the beginning of the list up to the matched object, whereas `Y*` contains the tail. The action is executed for each pattern that matches the subject (assigning different values to variables). Patterns can be non-linear: `AccountList(X*,X*)` denotes a list composed of two identical sublists, whereas `AccountList(X*,x,Y*,x,Z*)` denotes a list that has twice the same element. Another feature of TOM patterns that's worth mentioning is the possibility to

embed negative conditions using the complement symbol '!' [4]. For instance, `!AccountList(X*,CCAccount(_),Y*)` denotes a list of accounts that does not contain a credit card account. Similarly, `!AccountList(X*,x,Y*,x,Z*)` stands for a list with only distinct elements, and `AccountList(X*,x,Y*,!x,Z*)` for one that has at least two distinct elements. There is no restriction on patterns, including complex nested list operators combined with negations. This allows the expression of different algorithms in a very concise and safe manner.

Eclipse plug-in for Tom. Since Tom is a conservative extension of Java, to make it easier to use, we have developed an Eclipse plug-in that provides most of the functionalities available for Java (wizards for new files and projects, edition with coloring and completion, automatic compilation, errors and warnings, *etc.*), making the edition of Tom files (`*.t`) transparent. A Tom program is an alternation between Java code and Tom code. When saving a `.t` file, the compilation is launched automatically, producing a Java file. Actually, the Tom parts are replaced with equivalent Java code, leaving untouched the rest of the code. After editing and saving a `.t` file, the eventual errors and warnings (both Tom and Java ones) are reported in the classical Problem View of Eclipse, as well as in the editor (similar to editing plain Java files). Given any hierarchy of Java classes, the plug-in also provides the necessary support to match against Java objects, as exemplified previously.

3 Behind the scene

Until now, we briefly exposed the matching capabilities of Tom without giving any details on how the `%match` can be transformed into Java code. Actually, the `%match` construct can be used with any Java object, given that some additional information about the structure of the object is provided. Therefore, for any object that we intend to match on, we have to provide a *mapping* — a piece of code that gives Tom the information he needs to test the Java type of the object, the equality between two objects of the same type as well as how to decompose it.

Writing the mappings is not difficult, but involves some prior knowledge about how they work. As we advocate for an effortless integration of pattern matching in Java, the plug-in embeds a *mapping generator* that produces mappings for any set of compiled java classes. We integrated the generator as a two-step wizard: a right click on a `.class` file (or on a folder) and choosing the option *Generate mappings*, opens a window to choose the destination folder and the name of the file that will contain the mappings; pressing *Finish* launches the generator and creates the file. If the user selected a folder to generate the mappings for, then the generator traverses recursively the folder and its subfolders producing mappings for all `.class` files. Then, all we have to do is to add `%include{mapping_file.tom}` in the Tom files where we want to use it.

There are two important characteristics of the generated mappings that are worth mentioning: first of all, for all the classes the generator also produces the

necessary mappings for matching lists of objects of that kind. For instance, given the class `CCAccount`, this allows to write the following code:

```
CCAccount cc = new CCAccount(...);
%match(cc) {
 CCAccount(_) -> { ... }
 ...
}
```

```
List ccList = getCCAccountList();
%match(ccList) {
 CCAccountList(X*,CCAccount(_),Y*) -> { ... }
 ...
}
```

The second important characteristic of the generated mappings is that they offer the full support of the Java polymorphism. Actually, the type of a class is specified in the mappings as the type of the highest class in its hierarchy (the one before `Object`). This is very useful, for instance, when using polymorphic queries in JPA. In our case, `CCAccount` and `SAccount` inherit from the class `Account`, that has a field `accNum` of type `String`. We can write the code:

```
List accounts = getAccountList(); // retrieves all accounts from DB
%match(accounts) {
 AccountList(X*,a@Account(accNum),Y*) -> {
   print(accNum); //prints the number of any account (CCAccount or SAccount)
   %match(a) {
     SAccount(Owner(name)) -> { print("SAccount:" + name); }
     CCAccount(_)          -> { print("CCAccount"); }
   }
}}
```

The notation `@` denotes an alias which stores in `a` the `Account` that is matched.

4 Application scenarios and related work

We presented the `%match` construct that is well suited for integrating pattern-matching facilities in new or already existing projects. Tom is a lot richer, and most important, it offers the possibility to clearly separate the notions of transformation rules, expressed with `%match`, from their application control, encoded using *strategies*. This is more generally a software development methodology, which increases both the development time and reduces maintenance costs through its flexibility.

Tom is an environment for defining transformations. Compared to other term rewriting based languages, like ASF+SDF, Maude, ELAN, an important advantage of Tom is its seamless integration in any Java project, which enables its usage in an industrial context.

Tom is used in several companies to develop applications that range from query optimizers for Xquery, non-trivial transformations of SQL and OLAP queries, to simplifications of logic formulae. Recent users report also the use of Tom for statical analysis of JSP pages.

It is also used in academia for the development of proof assistants, for encoding and verification of security policies, and other different rewrite-based applications.

Tom is a perfect environment for defining DSLs as well, and consequently the Tom compiler is one of the applications developed in Tom. Most of the work of the Tom compiler, as of any preprocessor as a matter of fact, is to perform transformations on an AST (Abstract Syntax Tree) initially produced by the parser, until the Tom code parts are completely transformed into Java ones. Therefore, most of the code is concerned with matching different parts of the AST, and is written entirely using `%match` blocks combined with strategies.

Other languages provide pattern-matching extensions for Java: Scala, Pizza, JMatch, *etc*, but they only provide a basic pattern-matching. More specifically, they lack the list-matching, as well as the negative conditions. Other rule-based languages like JBoss Rules or Jess have *business rules* as their application domain, and not program transformation.

5 Conclusion

We have presented a tool for a smooth integration of pattern matching in Java. It is a mature implementation, used both in academia and industry.

Using such an approach is quite straightforward, even in an existing project: only a few clicks and an include have to be done. After a short learning curve, the use of pattern matching greatly improves the quality of the implementations. By separating the retrieval of data from its transformation, and by making explicit the structure of objects that are manipulated, the code becomes easier to read, to understand, and to maintain. The use of high-level constructs such as list matching or complement symbols also renders the code safer.

While actively working on the integration of graph matching facilities to handle data-structures with cycles, we also study the possibility to make the integration simpler, by automatically generating the mappings and the inclusion statement. This would even require less effort from the programmer to use Tom.

References

1. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 36–47. Springer-Verlag, 2007.
2. E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.
3. M. Keith and M. Schincariol. *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006.
4. C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In *Proceedings of the 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 110–124. Springer Verlag, 2007.