A Practical Study of Control in Objected-Oriented–Functional–Logic Programming with Paisley

Baltasar Trancón y Widemann
Ilmenau University of Technology, DE
baltasar.trancon@tu-ilmenau.de

Markus Lepper
<semantics/> GmbH, DE

Paisley is an extensible lightweight embedded domain-specific language for nondeterministic pattern matching in Java. Using simple APIs and programming idioms, it brings the power of functional–logic processing of arbitrary data objects to the Java platform, without constraining the underlying object-oriented semantics. Here we present an extension to the Paisley framework that adds pattern-based control flow. It exploits recent additions to the Java language, namely functional interfaces and lambda expressions, for an explicit and transparent continuation-passing style approach to control. We evaluate the practical impact of the novel features on a real-world case study that reengineers a third-party open-source project to use Paisley in place of conventional object-oriented data query idioms. We find the approach viable for incremental refactoring of legacy code, with significant qualitative improvements regarding separation of concerns, clarity and intentionality, thus making for easier code understanding, testing and debugging.

Keywords: pattern matching; control flow; embedded domain-specific language; object orientation; refactoring

1 Introduction

The object-oriented paradigm, in its pure form, suffers from strongly asymmetric expressivity with respect to structured data. On the one hand, *constructors* and *factory methods* allow for a term-like notation for the construction of data. On the other hand, the corresponding dedicated tools for the query and deconstruction of data, namely dynamic *type tests*, *casts* and *getter methods*, are not only very different in appearance, but also markedly less expressive and compositional. Programming style patterns and idioms such as *iterators* and *visitors* have been developed to amend the issue, but suffer from restricted applicability and heavy impact on code structure, resulting in imprecise and clumsy practical usage.

By contrast, declarative languages typically support a notation that is directly inverse to, and hence as clear, lightweight, precise and expressive as, data construction: *pattern matching*. The denotational semantics of patterns rely on the invertible algebraic structure of declarative data models. Hence they do not carry over to arbitrary object-oriented data models, whose basic principles such as transcendental identity, mutable state and data abstraction are fundamentally at odds.

Multi-paradigm languages such as Scala [4] have demontrated that object-oriented programming can benefit greatly from an approach to pattern matching that is more operational and hence adequate to imperative program semantics. By contrast, approaches such as JMatch [3], which impose declarative pseudo-semantics on objects and imperative code, have never successfully reached the mainstream.

We have designed Paisley [6] as a lightweight embedded domain-specific language, that is a library of classes and idioms, for pattern matching in Java. Paisley integrates closely with the imperative object-oriented paradigm, even closer than Scala's built-in patterns. It supports user extensions, combinatorial

S. Schwarz and J. Voigtländer (Eds.): 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (WLP'15/'16/WFLP'16).
EPTCS 234, 2017, pp. 150–164, doi:10.4204/EPTCS.234.11

```
public class Pair {

  private Object car, cdr;
  public Pair (Object car,
               Object cdr) {
    this.car = car;
    this.cdr = cdr;
  }
  public Object getCar() {
    return car;
  }
  public Object getCdr() {
    return cdr;
  }
  public static final Object empty;
}
```

```
Pattern⟨Object⟩ isPair = isInstanceOf(Pair.class);
Motif⟨Pair, Object⟩ asPair = forInstancesOf(Pair.class);

Pattern⟨Object⟩ pair (Pattern⟨Object⟩ pcar,
                      Pattern⟨Object⟩ pcdr) {
  return asPair.apply(car.apply(pcar)
                      .and(cdr.apply(pcdr)));
}
Motif⟨Object, Pair⟩ car = transform(Pair::getCar);


Motif⟨Object, Pair⟩ cdr = transform(Pair::getCdr);


Pattern⟨Object⟩ isEmpty = eq(Pair.empty);
```

Figure 1: LISP lists, Java data model (left) and Paisley pattern library (right)

abstractions, nondeterminism by backtracking, encapsulated search, and metaprogramming, without requiring changes to either the Java host language or the code of the data object models to be queried.

In past papers, we have demonstrated how to use Paisley to sanitize legacy query interfaces [5], how to combine nondeterminism and search-plan metaprogramming to solve logical puzzles [7], and how to express complex relational queries in pattern (Kleene) algebra [8], respectively. In summary, Paisley turns Java into an effective functional–logic programming language, without constraining the use of features of the object-oriented host environment. However, the demonstrations have so far focused on encapsulated search, and omitted a different aspect that is ubiquitous in declarative pattern matching: the joint specification of *data and control flow* in case distinctions by pattern matching *clauses*.

The present paper fills this gap. The Java language has adopted, in its recent version 8, mechanisms for local functional programming that are well-integrated with the object-oriented background. We demonstrate how to use these to encode the right hand sides of pattern matching clauses as continuations, such that the ordinary Java means for clause selection, namely conditional statements and short-circuiting Boolean operators, can be used freely and effectively. Unlike for the preceding papers [7, 8], we illustrate the real-world use of this novel expressivity not by constructing a new program for the purpose, but instead by reengineering parts of an existing third-party open-source project. We discuss and evaluate the impact of our modifications both qualitatively and quantitatively.

2 Paisley in a Nutshell

As a running example, consider Figure 1 (left): a simple data model of LISP-style universal lists, implemented in Java in object-oriented textbook style. It is a simple matter to denote the construction of a list, for instance of three elements x, y and z. It is however a very different matter to use the public API to denote the inverse operation: checking whether a given list contains exactly three elements, and if so, extract them as x, y and z, or otherwise do something else. See Figure 2 (top and center). The style

```
list1 = new Pair(x, new Pair(y, new Pair(z, empty)));
```

```
boolean success = false;                                    // [S]
if (list1 instanceof Pair) {                                // [T]
  Pair pair1 = (Pair)list1;                                 // [C]
  Object x = pair1.getCar();                                // [B,P]
  Object list2 = pair1.getCdr();                            // [b,P]
  if (list2 instanceof Pair) {                              // [T]
    Pair pair2 = (Pair)list2;                               // [C]
    Object y = pair2.getCar();                              // [B,P]
    Object list3 = pair2.getCdr();                          // [b,P]
    if (list3 instanceof Pair) {                            // [T]
      Pair pair3 = (Pair)list3;                             // [C]
      Object z = pair3.getCar();                            // [B,P]
      Object list4 = pair3.getCdr();                        // [b,P]
      if (list4 == empty) {                                 // [T]
        succeed(x, y, z);                                   // [R]
        success = true;                                     // [S]
      }
    }
  }
}
if (!success)                                               // [S]
  fail();                                                   // [R]
```

```
Variable⟨Object⟩ x = new Variable⟨⟩(),
                 y = new Variable⟨⟩(),
                 z = new Variable⟨⟩();                      // [B]
Pattern⟨Object⟩ triple = pair(x, pair(y, pair(z, isEmpty)));// [T,C,P,b]
if (triple.match(list1))                                    // [S]
  succeed(x.getValue(), y.getValue(), z.getValue());        // [R,B]
else                                                        // [S]
  fail();                                                   // [R]
```

Figure 2: Construction of a list (top) and its inverse operation with plain Java (center) and Paisley (bottom). For comments see text.

mandated by naïve direct use of the data model API, and prevalent in practice, that is both in teaching and industry, is deficient in several ways:

- It is low-level and of little documentation value regarding the intentions of the programmer.
- It lacks compositionality; although the code structure is repetitive, its fragments cannot be easily reused, nor inspected, understood, analyzed, tested and debugged independently.
- Its sheer verbosity leaves ample room for control and data flow errors.
- It entangles concerns that should, and can, be separated.

In order to illustrate the last point, we have annotated each line of code with the respective operational concern: type and predicate testing [T], type casting [C], binding of variables of interest [B] and of temporary variables [b], projection to subelements [P], success management [S], and finally actual reaction [R]. The design of Paisley is founded on an object-oriented model and API that separates these concerns, and reifies their instances as first-class citizens, as orthogonally as possible.

The following exposition is a summary of the core layer of Paisley, as far as required for appreciation of the novel contributions discussed in the remainder of this paper. They are presented in synopsis in Figure 3. The core-level class and method names are printed in slanted font, to distinguish them from host-level and user-level items. Note that the use of generic types has been simplified for the sake of easy reading. For more details see [6].

Patterns that potentially match target data of type A are instances of class *Pattern*⟨A⟩. At the core of the pattern API is its method **boolean** *match*(A target). The type parameter ensures static type safety, to the usual degree of Java generics. The Boolean return value indicates success. Pure tests [T] implement *match* without side effects, either manually or by wrapping a predicate object with the static factory method *test*. There are generic factory methods for type and equality test, *isInstanceOf*, and *eq*, respectively.

Local success management [S] is implemented by the logical binary operators *and*/*or*. Nondeterminism, such as introduced by *or*, is implemented by explicit backtracking; after a successful initial match, one may call **boolean** *matchAgain*() to obtain further solutions, and iterate as long as successful. Note that the *and* combinator is different from trivial sequential matching; it is fully distributive over backtracking, analogous to the Prolog *comma* operator. Operationally, in p.*and*(q), q is (re)started after each successful match for p. Denotationally, the dependent sum of solutions is formed, which degenerates to the Cartesian product if q is independent of the state of p.

Data flow is by side effects only: Its simplest form is binding [B], implemented by the subclass *Variable*⟨A⟩ that succeeds deterministically and binds the matched target data, such that it can be retrieved if a successful match of the containing pattern has been performed; otherwise the bound value is undefined. Temporary variables [b] can often be elided by means of point-free pattern combination.

Transformative data processing, such as type casting [C] and projection [P], is implemented by contravariant lifting; a data transformation of type $\mathsf{B} \to \mathsf{A}$ is lifted to a pattern transformation of the opposite type $\mathit{Pattern}\langle\mathsf{A}\rangle \to \mathit{Pattern}\langle\mathsf{B}\rangle$. For instance, consider a getter method A getFoo() of class B. The corresponding lifting is a pattern factory of roughly the following functionality:

```
Pattern⟨B⟩ foo(Pattern⟨A⟩ p) {
  return new Pattern⟨B⟩() {
    public boolean match(B target) {
      return p.match(target.getFoo());
    }
  };
}
```

Pattern factories are again reified in Paisley as class *Motif*⟨A,B⟩ with methods *apply* and *then* for application and composition, respectively, thus enabling function-level pattern metaprogramming. The contravariant lifting operator itself is available as a metafactory method *transform*. The function to be lifted is typed with a *functional interface*. This novel feature of Java 8 emulates the structural type operator $(\to)$ in Java's nominal type system, with automatic coercion to compatible functional interfaces. Thus the preceding example can be shortened to the explicit lifting of a function object,

```
Motif⟨A, B⟩ foo = transform(f);
```

```
public abstract class Pattern⟨A⟩ {
  public boolean match(A target);
  public boolean matchAgain();
  public Pattern⟨A⟩ and(Pattern⟨A⟩ p);
  public Pattern⟨A⟩ or(Pattern⟨A⟩ p);
}
```

```
public class Variable⟨A⟩ extends Pattern⟨A⟩ {
  private A value;
  public boolean match(A target) {
    value = target;
    return true;
  }
  public boolean matchAgain() {
    return false;
  }
  public A getValue() {
    return value;
  }
}
```

```
public class Motif⟨A, B⟩ {
  public Pattern⟨B⟩ apply(Pattern⟨A⟩ p);
  public ⟨C⟩ Motif⟨C, B⟩ then(Motif⟨C, A⟩ m);
  public List⟨A⟩ eagerBindings(B target);
  public Iterable⟨A⟩ lazyBindings(B target);
}
```

```
  // miscellaneous
  public static ⟨A⟩ Pattern⟨A⟩ test(Predicate⟨A⟩ p);
  public static Pattern⟨Object⟩ isInstanceOf(Class⟨?⟩... c);
  public static Pattern⟨Object⟩ eq(Object o);

  public static ⟨A, B⟩ Motif⟨A, B⟩ transform(Function⟨B, A⟩ f);
  public static ⟨A⟩ Motif⟨A, Object⟩ forInstancesOf(Class⟨A⟩ c);

  public static ⟨A⟩ Motif⟨A, A⟩ star(Motif⟨A, A⟩ m);
  public static ⟨A⟩ Motif⟨A, A⟩ plus(Motif⟨A, A⟩ m);
```

Figure 3: Paisley core API

```
Motif⟨Object, Object⟩ pairCar = asPair.then(car);
Motif⟨Object, Object⟩ pairCdr = asPair.then(cdr);

Motif⟨Object, Object⟩ nthcdr = star(pairCdr);                    // any pair cell
Motif⟨Object, Object⟩ nth = nthcdr.then(pairCar);                // any element
```

Figure 4: Enumeration of LISP list elements with Paisley

where f is a method reference, B::getFoo, or the equivalent lambda expression, (B b)→ b.getFoo(), which are both significant improvements in expressivity over the pre-Java 8 anonymous class notation:

```
Motif⟨A, B⟩ foo = transform(new Function⟨B, A⟩() {
  public A apply(B b) {
    return b.getFoo();
  }
});
```

Corresponding to the type test operator *isInstanceOf*, there is an analogous pattern transformation for casts, *forInstancesOf*. The pattern transformations of type *Motif*⟨A, A⟩ form a Kleene algebra with operators *star* and *plus*, which we have put to good use for relational programming [8].

In order to apply the Paisley framework to the LISP list data model from Figure 1, pattern operators for the concrete API of class Pair need to be defined. This model layer of Paisley is extensible and typically user-defined; the libary provides only bindings for elementary data, such as classes from the package java.lang. Thus users are free to define their own pattern access style, and to use the functionality and idioms of the Paisley framework pragmatically, in whatever way appears most natural and convenient. Furthermore, since patterns merely bind to the public API of a data model, no privileged access to source code or runtime objects is required. The development of pattern bindings for a data model is modularly independent from the development of internals of the data model itself.

Figure 1 (right) shows a canonical implementation of patterns for class Pair; individual reifications of the type test and cast operation and getter methods, plus a complex pattern constructor as the inverse of the data constructor. Compare with Figure 1 (left). Figure 2 (bottom) shows the use of patterns to express the analog of Figure 2 (center). A clear separation of concerns as three sequential statements has been achieved by the basic idiom of Paisley pattern usage:

1. allocate pattern variables;
2. construct a complex pattern term that handles testing, casting, projection and temporary variables internally and transparently;
3. match, manage success, observe variable bindings and react.

In previous applications [7, 8] we have discussed how to further encapsulate variable allocation and binding and success management for encapsulated search over nondetermistic patterns in logic and functional–logic style, respectively. For instance, enumerating the pair cells or elements of a list nondeterministically is a simple matter of a handful of relational pattern combinators, as shown in Figure 4. The class *Motif* supports concise encapsulated search, exposed in collection or iterator style, via methods *eagerBindings* or *lazyBindings*, respectively. For instance, enumerate all elements of a list as follows:

```
for (Object e : nth.lazyBindings(list))
  processElement(e);
```

```
public class Pattern⟨A⟩ {
  ...
  public Pattern⟨A⟩ andThen(Runnable r);
  public Pattern⟨A⟩ orElse(Runnable r);
}
```

```
  public static ⟨A⟩ boolean testThen(A target, Pattern⟨A⟩ pat, Runnable r) {
    return pat.andThen(r).match(target);
  }
```

```
  public static boolean otherwise(Runnable r) {
    r.run();
    return true;
  }
  public static void ensure(boolean success) {
    if (!success)
      throw new MatchException();
  }
```

Figure 5: Continuation extensions in core Paisley

In the next section we propose a general, concise, structured imperative usage style, appropriate for situations where more fine-grained interleaving of pattern-level and user-level code is required.

3 Control for Paisley

The technical contribution of the present paper is an extension of the Paisley framework, with the purpose of integrating pattern-based control flow – case distinctions where clauses are selected not by plain Boolean conditions but rather by patterns, such that variable bindings may occur as a side effect and result in data flow to the selected clause.

Our solution is true to the Paisley principle that object-oriented programmers should be given declarative expressivity without depriving them of operational intuition. Thus, we require a clean separation of concepts, to the effect that anything that looks like ordinary control flow of the host language actually behaves as such, and that the entanglement of control and data flow that is specific to pattern matching is encapsulated and abstracted appropriately. We find the ideal tool for the job in a recent major addition to the Java language, namely *lambda expressions*.

Consider a more complex variation of the data query example task from the preceding section, namely to succeed for any list, and distinguish the four cases of zero, one, two, and three-or-more elements. Of course, pattern constructs in analogy to Figure 2 (bottom) can be used for each case independently. But the resulting code would be statically and dynamically redundant, because the common matching effort is not shared between cases. The solution is to interleave user-level success management code with pattern-level testing and projection code, in a nested fashion at each level of pair.

Consider Figure 6 (left) ahead for our proposed solution style. The Paisley binding for the data model is extended by adding a functional interface representing the *continuation transform* of the data class

Pair (top). Then a complex clause operator pairThen, along the lines of Figure 2, which handles data flow and match execution internally, can be defined (center). Its function is to call the continuation in the event of success, with the data subitems obtained via pattern variable bindings. The success flag is returned for external cascading management. This yields an idiom for pattern-based clauses, with Boolean expressions of the form

pairThen(list, (car, cdr) → {...})

where the body of the lambda expression that instantiates the continuation interface is executed as a side effect, if and only if the query finds that the inverse of the construction

list = **new** Pair(car, cdr)

applies. Additionally, the Boolean expression can be wrapped in a pattern by means of the *test* factory (bottom), for immediate use

pair((car, cdr) → {...}).*match*(list)

or compositional embedding. The resulting style is concise and elegant; data flows directly to well-scoped variables car and cdr, without need to allocate pattern variables and reason dynamically about their definedness.

Specific continuations, clause operators and pattern wrappers are user-level constructs specific to a data model. For their use, only small extensions to the Paisley core level are required; see Figure 5. We add pattern methods *andThen* and *orElse* to affix a parameterless continuation, of functional interface type Runnable, in the logically obvious way. The generic clause operator *testThen* corresponds to the wrapped form *andThen*. Auxiliary methods *otherwise* and *ensure* convert between **boolean** expressions and **void** continuations, respectively, to compensate for the lack of implicit coercions in Java.

Figure 7 explores the space of expressivity of the contination-based notation. There are two independent degrees of freedom, namely the Java-level choice of conditional statements versus conditional operators, and the Paisley-level choice of Boolean clause expressions versus wrapped pattern applications. Thus we obtained a synopsis of four differently styled but structurally equivalent solutions to the problem posed above. We feel that neither of these styles takes natural precedence, and leave the choice to the taste and convenience of the user.

If only the success of a pattern matters, but no data flow of extracted subitems is required, then a simplified query implementation with parameterless continuations suffices. The corresponding operators are depicted in Figure 6 (right). Note that the body of pairThen is defined verbosely only for the sake of synoptic comparison; it can be constructed more concisely in a single expression:

testThen(target, isPair, r)

Omitting unneccessary continuation parameters is beneficial with respect both to choice of efficient implementation and to temporary variable hygiene.

4 Case Study

We have applied the Paisley style in general, and the control flow notation presented in the previous section in particular, to a real-world, pre-existing, open-source Java project. We have chosen the Kawa 2.1 [2] implementation of the Scheme language, a GNU software package, for this purpose. The rationale is that complex queries of a simple data model, namely the LISP lists introduced as the running example above, is a pervasive topic in Scheme implementations.

```
interface PairContinuation {
  void cont(Object car, Object cdr);
}
```

```
interface Runnable {    // package java.lang
  void run();
}
```

```
static boolean pairThen(Object target,
                        PairContinuation pc) {
  Variable⟨Object⟩ car = new Variable⟨⟩(),
                   cdr = new Variable⟨⟩();
  if (pair(car, cdr).match(target)) {
    pc.cont(car.getValue(), cdr.getValue());
    return true;
  }
  else
    return false;
}
```

```
static boolean pairThen(Object target,
                        Runnable r) {

  if (isPair.match(target)) {
    r.run();
    return true;
  }
  else
    return false;
}
```

```
static Pattern⟨Object⟩ pair(PairContinuation pc) {
  return test(x → pairThen(x, pc));
}
```

```
static Pattern⟨Object⟩ pair(Runnable r) {
  return test(x → pairThen(x, r));
}
```

Figure 6: LISP list continuations (top), clause operators (center) and pattern wrappers (bottom); with data flow (left) and without (right)

```
if (pairThen(list1, (x, list2) → {
  if (pairThen(list2, (y, list3) → {
    if (pairThen(list3, (z, list4) → {
      case3orMore(x, y, z);
    }));
    else case2(x, y);
  }));
  else case1(x);
}));
else case0();
```

```
if (pair((x, list2) → {
  if (pair((y, list3) → {
    if (pair((z, list4) → {
      case3orMore(x, y, z);
    }).match(list3));
    else case2(x, y);
  }).match(list2));
  else case1(x);
}).match(list1));
else case0();
```

```
pairThen(list1, (x, list2) → ensure(
  pairThen(list2, (y, list3) → ensure(
    pairThen(list3, (z, list4) →
      case3orMore(x, y, z))
    || otherwise(() → case2(x, y))))
  || otherwise(() → case1(x))))
|| otherwise(() → case0())
```

```
pair((x, list2) →
  pair((y, list3) →
    pair((z, list4) →
      case3orMore(x, y, z)
    ).orElse(() → case2(x, y)).match(list3)
  ).orElse(() → case1(x)).match(list2)
).orElse(() → case0()).match(list1)
```

Figure 7: Styles of complex list deconstruction with continuations and conditional statements – with clause operators (left) and pattern wrappers (right); as conditional statements (top) and expressions (bottom)

```
static Pattern⟨Object⟩ triple(Pattern⟨Object⟩ x, Pattern⟨Object⟩ y, Pattern⟨Object⟩ z) {
  return pair(x, pair(y, pair(z, isEmpty)));                    // cf. Figure 2
}
static Pattern⟨Object⟩ singleton(Pattern⟨Object⟩ x) {
  return pair(x, isEmpty);
}
static boolean singletonThen(Object target, Runnable r) {
  return testThen(target, pair(any, isEmpty), r);
}
```

Figure 8: Advanced list clause vocabulary

Since we are currently interested in qualitative evaluation, we have not attempted a full reengineering of the Kawa code. Instead, we have exploited the properties of Paisley as a lightweight embedded domain-specific language, that can be used locally without impact on the global structure of a program, and experimented with selected fragments that exhibit typical programming style illustratively.

Figures 9, 10 and 11 (left) each show a code fragment from a source file in `kawa/standard/`. Reaction code has been omitted, indicated by `...` markers. The examples corroborate that our depiction of the naïve, genuinely object-oriented query style in Figure 2 is not a parody but standard practice. The various concerns are entangled in a way that is highly non-compositional and obscures the programmer's intentions, hence the need for a comment. A notable and typical idiom in this style of imperative programming is the reuse of temporary reference variables in queries, and their assignment as a side effect in the midst of expressions. We regard the idiom as intensely obfuscated and error-prone.

The right side of each Figure shows the respective equivalent Paisley code. We have aligned all code vertically to highlight the stuctural correspondence as far as possible, rather than follow the natural structure of complex Paisley expressions by themselves. Because matching for one-element lists is a recurring theme in Figures 10 and 11, we add a corresponding clause operator and pattern wrapper to the model-specific library; see Figure 8. This simple expedient results in considerable code reuse and simplification.

We trust that the direct comparison speaks for itself regarding simplicity, compositionality and clarity, but name a few points of particular interest:

- Recurring patterns can be named mnemonically, simultaneously increasing the reuse and the intentional documentation value of code, and decreasing the room for errors.
- Levels of Paisley abstraction can be mixed; for instance, the lower third of Figure 10 shows a weird query with doubly negated continuation, nested within a perfectly regular encapsulated search.
- Figure 11 demonstrates that (re)assignment of temporary variables can be abstracted away, thus achieving full referential transparency, even for complex code.

Apart from the qualitative and stylistic comparison, we have investigated simple quantitative criteria; see Table 1. For each example, we compare the original and the paisleyfied version with respect to three code metrics:

1. *The number of lines of code pertaining to matching tasks.* Reaction code and context are ignored for this purpose. Note that we have marked some lines of the Paisley version with //∗, to indicate that the line break is solely due to alignment with the more verbose original version, and hence discounted.

Table 1: Quantitative assessment of case study examples

Example	Lines of Code			Cyclomatic Complexity			Temporary Assignments		
	original	Paisley	*saving*	original	Paisley	*saving*	original	Paisley	*saving*
`export`	18	10	*44%*	7	1	*86%*	6	3	*50%*
`module_static`	26	17	*35%*	14	3	*79%*	7	4	*43%*
`IfFeature`	28	16	*43%*	12	2	*83%*	10	3	*70%*

2. *The contribution of matching code to the cyclomatic complexity of the method.* Reaction code and the baseline of one are ignored for this purpose. Note that control-flow branches are moved into pattern structure, most notably the logical connectives *and* and *or*.
3. *The number of assignments to temporary variables during matching.* We regard a variable as temporary only if it is not observed directly by reaction code, and count assignments rather than declarations to account for the variable-reusing style of the original code. Parameters of Paisley continuations count as implied assignments.

We find significant improvements for all three metrics, even if the meaningfulness of lines of code and cyclomatic complexity, both statement-centric metrics, are somewhat diminished in the more expression-centric Paisley style.

5 Conclusion

We have demonstrated how to implement fine-grained integration of Paisley pattern-matching logic and data flow with Java control flow, using the novel lambda expressions to encode the right-hand sides of pattern clauses as continuations. This achievement greatly widens the scope of applicability of Paisley to more complex situations.

With the Kawa case study, we have demonstrated that the Paisley approach solves a practically relevant problem, and can be used effectively to reengineer legacy code in a local and incremental fashion. This is possible because Paisley is lightweight by design, and abstracts only from operational low-level burdens of matching *procedures*, without imposing alternative, declarative semantics on data and pattern *objects*.

5.1 Related Work

The most comprehensive attempt to combine the Java language with nondeterministic pattern matching is JMatch [3]. Their approach is both semantically more ambitious, adding declarative features to the Java type system in the form of so-called *modal interfaces*, and technologically more heavyweight, requiring implementation mechanisms such as coroutines and continuation-passing style transforms; thus it is implemented as a proprietary language extension.

In [1] they have reported reductions of program complexity similar to our present findings. Because of their intrusion into the type system, it is possible to reason statically about the totality of JMatch pattern clauses, whereas our approach shares the fate of most imperative object-oriented code, and can only be feasibly verified informally by inspection and testing. On the other hand, because of the heavy-handed implicit transformation of JMatch programs, the possible interferences with other language features such as non-local control flow, concurrency, instrumentation and debugging are unclear. Furthermore, despite

recent award-winning theoretical publications [1], their implementation is stuck with an obsolete snapshot of the host language Java 1.4 (sic!), and hence of no practical relevance for contemporary software production.

5.2 Outlook

Our treatment of pattern matching in conditional control flow enforces deterministic use: the backtracking method *matchAgain* is not invoked explicitly during clause selection. Only the first solution of a pattern is used, and the continuation is accordingly called at most once. However, this does not preclude *internal* nondeterminism; subpatterns may have to backtrack in order to find that solution. As a future generalization, we consider both loop-like (eager) and iterator-like (lazy) control-flow notations for *reentrant* continuations selected by nondeterministic patterns.

The refactoring changes to the Kawa code base do not produce sufficient regular dynamic coverage in order to evaluate the performance of Paisley pattern matching in relation to naïve implementations. We are looking forward to other case studies where data querying is more predictibly dominant in program running times. In particular, we are curious about the capability of just-in-time compilers to optimize 'hot' Paisley code in realistic scenarios.

As a final optimistic observation, Paisley has proven resilient to evolution of the host language, in marked contrast to JMatch, because it is lightweight and thus syntactically and semantically reducible. We have anticipated the use of functional interfaces, based on common-sense extrapolation from Pizza and Scala, and are now immediately ready to reap the benefits of lambda expressions. Whatever the new features of Java 9 and 10 will be, the odds are that Paisley can profit.

References

[1] C. Isradisaikul and A. C. Myers. "Reconciling Exhaustive Pattern Matching with Objects". In: *ACM Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 2013, pp. 343–353. DOI: 10.1145/2499370.2462194.

[2] *Kawa: The Kawa Scheme language*. Version 2.1. 2015. URL: http://www.gnu.org/software/kawa/.

[3] J. Liu and A. C. Myers. "JMatch: Iterable Abstract Pattern Matching for Java". In: *Practical Aspects of Declarative Languages*. Vol. 2562. Lecture Notes in Computer Science. Springer, 2003. DOI: 10.1007/3-540-36388-2_9.

[4] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. 2nd ed. artima, 2010.

[5] B. Trancón y Widemann and M. Lepper. "Paisley: pattern matching à la carte". In: *Proceedings 5th International Conference on Model Transformation (ICMT 2012)*. Vol. 7307. Lecture Notes in Computer Science. Springer-Verlag, 2012, pp. 240–247. ISBN: 978-3-642-30475-0. DOI: 10.1007/978-3-642-30476-7_16.

[6] B. Trancón y Widemann and M. Lepper. "Paisley: A Pattern Matching Library for Arbitrary Object Models". In: *Software Engineering 2013, Workshopband*. Ed. by S. Wagner and H. Lichter. Vol. 215. Lecture Notes in Informatics. Gesellschaft für Informatik, 2013, pp. 171–186. ISBN: 978-3-88579-609-1. URL: http://www.se2013.rwth-aachen.de/downloads/proceedings/SE2013WS.pdf.

[7] B. Trancón y Widemann and M. Lepper. "Some Experiments on Light-Weight Object-Functional-Logic Programming in Java with Paisley". In: *Declarative Programming and Knowledge Management*. Ed. by M. Hanus and R. Rocha. Vol. 8439. Lecture Notes in Computer Science. Springer-Verlag, 2014, pp. 218–233. DOI: 10.1007/978-3-319-08909-6_14.

[8] B. Trancón y Widemann and M. Lepper. "Interpreting XPath by Iterative Pattern Matching with Paisley". In: *Proceedings 23rd International Workshop on Functional Logic Programming (WFLP 2014)*. Vol. 1335. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 108–124.

```
Object symbol; Translator tr;                                    // context

if (symbol instanceof Pair) {
  // Match (rename name1 name2)
  Pair psym = (Pair) symbol;
  Object symcdr, symcddr; Pair psymcdr;
  if (tr.matches(psym.getCar(), "rename")
      && (symcdr = psym.getCdr()) instanceof Pair
      && ((symcddr = (psymcdr = (Pair) symcdr)
           .getCdr()) instanceof Pair)) {
      Pair psymcddr = (Pair) symcddr;
      Object symcdddr = psymcddr.getCdr();
      Object name1 = tr.namespaceResolve(psymcdr.getCar());
      Object name2 = tr.namespaceResolve(psymcddr.getCar());
      if (symcdddr == LList.Empty
          && name1 instanceof Symbol
          && name2 instanceof Symbol) {
        ...
      }
    }
}
```

```
Object symbol; Translator tr;                                    // context

Variable⟨Symbol⟩ vname1 = new Variable⟨⟩(),
                 vname2 = new Variable⟨⟩();
Motif⟨Symbol, Object⟩ nr = transform(x → tr.namespaceResolve(x))
                           .then(forInstancesOf(Symbol.class));
if (triple(                                                      // *
          test(x → tr.matches(x, "rename")),
          nr.apply(vname1),
          nr.apply(vname2)).match(symbol)) {

  Symbol name1 = vname1.getValue();
  Symbol name2 = vname2.getValue();

    ...

}
```

Figure 9: Example code fragment (`kawa/standard/export.java`, lines 52ff.) in Kawa original (left) and Paisley style (right)

```
Pair st; Translator tr;                                          // context

    Object list = st.getCdr();
    ...
    if (list instanceof Pair
          && (st = (Pair) list).getCdr() == LList.Empty
          && st.getCar() instanceof Boolean) {
        if (st.getCar() == Boolean.FALSE)
            ...
        else
            ...
    } else if (list instanceof Pair
                && (st = (Pair) list).getCdr() == LList.Empty
                && st.getCar() instanceof Pair
                && tr.matches((st = (Pair) st.getCar()).getCar(),
                              Scheme.quote_str)) {
        Object cdr = st.getCdr();
        if (cdr != LList.Empty
              && (st = (Pair) cdr).getCar() instanceof SimpleSymbol
              && st.getCar().toString() == "init-run") {                 // sic!
            ...
        } else {
            ...
        }
    } else {



        while (list != LList.Empty) {
            if (! (list instanceof Pair)
                  || ! ((st = (Pair) list).getCar() instanceof Symbol)) {
                ...
            } else {
                ... // uses (st, (Symbol)st.getCar())
            }
            list = st.getCdr();
        }
    }
```

```
Pair st; Translator tr;                                          // context

    Object list = st.getCdr();
    ...
    if (singleton(                                                // *


                  eq(Boolean.FALSE))
        .andThen(() → ...)
        .or(singleton(eq(Boolean.TRUE))
            .andThen(() → ...))
        .or(singleton(                                            // *

                pair(
                    test(x → tr.matches(x,                        // *
                                        Scheme.quote_str)),

                    pair(
                        test(x → x instanceof SimpleSymbol
                                  && x.toString() == "init-run"),
                        any).andThen(() → ...)
                    .orElse(() → {
                        ...
                    }))))).match(list));
    else {
        Variable⟨Symbol⟩ symbol = new Variable⟨⟩();
        Variable⟨Pair⟩ p = new Variable⟨⟩();
        for (Object r : nthcdr.lazyBindings(list)) {
            if (!asPair.apply(car.apply(asSymbol(symbol))
                              .and(p)).match(r)) {
                ...
            } else {
                ... // uses (p.getValue(), symbol.getValue())
            }

        }
    }
```

Figure 10: Example code fragment (`kawa/standard/module_static.java`, lines 23ff.) in Kawa original (left) and Paisley style (right)

```
Object form; Translator tr;                                          // context

    form = tr.namespaceResolve(Translator.stripSyntax(form));
    if (form instanceof String || form instanceof SimpleSymbol) {
        ...
    }
    if (form instanceof Pair) {
        Pair pair = (Pair) form;
        Object keyword = Translator.stripSyntax(pair.getCar());
        if (keyword == orSymbol || keyword == andSymbol) {
            Object rest = pair.getCdr();
            while (rest instanceof Pair) {
                pair = (Pair) rest;
                ...
                rest = pair.getCdr();
            }
            ...
        }
        if (keyword == notSymbol) {
            Object rest = pair.getCdr();
            if (rest instanceof Pair) {
                Pair pair2 = (Pair) rest;
                if (pair2.getCdr() == LList.Empty)
                    ...
            }
            ...
        }
        if (keyword == librarySymbol) {
            Object rest = pair.getCdr();
            if (rest instanceof Pair) {
                Pair pair2 = (Pair) rest;
                if (pair2.getCdr() == LList.Empty)
                    ...
            }
            ...
        }
    }
```

```
Object form; Translator tr;                                          // context

    isInstanceOf(String.class, SimpleSymbol.class).andThen(() → {
        ...
    })
    .or(pair((keyword, rest) → {

        eq(orSymbol).or(eq(andSymbol)).andThen(() → {

            for (Pair pair : nthcdr.then(asPair).lazyBindings(rest)) {

                ...

            }
            ...
        })
        .or(eq(notSymbol).andThen(() → {

            singletonThen(rest, ()→ {

                ...

            });
            ...
        }))
        .or(eq(librarySymbol).andThen(() → {

            singletonThen(rest, ()→ {

                ...

            });
            ...
        })).match(Translator.stripSyntax(keyword));
    })).match(tr.namespaceResolve(Translator.stripSyntax(form)));
```

Figure 11: Example code fragment (`kawa/standard/IfFeature.java`, lines 42ff.) in Kawa original (left) and Paisley style (right)