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

▷ Baltasar Trancón y Widemann Markus Lepper

Ilmenau University of Technology

semantics GmbH

WFLP
2016-09-13//14

Exercise #1: LISP Lists in Java

```
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;
}
```

Exercise #2: Construct a Triple (x y z)

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

Exercise #3: Deconstruct a Triple (x y z)

```
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
```

B	ext. binding
b	int. binding
C	coercion
P	projection
T	testing
S	success man.
R	reaction

Embarassing Questions

1. What is wrong with this code?
   - complexity
   - entanglement of concerns
   - lack of compositionality
2. What is essential about the complexity?
   - in textbook OO style: *everything*
   - in programming at large: **very little**
3. Can we do better?
   - what about declarative languages?
   - can't we use pattern matching too?
   - what about semantics?

Exercise #4: Deconstruct a Triple (x y z), Nicely

```
Variable<Object> x = new Variable<>(),
                 y = new Variable<>(),
                 z = new Variable<>();                B
Pattern<Object> triple =

  pair(x, pair(y, pair(z, isEmpty)));                 bCPT

if (triple.match(list1))                                  S
  succeed(x.getValue(), y.getValue(), z.getValue());  B     R
else                                                      S
  fail();                                                R
```

The Paisley Way of Pattern Matching

- Lightweight embedded DSL in Java (subst C#, . . .)
  - library + idioms + extension guidelines
- No restrictions on host OO semantics
  - data abstraction
  - transcendental identity
  - mutable state
  - operationally grounded
  - . . .
- Reify patterns as objects

  syntax object graph
  semantics operational, API

- Layered codebase

  core separation of concerns, combinators (generic)
  binding (CPT)-patterns for a datatype (custom)

Basic API #1

```
class Pattern<A> {
  boolean match(A target);
  boolean matchAgain();
}

class Variable<A> extends Pattern<A> {
  private A value;
  A getValue();
}
```

- success management & nondeterminism (backtracking)
- variable binding by side effect (assignment)

Basic API #2

- Primitives any, eq, isInstanceOf, . . .
- Combinators and, or, star, plus, . . .
- Liftings
  - predicates to test patterns
  - functions to pattern transforms (contravariant)

Exercise #3.5: LISP List Patterns in Java/Paisley

```
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);

Motif<Object, Object> pairCar = asPair.then(car);
Motif<Object, Object> pairCdr = asPair.then(cdr);

Motif<Object, Object> nthcdr = star(pairCdr);
Motif<Object, Object> nth = nthcdr.then(pairCar);
```

Some Advanced Features

- Host-level metaprogramming
  - Search plans for cryparithmetic puzzles [WFLP'13]
- Kleene algebra of pattern endomorphisms = relational programming
  - XPath interpreter [WFLP'14]

Pattern Matching Clauses

$$\textbf{case}\ a\ \textbf{of}\ \{p(x, q(y)) \to e; \cdots\}$$

$$\Big(\big(\lambda\, xy.\, p(x, q(y))\big)^{-1} \mathbin{;} (\lambda\, xy.\, e) \oplus \cdots\Big)\, a$$

- Pattern clauses introduce local variables
- Joint control&data flow to rhs

Java 8 Goes Functional

- Functional interfaces

```
interface myFun { // emulates type A -> B
  B foo(A a); // name arbitrary
}
```

- Lambda expressions

```
MyFun f = (a) -> new B(a);
```

- Method references

```
class A {
  B getX();
}
MyFun g = A::getX;
```

Boilerplate

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

  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;
 }
```

Exercise #5: Deconstruct a Triple (x y z), Again

```
if (pairThen(list1, (x, list2) -> {
  if (pairThen(list2, (y, list3) -> {
    if (pairThen(list3, (z, list4) -> {
      success(x, y, z);
    }));
  }));
}));
...
```

- one of several idiomatic suggestions
  - alternatively use || instead of **if**
- custom continuations for complex patterns

Test Case: Kawa

- GNU implementation of Scheme on JVM
  - open-source Java project
  - pervasive LISP lists (exactly as shown)
  - lots of deconstruction code (worse than shown)
- Incremental refactoring
  - define custom Paisley bindings
  - replace imperative code by pattern matching
  - run regression tests
  - inspect & evaluate

Preliminary Results

Example	Lines of Code			Cyc. Complexity			Temp. Ass.		
	orig	P	*save*	orig	P	*save*	orig	P	*save*
`export`	18	10	*44%*	7	1	*86%*	6	3	*50%*
`m_static`	26	17	*35%*	14	3	*79%*	7	4	*43%*
`IfFeature`	28	16	*43%*	12	2	*83%*	10	3	*70%*

- Comparable to more intrusive declarative approaches with custom compilers (JMatch)
- Analysis far from complete
  - no coverage results
  - no performance results

Summary

- Nondeterminstic pattern matching in Java
  - reification of imperative queries
- Lightweight implementation
  - small library (1–2 kLoC)
  - data binding extensions
  - programming idioms
- Exploit lambda expressions for control
  - joint control&data flow to PM clauses

Outlook

- More complete case study
  - metrics
  - practical impact
- Experiment with control idioms
- Measure performance
- Study optimization
  - particularly for deterministic patterns
  - JIT compiler?