Functional Perl: Programming with Recursion Schemes in Python

Robert J. Simmons   Nels E. Beckman
Carnegie Mellon University
{rjsimmon,nbeckman}@cs.cmu.edu

Dr. Tom Murphy VII, Ph.D.
The tom7.org foundation
{tom7}@tom7.org

Abstract

Algorithms that are fundamentally expressions of structural induction over a polynomial data type are famously awkward to implement in object-oriented programming languages, leading to three-day-bucket hacks like the "Visitor Pattern." We show that the exception-handling mechanism present in most object-oriented languages already suffices.

Conventional wisdom dictates that exceptions are in essence about the non-local propagation of errors. Upon close scrutiny, we find the contrary: exceptions are fundamentally about unchecked depth-1 structural pattern matching.

We give examples of this programming idiom in Python, Perl, and Java. JavaScript requires a language extension, which we build and deploy. We then show how two styles by which this style may be re-integrated into programming languages in the ML family. We conclude by suggesting design changes that would facilitate the use of this idiom in other languages.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Standards

General Terms Algorithms, Design, Languages, Standardization

Keywords pattern matching, recursion schemes, structural induction, python, perl, java, javascript, types

1. Introduction

Pattern matching is a convenient way of representing polynomial datatypes, which can be informally thought of as a generalization of tree-like data. An extremely simple example is an abstract syntax for a language with integers and addition:

```
datatype tm =
  Int of int
| Plus of tm * tm
```

The big-step evaluation semantics for this language is quite straightforward:

$$\frac{tm_1 \Downarrow x_1 \quad tm_2 \Downarrow x_2 \quad x_1 + x_2 = x_3}{tm_1 + tm_2 \Downarrow x_3} \qquad \frac{}{\mathsf{int}(x) \Downarrow x}$$

Using the pattern matching syntax present in the ML family of languages, the implementation of this big-step evaluation semantics is similarly straightforward:

```
fun eval tm =
  case tm of
    Int x => x
  | Plus (tm1, tm2) => eval tm1 + eval tm2
```

We can also similarly implement a small step semantics for this language as defined by the following rules:

$$\frac{}{\mathsf{int}(x)\ \mathsf{value}}$$

$$\frac{tm_1 \mapsto tm_1'}{tm_1 + tm_2 \mapsto tm_1' + tm_2} \qquad \frac{tm_1\ \mathsf{value} \quad tm_2 \mapsto tm_2'}{tm_1 + tm_2' \mapsto tm_1 + tm_2'}$$

$$\frac{x_1 + x_2 = x_3}{\mathsf{int}(x_1) + \mathsf{int}(x_2) \mapsto \mathsf{int}(x_3)}$$

The ML implementation is a straightforward implementation of these inference rules, with the exception of the need to define an additional `cast` function that we call to get the value of the enclosed integers once we know the terms to be a values.[1]

```
exception Stuck

fun value tm =
  case tm of
    Int x => true
  | _ => false

fun cast tm =
  case tm of
    Int x => x
  | _ => raise Stuck

fun step tm =
  case tm of
    Int x => raise Stuck
  | Plus(tm1, tm2) =>
    if value tm1
    then if value tm2
         then Int(cast tm1 + cast tm2)
         else Plus(tm, step tm2)
    else Plus(step tm1, tm2)
```

[1] In a language without static typing and with distinct varieties of values, these casts can, of course, fail if we get the *wrong variety* of value.

```
interface tm
  { public <T> T accept(Visitor<T> visitor); }
class Int implements tm {
  int x;
  public Int(int x) { this.x = x; }
  public <T> T accept(Visitor<T> v)
    { return v.visit(this); }
}
class Plus implements tm {
  tm e1;
  tm e2;
  public Plus(tm e1, tm e2) {
    this.e1 = e1;
    this.e2 = e2;
  }
  public <T> T accept(Visitor<T> v)
    { return v.visit(this); }
}

interface Visitor<T> {
  public T visit(Plus p);
  public T visit(Int i);
}

public class VisitorExample {
  static int eval(tm tm) {
    return tm.accept(new Visitor<Integer>(){
      public Integer visit(Int i) { return i.x; }
      public Integer visit(Plus p) {
        return p.e1.accept(this)
            + p.e2.accept(this);
      }
    });
  }

  public static void main(String[] args) {
    tm ex_1 =
      new Plus(new Plus(new Plus(new Int(1),
                                 new Int(2)),
                        new Int(3)), new Int(4));
    System.out.println
      ("Evaluating: Plus(Plus(Plus(1,2),3),4) - "
       + eval(ex_1));
  }
}
```

Figure 1. Case study implemented with the Visitor Pattern in Java.

While there is a somewhat more pleasant way to implement the Step function using nested pattern matching, we will only consider the "shallow" pattern matching we see here in this paper.

1.1 Object-Oriented languages and the Visitor Pattern

While ML-family languages suggest describing different types of syntax as different branches of a subtype, most object oriented languages suggest describing different types of syntax as different objects that either implement the same interface or extend the same subclass. The traditionally understood way of performing elegant recursion over these structures is known as the Visitor Pattern; the implementation of big-step evaluation using the Visitor Pattern can be seen in Figure 1.

The Visitor Pattern, whatever its merits may be, is certainly quite different in character from the depth-1 pattern matching used in our ML example. However, because the *exception handling mechanism*, which Java also incorporates, is fundamentally about unchecked depth-1 structural pattern matching, we can seek to use exception handling to provide a similar sort of depth-1 structural pattern matching to the Java language and other object-oriented programming languages. The rest of the paper explores this possibility.

```
class tm:
    pass
class Int(tm):
  def __init__(self, x):
      self.x = x
class Plus(tm):
  def __init__(self, e1, e2):
    self.e1 = e1
    self.e2 = e2

def eval(tm):
  try: raise tm
  except Int: return tm.x
  except Plus: return eval(tm.e1) + eval(tm.e2)

def value(tm):
  try: raise tm
  except Int: return True
  except: return False

def step(tm):
  try: raise tm
  except Plus:
    if(not value(tm.e1)):
      return Plus(step(tm.e1), tm.e2)
    elif(not value(tm.e2)):
      return Plus(tm.e1, step(tm.e2))
    else: return Int(tm.e1.x + tm.e2.x)

ex_1 = Plus(Plus(Plus(Int(1),Int(2)),
                Int(3)),Int(4))
print ("Evaluating:  Plus(Plus(Plus(1,2),3),4) - "
       + str(eval(ex_1)))
print ("Stepping x3: Plus(Plus(Plus(1,2),3),4) - "
       + str(step(step(step(ex_1))).x))
```

Figure 2. Python pattern matching case study.

2. Discovery and implementation in Python

Imagine we have the abstract syntax of our example encoded in Python in the standard way, as a class hierarchy where subclasses `Int` and `Plus` extend the basic abstract syntax class `tm`.

```
class tm:
  pass
class Int(tm):
  def __init__(self, x)
  self.x = x
class Plus(tm):
  def __init__(self, e1, e2)
  self.e1 = e1
  self.e2 = e2
```

The big-step evaluation semantics of this language could be written by explicitly using `isinstance` tests, but the repeated use of `elif isinstance(tm, ...)` is bulky and unsatisfying, which only becomes more true when the code has many, many branches:

```
def eval(tm):
  if isinstance(tm, Int):
    return tm.x
  elif isinstance(tm, Plus):
    return eval(tm.e1) + eval(tm.e2)
  else: raise RuntimeError
```

Note the final line of the above code, which raises a runtime error if the evaluated term is neither an `Int` nor a `Plus`. This is important in case we extend the abstract syntax tree with more branches; we want something equivalent to the "nonexhaustive match exception" in an ML-family language.

2.1 Utilizing the exception handling mechanism

The first key observation for introducing our idiom is the fact that every single object in Python can be treated as an exception and thrown; the language designers suggest that exceptions derive from the `Exception` class, but there is no enforcement of this mechanism. The second key observation is that the branches of an exception handling call can check the object's class tag when deciding which branch of the exception handling to consider. The motivation is to allow, say, a divide-by-zero error or a user-input error to be both caught at the same place but handled in different ways; we can use it, however, to do exactly the `isinstance` test from before in a more concise, pleasing notation:

```
def eval(tm):
  try: raise tm
  except Int: return tm.x
  except Plus: return eval(tm.e1) + eval(tm.e2)
```

Furthermore, this form of evaluation introduces a natural notion of a nonexhaustive match exception: if we add new kinds of terms (say, a multiplication) and don't handle them in the `eval` function, the *very abstract syntax tree node* that we did not handle will be thrown as an exception: an error that potentially contains much more helpful information than an ML `Match` exception!

2.2 Adaptation in Java

The relatively-more-strongly-typed language Java also allows for the expression of our idiom by simply allowing every term to extend the class `RuntimeException`. Technically, all that is necessary for the use of our idiom is `Exception`, but by the use of `RuntimeException` we avoid dealing with compiler warnings caused by lack of exhaustiveness checking (similar to the use of the MLton SML compiler's `-disable-ann nonexhaustiveMatch` option).

Our case study can be seen implemented in Java in Figure 3. It is particularly critical that, in Java, the exception handling mechanism handles appropriately casting the term `tm` to an integer `Int i` or an addition `Plus p`. The typed nature of Java makes the repeated `instanceof` tests much more painful than they were in Python, because we must not only do repeated if-then-else statements but also downcast terms in branches where we *statically know that the cast is safe already* as a result of the `instanceof` test. The exception-handling idiom is free of this particular annoyance.

```
class tm extends RuntimeException {}
final class Int extends tm {
  final Integer x;
  Int(Integer x) { this.x = x; }
}
final class Plus extends tm {
  final tm e1;
  final tm e2;
  Plus(tm e1, tm e2) {
    this.e1 = e1;
    this.e2 = e2;
  }
}

public class PatternMatch {

  static Integer eval(tm tm) {
    try{throw tm;}
    catch(Int i) { return i.x; }
    catch(Plus p)
      { return eval(p.e1) + eval(p.e2); }
  }

  static Boolean value(tm tm) {
    try{throw tm;}
    catch(Int i) { return true; }
    catch(Plus p) { return false; }
  }

  static tm step(tm tm) {
    try{throw tm;}
    catch(Plus p) {
      if( !value(p.e1) )
        return new Plus(step(p.e1), p.e2);
      else if( !value(p.e2) )
        return new Plus(p.e1, step(p.e2));
      else
        return new Int
          ( ((Int)p.e1).x + ((Int)p.e2).x );
    }
  }

  public static void main(String[] args) {
    tm ex_1 =
      new Plus(new Plus(new Plus(new Int(1),
                                 new Int(2)),
                        new Int(3)), new Int(4));
    System.out.println
      ("Evaluating: Plus(Plus(Plus(1,2),3),4) - "
       + eval(ex_1));
  }
}
```

Figure 3. Java pattern matching example.

3. Implementation via language extension

Some languages do not have the essential aspect required for case analysis to be implemented in the exception-handling mechanism. We describe an extensive effort to implement this functionality in JavaScript, as well as commenting on a language where that extensive effort had already been done: Perl.

3.1 JavaScript

JavaScript is a curly-like language oriented on the concept of JavaScript objects, which are functions [1]. The typical way to implement pattern matching would be to use a series of `instanceof` tests, like in Java or Python. As before, we'd like to improve this code to something like what appears in Figure 4.

Unfortunately this does not work because it is not legal JavaScript code. It doesn't parse or run or anything. A typical blunder: JavaScript denies the nature of exceptions by only allowing them to

```
var Int = function(i) { this.i = i; };
var Plus = function(tm1, tm2) {
    this.tm1 = tm1; this.tm2 = tm2;
};

function big_eval(tm) {
  try {
    throw tm;
  } catch (Int(e)) {
    return e;
  } catch (Plus(e)) {
    return new Int(big_eval(e.tm1).i +
                   big_eval(e.tm2).i);
  } catch (e) {
    throw 'stuck';
  }
}
```

Figure 4. Desired JavaScript code.

```
function big_eval(tm) {
  try {
    throw tm;
  } catch (e) {
    throw 'stuck';
  } with (Int(e)) {
    return e;
  } with (Plus(e)) {
    return new Int(big_eval(e.tm1).i +
                   big_eval(e.tm2).i);
  }
}
```

Figure 5. Big-step evaluator, using the language extension.

be used for the non-local propagation of errors, requiring a different language feature to be used for pattern matching (`instanceof`). In many languages we would be at an impasse. Luckily, the authors of JavaScript had the foresight to foresee the need to improve the language; it is possible for us to build language extensions within our programs. We can create a full-featured exception mechanism that we can then use to encode our programming technique.

The extension works by co-opting the rarely-used `with` keyword to create a new iterated, pattern-matching `try` ... `catch` construct. The above big-step evaluation code can then be written as in Figure 5. This is basically the same as in the previous languages, except as a stylistic choice the required default case is written first (it is legal for it to be blank or to re-raise the same exception).

You can't run this code because it is not legal JavaScript. If you try to run it JavaScript will be like, huh? because this is not what the `with` keyword does. To run the code you must first enhance the `big_eval` function using our language extension:

```
  var tm = new Plus(new Int(5), new Int(2));
  var f = big_eval.enhance();
  alert(f(tm).i);
```

The `enhance` property is added to every function by our language extension. It enhances the semantics of the language to add the iterated `try`... `catch`... `with` construct. Running `enhance` returns a new function that works right.

The extension works by converting the function value to source code, modifying the source code, and then evaluating the source code to get a new function value.

```
Function.prototype.enhance = function() {
  var s = '' + this;
  var name;
  if (s.indexOf('function (') == 0) {
    s = 'function $(' + s.substr('function ('.length);
    name = '$';
  } else {
    name = /function *([a-zA-Z0-9_$]+) *\(.*/m.
           exec(s)[1];
  }
  var s = '(function(){' + rewrite(s) +
         '\nreturn ' + name + '})()';
  return eval(s);
};
```

A function value's source code might be a function declaration or a function expression.[2] We put it in a normalized declaration form, giving the function the name $ if it doesn't have one. We wrap the function declaration in another function and call it immediately, because functions are the only way[3] to introduce new scopes in JavaScript (here thankfully the designers understood that the essence of functions is scope delineation).

The bulk of the language extension is implemented in the recursive `rewrite` function. The function rewrites a source code string; the source code can represent a declaration or an expression or anything. Using strings to represent data helps us avoid tricky type errors that can arise when there's a difference between expressions and declarations. The implementation of `rewrite` appears in Figure 7.

The rewrite function parses the source code to find appearances of the `catch` keyword, then takes any `with` blocks that appear after them, and rearranges the code just so. It correctly handles the case that uses of the extension are nested, as well as recursively-enhanced functions. The details of the implementation are subtle, mainly having to do with fiddly bits. An example may help the reader; the big-step evaluation from Figure 5 is translated to the code in Figure 7.

It is worth noting in a smaller font that the language extension does not handle the case that source code appears inside string or regular-expression literals. Another way to put this is that the language extension successfully ferrets out not just *uses* of the new feature but *mentions* of it as well.

3.2 Perl

Because of the pun on "Functional Pearl" we really wanted to implement this in Perl, but none of the co-authors really wanted to learn Perl and we couldn't find Jason [4].

However, while Perl lacks the instanceof-checking exception capability that is critical to our purposes, there is a Perl module, `Error.pm`, that appears to enrich Perl with the necessary features by a strategy similar to the one we investigated for JavaScript.[4] Unfortunately the name of this module also describes what happens when the authors tried to use it.

4. Re-incorporation in ML

Of course, this observation that case analysis can be implemented by pattern matching can be ported back into Standard ML. Standard ML includes a single *extensible* datatype, named `exn`, which also

[2] It might also be a native function, in which case we cannot enhance it. This is okay, because native functions probably did not use our language extension.

[3] One can also use the `with` keyword but that might mess up the language extension, like if someone tries to enhance an already-enhanced function, or the enhancer function itself.

[4] http://search.cpan.org/ shlomif/Error-0.17016/lib/Error.pm

```
function next(s) {
  var depth = 0;
  for (var i = 0; i < s.length; i++) {
    if (s[i] == '{' || s[i] == '(') {
      depth++;
    } else if (s[i] == '}' || s[i] == ')') {
      depth--;
      if (depth == 0) return { p: s.substr(0, i + 1),
                               s: s.substr(i + 1) };
      if (depth < 0) throw 'parse error';
    }
  }
  throw 'parse error 2';
}

function rewrite(s) {
  var out = '';
  for(;;) {
    var pos = s.indexOf('catch');
    if (pos >= 0) {
      pos += 'catch'.length;
      out += s.substr(0, pos);
      s = s.substr(pos);
      var parens = next(s);
      var body = next(parens.s);
      out += parens.p;
      var def = body.p;
      out += '{\nif(0);';
      for (;;) {
        s = /[ \n\t]*([\s\S]*)/m.exec(body.s)[1];
        if (s.indexOf('with') == 0) {
          var s = s.substr('width'.length);
          var pp = next(s);
          var body = next(pp.s);
          var ce = pp.p.substr(1, pp.p.length - 2);
          var ctor = ce.substr(0, ce.indexOf('('));
          var e =
              next(ce.substr(ce.indexOf('('))).p;
          out += ' else if (' + e + ' instanceof ' +
              ctor + ') ' + rewrite(body.p);
          s = body.s;
        } else {
          break;
        }
      }
      out += ' else ' + rewrite(def) + '}';
    } else {
      out += s;
      break;
    }
  }
  return out;
}
```

Figure 6. The `rewrite` function, which implements the bulk of the language extension.

happens to be the type of ML exceptions. A typical elegance: ML emphasizes the dual purpose of exceptions as both a mechanism for non-local error propagation and for pattern matching by using the `exception` keyword to extend the `exn` datatype:

```
  exception Int of int
  exception Plus of exn * exn
```

```
(function(){function big_eval(tm) {
    try {
        throw tm;
    } catch (e){
if(0); else if ((e) instanceof Int)  {
        return e;
    } else if ((e) instanceof Plus)  {
        return new Int(big_eval(e.tm1).i +
                       big_eval(e.tm2).i);
    } else  {
        throw "stuck";
    }}}
return big_eval})()
```

Figure 7. The automatically-enhanced version of Figure 5.

```
structure PatternMatch = struct

  exception Int of int
  exception Plus of exn * exn

  fun eval tm =
    (raise tm) handle
      Int x => x
    | Plus (tm1, tm2) => eval tm1 + eval tm2

  fun value tm =
    (raise tm) handle
      Int x => true
    | _ => false

  fun cast tm =
    (raise tm) handle
      Int x => x

  fun step tm =
    (raise tm) handle
      Plus(tm1, tm2) =>
      if value tm1
      then if value tm2
           then Int(cast tm1 + cast tm2)
           else Plus(tm, step tm2)
      else Plus(step tm1, tm2)

  val main =
    let val ex_1 =
      Plus(Plus(Plus(Int 1,Int 2),Int 3),Int 4)
    in
     print
     ("Evaluating:  Plus(Plus(Plus(1,2),3),4) - "
      ^ Int.toString (eval ex_1) ^ "\n\n");
     print
     ("Stepping x3: Plus(Plus(Plus(1,2),3),4) - "
      ^ Int.toString
             (cast(step(step(step(ex_1)))))
      ^ "\n\n")
    end
end
```

Figure 8. Re-implementation into SML.

The SML parser requires that `(raise tm)` be surrounded by parenthesis, but otherwise the code retains the essential character, as shown in Figure 8. Of course, because the extensible datatype `exn` is itself amenable to standard case-analysis, we could instead define our syntax as branches of the `exn` as we do in Figure 8 and still do a regular pattern-match on it:

```
fun eval tm =
  case tm of
    Int x => x
  | Plus (tm1, tm2) => eval tm1 + eval tm2
```

There are two problems with this approach, however. First, in the case of an unhandled piece of syntax we get an uninformative `Match` exception instead of the offending piece of syntax itself. Second, we because SML attempts to do exhaustiveness checking on case analysis, every case analysis on an `exn` that does not have a catch-all case runs afoul of the static exhaustiveness checking. Getting a lot of non-exhaustive match warnings when compiling doesn't bother some people, but it can be annoying [3].

5. Conclusion and future work

There is much work to be done in exploring the beautiful synthesis of exception-handling and pattern matching. For instance, a serious shortcoming of our exception-handling case analysis is that it does not allow *nested* case analysis, which suggests a new language feature – nested exception handling – for Java/Python family languages.

Similarly, the exception-handling idiom allows Java/Python family languages to do a particularly crazy duck-typing-ish thing. Say I want to just enumerate all the integer subterms of my expression, which is extended from the example in the paper to include `Plus`, `Minus`, `Times`, and `Div`. As long as I called the left hand subterm `e1` and the right-hand subterm `e2` across these different terms, I can write this code very concisely in Python:

```
def subterms(tm):
  try: raise tm
  except Int: print "Subterm: " + str(tm.x)
  except (Plus, Minus, Times, Div):
    subterms(tm.e1)
    subterms(tm.e2)

ex_1 = Plus(Times(Div(Int(1),Int(2)),
                Int(3)),Minus(Int(4),Int(9)))
subterms(ex_1)
```

Similarly, in Java this could be done by making all the binary operations extend a `tmBinary` class that, in turn, extends the `tm` class. In JavaScript, we can continue our approach of implementing Java- and Python-like extensions via syntactic rewrites and then use them in an idiomatic way to simulate a native pattern matching feature. We might think to ask that such functionality be incorporated into ML family languages, were it not for the fact that *it's already there* by way of one of the wackier SML-NJ-specific extensions of Standard ML, "or-patterns," which apparently aren't any more insane insane than focusing was already [2].

```
fun subterms tm =
  case tm of
    Int x =>
    print("Subterm: " ^ Int.toString x ^ "\n")
  | (Plus(t1,t2) | Times(t1,t2) | Mult(t1,t2)
      | Div(t1,t2)) => (subterms t1; subterms t2)
```

For a larger code example (the implementation of the static and dynamic semantics of Plotkin's PCF), see our extended technical report [5].

Acknowledgments

Milo Polte was instrumental in the discovery of this idiom by encouraging the first author to learn Python. Neel Krishnaswami was the first person who observed that this discovery was probably ridiculous, and Noam Zeilberger reminded the first author what or-patterns are and how they work.

References

[1] ECMA. ECMAScript language specification. Technical Report ECMA-262, ECMA, December 1999.

[2] N. R. Krishnaswami. Focusing on pattern matching. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 366–378, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480927.

[3] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.

[4] J. Reed. Gd and l - systems. *The Perl Journal*, 3(1), 1998.

[5] R. J. Simmons. Functional Perl: Recursion schemes in Python (extended case study). Documented code example(s). ACH-BOVIK-2010-003, Association for Computational Heresy, Pittsburgh, Pennsylvania, 2010.