Pattern matching in concatenative programming languages

Daniel Ehrenberg
Computer Science Department
Carleton College
Northfield, MN 55057
ehrenbed@carleton.edu

Abstract

Pattern matching is a useful, convenient tool for processing algebraic data types in functional programming languages. This paper discusses a method for porting pattern matching to a family of programming languages called *concatenative*, which does not use named variables. The system is not fixed, like a traditional pattern matching system, but rather arbitrarily extensible: any function can be used as a pattern if the right information is given to the pattern matcher. This generalization of pattern matching has been useful in libraries for XML processing and unit conversion.

1 Introduction

The idea of pattern matching is that, to see the contents of an object, the same syntax can be used as in creating the object, only on the left hand side of a binding rather than the right. This makes it very easy to write declarative code which manipulates functional data structures. In many functional programming languages, including Haskell, pattern matching is a fundamental built-in construct. Pattern matching systems have also been constructed as macro libraries in languages like Common Lisp.

Certain programming languages, termed concatenative programming languages, express a program in terms of the composition of several smaller programs, typically without the use of named variables. Instead, a stack is used to pass data around [7]. This paper uses a particular concatenative programming language called Factor [2]. Aside from the unusual property of being concatenative, Factor is fairly similar to Lisp. In Factor, objects are allocated on a garbage-collected heap, as in other functional programming languages. Factor is dynamically typed.

It is not immediately obvious how to do pattern matching in a concatenative programming language, when variables are not used. A fundamental rethink of pattern matching, as calculating the inverse of a function, leads to a clear solution to this problem.

We have constructed an efficient pattern matching library for Factor . A composition of functions, each of whose inverse is known, can easily be inverted using the formula $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$. Functions that are used in pattern matching are typically not surjective, meaning that the inverse must be a partial function.

This conception of pattern matching naturally leads to a significant generalization of existing systems. One simple thing that this new system can do is "pattern match" over certain simple bijective arithmetical formulas, inverting them. The system has been used to construct a practical library for encoding numbers with units. With this library, centimeters can be converted to inches by applying the centimeter constructor and then inverting the inches constructor. The library makes XML parsing easier by using the same syntax to parse XML as to construct it.

2 Background

2.1 Pattern matching in Haskell

In this paper, we use Haskell as a model for the typical semantics of pattern matching. Haskell is a statically typed, non-strict, purely functional programming language. Haskell pattern matching, like that of most functional languages, operates on algebraic data types. For example, we could implement a list of integers as

```
data List = Cons Int List | Nil
```

This statement declares that a list is either a `Nil` or a `Cons`, where the latter has two slots, one for an integer and one for another list. To construct a list, we can write code which treats `Cons` and `Nil` as ordinary functions, where `Cons` takes two arguments and `Nil` takes none. For example, the list `'(1 2 3)` can be constructed as `Cons 1 (Cons 2 (Cons 3 Nil))`. To get information out of the slots, pattern matching is used. For example, to get the head of the list, the `Cons` constructor can be used on the left hand size of =:

```
head (Cons x xs) = x
```

Pattern matching only succeeds if the argument to `head` is actually a `Cons`, rather than a `Nil`. This can be used to dispatch on what kind of argument is given. For example, a function to sum a list can be implemented as

```
sum (Cons x xs) =  x + (sum xs)
sum Nil = 0
```

Conceptually, when `Cons` or `Nil` are put on the left hand side of the assignment, they're used backwards. You try to undo `Cons` into the variables `x` and `xs`, and if this fails, you go on to trying to undo `Nil`. This is not how pattern matching in Haskell is typically explained, but it is equivalent to the standard definition, and this observation leads to a generalization which we use.

2.2 The concatenative programming model

Factor is a stack-based programming language. This means that parameters are passed around on an implicit stack, rather than used as explicit arguments in a function call. For example, the function + pops the top two items off the stack and pushes their sum. A literal like `3` pushes the number 3 on to the stack. So, to add 1 and 2, the code `1 2 +` can be used. After executing this, the stack contains the number 3.

In a concatenative programming language, programs are constructed by concatenating smaller programs together. In Factor, the programs are conceptually functions from stacks to stacks. For example, `1` is a function which takes one stack and creates another stack, which is the original stack with 1 added on top. So, the program `1 2 +` is a function which is the composition of the three smaller functions, that is, $+ \circ 2 \circ 1$. All together, this amounts to the reverse Polish notation popularized on HP calculators and present in Forth.

Quotations are Factor's mechanism for anonymous functions. They are expressed simply as a number of functions enclosed in square brackets. For example, `[ 1 2 + ]` is a quotation which, when called, pushes the sum of 1 and 2 on the stack. Quotations are used in a variety of contexts. For example, `if` in Factor is simply a function which takes from the stack a boolean and two quotations, and calls one of the quotations depending on the value of the boolean.

Certain operations exist to manipulate the stack. The stack is manipulated to structure the flow of data between functions. These are described in Figure 1.The operations `swap` and

Input	Output
`x dup`	`x x`
`x drop`	
`x y swap`	`y x`
`x [ code ] dip`	`code x`

Table 1: Basic stack manipulation operations in Factor

`dip` are enough to perform any permutation on a fixed-size portion of the stack from the top, and `dup` and `drop` allow a value to be used in multiple places, or to be ignored. In practice, a few more operations are used, and lexically scoped variables are available in Factor.

Functions in Factor, called *words* by Forth tradition, are defined beginning with `:`, and terminated with `;`. For example, the following code splits a comma-delimited string up into pieces, interprets each as a number, outputting the positive numbers in sorted order. For the input `"2,1,3,-2,5"`, this would output the array { `1 2 3 5` }.

```
: parse-numbers ( string -- numbers )
    "," split
    [ string>number ] map
    [ 0 >= ] filter
    natural-sort ;
```

Many functions come out written very cleanly in this "pipelined" style. The *stack effect* `( string -- numbers )` is merely a comment, documenting that the word `parse-numbers` pops a string off the top of the stack and pushes an array of numbers.

Factor is object-oriented in that all values are objects, and methods can dispatch on them. One way to define new classes is to make a *tuple*, with a number of *slots*. For example, to make a cons cell class, and a nil class, the following syntax is used.

```
TUPLE: cons car cdr ;
TUPLE: nil ;
```

The slots of the `cons` can be of any type, as Factor is dynamically typed. The following declarations define constructors for the tuples, called `<cons>` and `<nil>`.

```
C: <cons> cons
C: <nil> nil
```

The tuple class declaration implicitly defines some words which are useful. `nil?` is a predicate testing if the top of the stack is a `nil` tuple, and accessors like `car>>` extracts the value of the slot named `car`.

```
: sum ( list -- n )
```

```
    dup nil? [
        drop 0
    ] [
        dup car>> swap
        cdr>> sum +
    ] if ;
```

Factor has no built-in support for pattern matching, so this code is not nearly as elegant as the Haskell version. In this paper, we attempt to rectify the situation. Once pattern matching is defined, the implementation will be

```
: sum ( list -- n )
    {
        { [ <nil> ] [ 0 ] }
        { [ <cons> ] [ sum + ] }
    } case ;
```

This is one of the simplest examples of use of the pattern matching system (this would be quite a useless paper if our chief accomplishment were a cleaner implementation of `sum`), and it can be used for much more.

3 Pattern matching as function inverse

The central idea of this paper is to reexamine pattern matching as inverting a function. What we want to do is invert a chunk of code. Sometimes, for a given word, we can figure out its inverse directly. Here are some simple examples of words whose inverse we can determine directly.

- For example, if we had a word `increment`, which increments the top of the stack, then its inverse would be to decrement the top of the stack.
- The inverse of `swap` is `swap`, as `swap swap` is a no-op.
- The inverse of `1`, that is, pushing the number 1 on the stack, is to pop off the top of the stack, asserting that it is 1. The inverse of `1` is defined only on stacks whose top element is 1, as the image of `1` (as a function from stacks to stacks) contains only stacks with 1 on top. In other words, `1` is not surjective, so its "inverse" is a partial function.
- The inverse of `<cons>` checks that the top of the stack is a `cons` tuple, and if so, gives a new stack with the `car` and `cdr` of the stack on top.

In a concatenative programming language, words are defined simply as the composition of other words. If we know the inverse of each of those words, and the larger word is not recursive, then we can derive its inverse. The identity needed is $[\ f\ g\ ]^{-1} = [\ g^{-1}\ f^{-1}\ ]$

From this identity, we can derive the inverse of, say, consing 1 onto the head of a list. The Factor code for that would be

```
: add-1 ( list -- newlist )
    1 swap <cons> ;
```

Now, the inverse of `add-1` can be derived in the following steps.

$$\begin{gathered}
\texttt{add-1}^{-1} \\
\texttt{(1 swap <cons>)}^{-1} \\
\texttt{<cons>}^{-1}\ \texttt{swap}^{-1}\ \texttt{1}^{-1} \\
\texttt{(check-cons dup car>> swap cdr>>) (swap) (check-1)} \\
\texttt{check-cons dup car>> swap cdr>> swap check-1}
\end{gathered}$$

In the above code, `check-cons` checks that the top of the stack is a cons cell, and leaves the stack unchanged if so. If not, it throws an exception. `check-1` checks that the top of the stack is 1. (These words don't exist in the implementation; they are used for expository purposes.)

3.1 Interface

In Factor, inverting a quotation is available through the `undo` word. For example, the above code could be expressed as `[ add-1 ] undo`. An exception is thrown if the inverse fails. Using Factor's macro system, the inverse is calculated at compile time when possible, and runtime when not. If it is calculated at runtime, the result is memoized so that it does not get computed more than once.

To make this useful in pattern matching, there is also a way to chain multiple `undo` calls together. The input is a sequence of pairs of quotations. Factor executes the inverse of the first of the pair, and if that succeeds, the second of the pair is executed. If that fails, then the next pair is attempted. This construct is called `case`. A simple example of its use is below, from earlier.

```
: sum ( list -- n )
    {
        { [ <nil> ] [ 0 ] }
        { [ <cons> ] [ sum + ] }
    } case ;
```

In the first branch, corresponding to `<nil>`, nothing is left on the stack if the inverse is successful. In the second branch, corresponding to `<cons>`, the stack has the `car` and `cdr` of the list if the inverse is successful.

It would be unreasonably restrictive to require that, for any piece of code, `code [ code ] undo` is the identity function on stacks where it is defined. For example, the `>array` pops a sequence (eg string, resizable vector, etc.) off the top of the stack and pushes an array with the same contents. We might define the inverse of this as checking that the top of the stack is an array, and if so leaving the stack as is. The problem here is that `>array` is not injective: many different sequences are mapped to the same array. It would be a significant and unnecessary restriction to require that only injective things be used.

However, it should always be true that `[ code ] undo code` is the identity function where it is defined. This is a result of the basic properties of functions. Even if $f : A \to B$ is neither injective nor surjective, for any $x \in f(A), f(f^{-1}(\{x\})) = x$, where $f^{-1}(X)$ here denotes the inverse image of the function. If this property were not followed, then the "inverse" would not be an in any meaningful sense.

Only a small subset of Factor is invertible by `undo`, as conditionals and recursion are not supported.

3.2 Implementation

In this design, a quotation is inverted by recursively inverting each word in the quotation, and each word that that calls, etc, until a word with a pre-defined inverse is reached. There are four types of pre-defined inverses.

- **Literals** are handled very simply. The inverse of `1` is `1 =/fail` where `=/fail` is a word which pops the top two items from the stack, and throws an exception if they are not equal.
- **Simple inverses** are the case where one word is replaced with one fixed piece of code for its inverse. The inverses here are looked up in a hashtable associated with each word, called the word properties. All words in Factor have this hashtable. For convenience, the word `define-inverse` can be used to set this word property. For example, the following expresses that the inverse of `exp` is `log`[1]:

  ```
  \ exp [ log ] define-inverse
  ```

- **Pop inverses** are where you have to pop off a fixed number of things from the compile-time "stack", that is, they need to consume literal arguments. For example, `map` needs to receive a quotation argument as a literal, and it will invert this quotation. A pop inverse can do anything it wants with the literal arguments, producing an inverse quotation as a result. If these arguments are not supplied lexically, it will fail. The inverse to `map` is[2]

  ```
  \ map 1 [ undo-quot [ map ] curry ] define-pop-inverse
  ```

  Here, `curry` is a word which takes the top two things from the stack, a quotation and an object, and creates a new quotation with the object prepended to it. `undo-quot` calculates the inverse of a quotation, given a quotation on the stack. The 1 indicates that one literal argument is required. It is required because `map` by itself has no well-defined inverse, though `[ <nil> <cons> ] map` does have a natural inverse.

[1]The \ syntax escapes a word so that it can be pushed on the stack rather than executed.
[2]The actual definition has checks to make sure that a sequence was given.

- **Math inverses** let you give an inverse for either half of a binary function. For the word `-`, for example, we want to provide an inverse for `3 -` as well as for `3 swap -`. Both of these are provided, as long as one argument is supplied literally. Here is how the inverse for `-` is defined:

  ```
  \ - [ + ] [ - ] define-math-inverse
  ```

  The first quotation specifies that the inverse to `3 -` is `3 +`, and the second quotation specifies that the inverse to `3 swap -` is `3 swap -`. Math inverses are flexible enough to invert the following word which converts a temperature in Fahrenheit to Celsius

  ```
  : fahrenheit>celsius ( deg-f -- deg-c )
      32 - 5/9 * ;
  ```

All together, these allow the inverse of `0 3array`[3] `[ 1 + ] map` to be equivalent to `[ 1 - ] map first3`[4] `drop`, with the addition of certain checks which might fail. These checks ensure that the argument is an array of length 3, and that after the mapping, 0 is the last element of the array.

In order to allow the maximum possible subset of Factor to be inverted, all words called from code passed to `undo` is inlined down to where an explicit inverse is defined. This allows for a primitive form of interprocedural constant folding pass, which increases the possibilities for what math functions can be inverted.

The compiled code from this system is as fast as if it were written by hand. Additionally, the implementation of `undo` is linear time in the size of the the input. The only overhead for `case` is the use of the exception handling system for flow control.

4 In practice

Because these inverses are just contained in word properties, other modules besides the main pattern matching module can also define inverses for words. In the following sections, we investigate two applications of pattern matching.

4.1 A units library

A library in Factor for representing quantities with associated units has been developed in Factor. The interface is simple: there are words like `cm` which convert a number into a

[3]`3array` is a word with the stack effect `( first second third -- array )` that creates an array with 3 elements.

[4]`first3` is a word with stack effect `( array -- first second third )` which gets the first, second and third elements of an array.

quantity with the unit attached, and there are words which manipulate quantities with attached units, such as `d+`. The interesting part is how numbers are extracted. The pattern matching system can be used directly for this purpose, but explicit definitions for the inverse are not required for individual definitions of units.

So to convert centimeters to inches, the programmer can use the code `1 inches [ cm ] undo`, where `inches` creates a length-dimensioned object of the given number of integers, and `cm` does the same for centimeters. This even works for arbitrarily complex definitions of units, such as converting between miles-per-gallon and kilometers per liter.

The inverse of both `inches` and `cm` can be derived from their definitions, which are as follows:

```
TUPLE: dimensioned n numerator denominator ;
C: <dimensioned> dimensioned

: m ( n -- dimensioned )
    { m } { } <dimensioned> ;

: cm ( n -- dimensioned )
    100 / m ;

: inches ( n -- dimensioned )
    2+27/50 * cm ;
```

In reading this code, remember that the math here is postfix, not infix. Also, note that the full definition symbolically reduces dimensions when there is redundancy in the units of the numerator and denominator, but this does not affect pattern matching behavior. In the definition of `m`, the syntax { `m` } indicates an array of length 1 containing the word `m`; there is no recursion.

To deal with more complicated usage patterns, math inverses are defined on words like `d+`. This, together with the constant folding described previously, lets conversion happen using `undo` with the following definitions:

```
: km/L ( n -- km/L )
    km 1 L d/ ;

: mpg ( n -- mpg )
    miles 1 gallons d/ ;
```

Because of the way pattern matching on literals works, if you try to convert an inch into seconds, there will be a pattern matching error. So this system lets you pattern match on what type of unit a given quantity is!

It takes a total of eight lines of code in the actual system to enable all of this. Before the pattern matching library was used, every unit had to be accompanied by a hand-coded conversion function to SI base units to achieve what is now automatic.

4.2 A library for manipulating XML

Factor's XML library provides syntax for XML literals, and for interpolating objects from the stack into these literals. This syntax allows for interpolating strings or other chunks of XML into an XML template. For example,

```
"hello" <XML <main><-></main> XML> xml-pprint
```

prints the output

```
<?xml version="1.0" encoding="UTF-8"?>
<main>
  hello
</main>
```

The syntax `<->` is used to provide a slot where the item on the top of the stack can go. With the pattern matching system of this paper, we can define the inverse of this interpolation, so that the following code

```
<XML <main>hi</main> XML>
[ <XML <main><-></main> XML> ] undo
```

leaves on the stack `"hi"`. The syntax `<XML XML>` expands at parse-time into pushing an XML literal on the stack, followed by a macro call to perform interpolation. This macro has a custom inverse defined which examines an XML document, matching it with the one involved in interpolation and outputting the values which are present in missing slots. Through this technique, XML can be processed in a high-level, declarative way without support from the language itself. Below is a simple example of XML processing using pattern matching[5].

```
: write-link ( url text -- ) ... ;
: write-bold ( text -- ) ... ;
: write-italics ( text -- ) ... ;

: process-tag ( tag -- )
    {
        { [ <XML <a href=<->> <-> </a> XML> ] [ write-link ] }
        { [ <XML <b> <-> </b> XML> ] [ write-bold ] }
        { [ <XML <i> <-> </i> XML> ] [ write-italics ] }
    } case ;
```

[5]Really, we'd want something recursive here, and we'd use `[XML XML]` to indicate that we are dealing with chunks of XML rather than a whole XML document, which would use the `<XML XML>` syntax

5 Related work and future direction

Though the pattern matching system described in this paper was developed independently, there has been earlier work in creating extensible pattern matching systems. One of the first was a system called Views, where one algebraic data type can be seen as another and pattern matched that way, if an appropriate conversion function is used. This allows for pattern matching to be more high-level, using an arbitrary view rather than the one which is used in the specific implementation of the data type [8]. Our system is significantly more general than Views, as the above examples demonstrate.

A more general proposal is an approach building pattern matching from a set of combinators [6]. In this approach, arbitrary functions can have an inverse defined on them, and these inverses are used for pattern matching. In an extended version, the List monad is used for backtracking. Neither of these can be implemented in Haskell directly because its syntax is not extensible in the way that these systems demand.

The J programming language has the conjunction *power*, written `^:` [4]. This iterates a given function some number of times. If this is used with a negative argument, the function is inverted. For example, `(2&o.)^:_1` represents the inverse cosine function. This is an example of a previous programming language feature for inverting arithmetic expressions, as our system does.

XML pattern matching exists in Scala as a built-in construct, and this provided inspiration for the equivalent construct in Factor [1]. An extensible pattern matching system is available for the F# programming language [5].

Constraint logic programming operates on a related, but more general model, where computation can be taken in a number of directions based on which variables are free and which are bound. It would be interesting to see how far this model could be taken in a constraint-based direction, perhaps where f^{-1} adds a constraint (which can fail immediately) rather than operating as a partial function which must produce its answer immediately.

In the current implementation, when a pattern fails, control is passed to the next pattern using Factor's exception handling system, but several alternatives are possible. Using a different system, backtracking could be implemented, in the way that logic programming languages use. In terms of implementation, this can be simulated with `call/cc` or Haskell's List monad.

Another direction to take would be to move away from simple bijections to something along the lines of the Harmony project [3]. The goal of this project is to create a language for bidirectional manipulation of trees, so only direction has to be written explicitly. Rather than insisting that all functions be injective, as our system does, or using indeterminate logic variables, Harmony's combinators have access to the original input when going backwards. This is not useful for simple pattern matching, but it would be interesting to see how closely related the two systems are.

6 Acknowledgements

Thanks Slava Pestov, for making Factor and helping with the theory of this, and Doug Coleman for constructing the units library. Thank you Gopalan Nadathur for pointing out the connection between this form of pattern matching and constraint programming.

References

[1] EMIR, B., MANETH, S., AND ODERSKY, M. Scalable programming abstractions for XML services.

[2] PESTOV, S. Factor programing language. `http://factorcode.org`, 2003.

[3] PIERCE, B. C., SCHMITT, A., AND GREENWALD, M. B. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003. Superseded by MS-CIS-05-02.

[4] SOFTWARE, J. J user manual. `http://www.jsoftware.com/help/user/contents.htm`, 2001.

[5] SYME, D., NEVEROV, G., AND MARGETSON, J. Extensible pattern matching via a lightweight language extension. *SIGPLAN Not. 42*, 9 (2007), 29–40.

[6] TULLSEN, M. First class patterns. In *In 2nd International Workshop on Practial Aspects of Declarative Languages, volume 1753 of LNCS* (2000), Springer-Verlag, pp. 1–15.

[7] VON THUN, M. Main page for the programming language JOY. `http://www.latrobe.edu.au/philosophy/phimvt/joy.html`, 2001.

[8] WADLER, P. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1987), ACM, pp. 307–313.