Extensible Pattern Matching in an Extensible Language (Extended Abstract)

Sam Tobin-Hochstadt

Northeastern University

Abstract. Pattern matching is a widely used technique in functional languages, especially those in the ML and Haskell traditions. Although pattern matching is typically not built into languages in the Lisp tradition, it is often available via libraries built with macros. We present a sophisticated pattern matcher for Racket, which extends the language using macros, supports novel and widely-useful pattern-matching forms, and is itself extensible with macros.

1 Extending Pattern Matching

The following Racket[1] [3] program finds the magnitude of a complex number, represented in either Cartesian or polar form as a 3-element list:

```
(define (magnitude n)
  (cond [(eq? (first n) 'cart)
         (sqrt (+ (sqr (second n)) (sqr (third n))))]
        [(eq? (first n) 'polar)
         (second n)]))
```

While this program accomplishes the desired purpose, it's far from obviously correct, and commits the program to the list-based representation. Additionally, it unnecessarily repeats accesses to the list structure making up the representation. Finally, if the input is `'(cart 7)`, it produces a hard-to-decipher error.

In contrast, the same program written using pattern matching is far easier to understand:

```
(define (magnitude n)
  (match n
    [(list 'cart x y) (sqrt (+ (sqr x) (sqr y)))]
    [(list 'polar r theta) r]))
```

The new program is shorter, more perspicuous, does not repeat computation, and produces better error messages.

The function can also be easily converted to arbitrary-dimensional coordinates:

```
(define (magnitude n)
  (match n
    [(list 'cart xs ...) (sqrt (apply + (map sqr xs)))]
    [(list 'polar r theta ...) r]))
```

[1] Racket is the new name of PLT Scheme.

This definition is much improved from the original, but it still commits us to a list-based representation of coordinates. By switching to custom, user-defined pattern matching forms, this representation choice can be abstracted away:

```
(define (magnitude n)
  (match n
    [(cart xs ...)
     (sqrt (apply + (map sqr xs)))]
    [(polar r theta ...) r]))
```

Our custom pattern matching form can use other features of Racket's pattern matcher to perform arbitrary computation, allowing us to simplify the function further by transparently converting Cartesian to polar coordinates when necessary:

```
(define (magnitude n)
  (match n
    [(polar r theta ...) r]))
```

In the remainder of the paper, we describe the implementation of all of these examples, focusing on user-extensibility. We begin with a history of pattern matching in Scheme, leading up to the current implementation in Racket, touching briefly on the adaptation of standard techniques from ML-style matching [4] and their implementation via macros [2,1]. Then we describe the implementation of sequence patterns (seen above with the use of `...`) and other pattern forms not found in conventional pattern-matching systems. Third, we describe how to make patterns user-extensible by exploiting the flexibility of Racket's macro system.

References

1. Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proceedings of the Eighth Workshop on Scheme and Functional Programming*, 2007.
2. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
3. Matthew Flatt and PLT. Reference: Racket. Reference Manual PLT-TR2010-reference-v5.0, PLT Scheme Inc., June 2010. `http://racket-lang.org/techreports/`.
4. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM.