Extensible Pattern Matching in an Extensible Language

Sam Tobin-Hochstadt

PLT @ Northeastern University
samth@ccs.neu.edu

Abstract. Pattern matching is a widely used technique in functional languages, especially those in the ML and Haskell traditions, where it is at the core of the semantics. In languages in the Lisp tradition, in contrast, pattern matching it typically provided by libraries built with macros. We present `match`, a sophisticated pattern matcher for Racket, implemented as language extension. using macros. The system supports novel and widely-useful pattern-matching forms, and is itself extensible. The extensibility of `match` is implemented via a general technique for creating extensible language extensions.

1 Extending Pattern Matching

The following Racket[1] [12] program finds the magnitude of a complex number, represented in either Cartesian or polar form as a 3-element list, using the first element as a type tag:

```
(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 from the `third` function.

In contrast, the same program written using pattern matching is far simpler:

```
(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. For this reason, pattern matching has become a ubiquitous tool in functional programming, especially for languages in the Haskell and ML families. Unfortunately, pattern matching is less ubiquitous in functional languages in the

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

Lisp tradition, such as Common Lisp, Scheme, and Racket. This is unfortunate, since as we demonstrate in the remainder of the paper, not only are the same benefits available as in Haskell or ML, but the extensibility provided by languages such as Racket leads naturally to expressive and extensible pattern matchers.

1.1 More Expressive Patterns

The function can also be easily converted to arbitrary-dimensional coordinates, using the `...` notation for specifying an arbitrary-length list:

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

Racket is untyped, so we can add argument checking in the pattern match to catch errors early. Here we use the `?` pattern, which tests the value under consideration against the supplied predicate:

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

This implementation is more robust than our original function, but it approaches it in complexity, and still commits us to a list-based representation of coordinates.

1.2 Custom Patterns

By switching to custom, user-defined pattern matching forms, we can simplify our patterns and the 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]))
```

```
      |  x                 variables
pat  ::= (? expr)          predicates
      |  (and pat)         conjunction
      |  'val              literal data
      |  (cons pat pat)    pairs
      |  (list pat)        fixed-length lists
```

Figure 1: Simple Patterns

We now have an implementation which avoids commitment to representation and handles arbitrary dimensions as well as argument checking. Without extensible pattern matching, this might require interface changes or a large number of helper functions. Instead, we have developed reusable abstractions which support the clear definition of exactly what we mean.

In the remainder of the paper, we describe the implementation of all of these examples, focusing on user-extensibility. We begin with a tour of pattern matching in Racket, touching briefly on the adaptation of standard techniques from ML-style matching [18] and their implementation via macros [9,7]. 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. Finally, we discuss related work, including a history of pattern matching in Scheme and Racket.

2 Pattern Matching in Racket

In this section, we describe both the interface and implementation of the most basic pattern matching constructs in Racket. The fundamental pattern matching form is `match`, with the syntax

```
(match expr [pat expr] ...)
```

The meaning of `match` expressions is the traditional first-match semantics. The grammar of some simple patterns is given in Figure 1. In reality, `match` provides many more forms for all of the basic data types in Racket, including mutable pairs, vectors, strings, hash tables, and many others; user-defined structures are discussed in Section 3. The semantics are as usual for pattern matching: variable patterns match any value and bind the variable in the right-hand side to the value matched, literals (written here with `'`) match just themselves, `list`- and `cons`-patterns match structurally, and `and`-patterns match if both conjuncts do. The only non-standard pattern is `(? expr)`. Here, `expr` must evaluate to a one argument function, and the pattern matches if that function produces `true` (or any other non-false value) when applied to the value being considered (the *scrutinee*). For example, the pattern `(and (? even?) x)` matches only even numbers and binds them to `x`. A `match` expression matches the scrutinee against each pattern in turn, executing the right-hand side corresponding to the first successful match. If no pattern matches, a runtime error is signaled.

2.1 Compilation to Racket

The basic compilation model adopted by `match` is the backtracking automata framework introduced by Augustsson [1] and optimized by Le Fessant and Maranget [18]. However, unlike these models, `match` is implemented as a macro, and thus produces plain Racket code directly, rather than exploiting lower-level mechanisms for tag dispatch and conditional branching. Each automata state is represented by a thunk (0-argument procedure) which may be tail-called by a later computation. Conditional checks simply use `if` tests and type-testing predicates such as `pair?`.

To compile the following `match` expression:

```
(match e_1
  [(and (? number?) x) e_2]
  [_ e_3])
```

the following code is generated:

```
(let ([tmp e_1]
      [f (λ () e_3)])
  (if (number? tmp)
      (let ([x tmp]) e_2)
      (f)))
```

Here, `f` is the failure continuation; it is invoked if the `number?` test fails. Testing is a simple conditional, and variable patterns translate directly to `let` binding. In larger patterns, multiple failure continuations are generated and may be called in multiple places.

One important detail is added to the compilation process by the presence of the `?` pattern. Since `?` patterns contain an *expression* rather than a subpattern, the expression is evaluated at some point during matching. This expression should be only evaluated once, to avoid needless recomputation as well as duplicate effects, but should not be computed if it is not needed. Finally, since backtracking automata may test data multiple times, the function *produced* by the expression may be called 0, 1, or more times during matching.

2.2 Static Checking

One key design choice is already fixed by the simple patterns, in particular by the presence of `?`: disjointness of patterns and completeness of matching are now impossible to check in the general case, just as in Haskell with pattern guards. Even in the absence of `?`, the untyped nature of Racket means that checking completeness of pattern matching is only possible in the trivial case when a catch-all pattern is provided. In view of these limitations, `match` does not warn the user when it cannot prove the absence of unreachable patterns or potentially uncovered cases.

In more restricted systems, such as the `cases` form provided by some textbooks [13,16], these problems become tractable, but the presence of expressive patterns such as `?` patterns makes it impossible here. We are investigating whether Typed Racket [24] makes more static checking possible.

2.3 Implementing Syntactic Extensions

The basic structure of the `match` implementation is common to many pattern matching compilers. The distinctive aspect of the implementation of Racket's pattern matcher is the use of syntactic extension in the form of macros to implement the `match` compiler. We briefly review Racket's syntactic extension mechanisms and describe how they are used for implementing `match`.

Defining Syntactic Extensions The fundamental extension form is `define-syntax`, which binds values to names in the *syntactic* environment, i.e., the environment used at compilation time. This form is primarily used to define *macros*, which are functions from syntax to syntax. The following is a macro that always produces the constant expression `5`:

```
(define-syntax (always-5 stx) #'5)
```

`always-5` is a function with formal parameter `stx`. When the `always-5` macro is used, this function is provided the syntax of the use as an argument. Macros always produce *syntax objects*, here created with the `syntax` constructor (abbreviated here `#'`). That syntax object is, of course, just the expression `5`. Since `always-5` is bound by `define-syntax`, it is a macro, and can be used in expressions such as `(+ 3 (always-5))`. This expression is rewritten *at compile time* to the expression `(+ 3 5)`, which then evaluates as usual.

Of course, most syntactic extensions examine their arguments. Typically, these are written with pattern matching to simplify destructuring.[2] For example, the following defines a macro which takes a function and an argument, and adds a debugging printout of the argument value:

```
(define-syntax (debug-call stx)
  (syntax-parse stx
   [(debug-call fun arg)
    #'(let ([tmp arg])
        (printf "argument ~a was: ~a\n" 'arg tmp)
        (fun tmp))]))


> (debug-call add1 (+ 3 4))
argument (+ 3 4) was: 7
8
```

Here the pattern `(debug-call fun arg)` binds the variables `fun` and `arg` which are then used in the result.

Finally, macros such as `debug-call` can perform computation in addition to pattern substitution to determine the resulting expression. We use this ability to define the `match` syntactic extension.

```
(define-syntax (simple-match stx)
  (define (parse-pat pat rhs)
    (syntax-parse pat
      [x:id
       #`[true                (let ([x tmp]) #,rhs)]]
      ['val
       #`[(equal? tmp 'val)                #,rhs]]))
  (syntax-parse stx
    [(simple-match e:expr [pat rhs] ...)
     (with-syntax ([(new-clause ...)
                    (syntax-map parse-pat
                                #'(pat ...) #'(rhs ...))])
       #'(let ([tmp e])
           (cond new-clause ...
                 [else (error "match failed")])))]))
```

Figure 2: A simple pattern matcher

The Basics of `match` Implementing complex language extensions such as `match` requires more care than extensions that are expressed as simple rewrite rules. A simple pattern matcher supporting only variable and literal patterns is given in Figure 2. The basic architecture is as follows:

- The expression being matched, `e`, is bound to a new temporary variable, `tmp`.
- Each clause, consisting of a pattern `pat` and a right-hand side `rhs`, is transformed into a clause suitable for `cond` by the `parse-pat` function. For literal patterns, the test expression is an equality check against `tmp`. For variable patterns, the test is trivial, but the variable from the pattern is bound to `tmp` in the right-hand side.
- Finally, all of the clauses are placed in a single `cond` expression with an `else` clause that throws a pattern-match failure.

This matcher is missing most of the features of `match`, but demonstrates the key aspects of defining `match` as a syntactic extension. In the subsequent sections, we will see how to extend this matcher via the same syntactic extension framework in which it is implemented.

A Production Implementation The `match` implementation first translates each pattern into an abstract syntax tree of patterns while also simplifying patterns to remove redundancy. For example, `list` patterns are simplified to `cons` patterns, and patterns with implicit binding are rewritten to use `and` with variable patterns.

Given a table of pattern structures, both column optimizations [18] and row optimizations [19] are performed to select the best order to for matching patterns and to coalesce related patterns. Finally, the Augustsson algorithm is used to generate the residual code.

[2] For more on the relationship between `match` and these pattern matchers, see Section 5.

3 Advanced Patterns

Extending the simple patterns described in Section 2, we now add repetition to lists, patterns that transform their input, and matching of user-defined structures. These patterns are necessary both to support the examples presented in Section 1, as well as introducing concepts that are used in the definition of `match` expanders.

3.1 Lists with Repetition

The first significant extension is the ability to describe arbitrary-length lists. Of course, we can already describe some uses of such lists using the `cons` pattern:

```
(match (list 1 2 3)
 [(cons (and x (? number?)) xs) x])
```

However, the use of `...` allows the specification of arbitrary-length lists with a specification of a pattern to be matched against each element. For example, to match a list of numbers:

```
(match (list 1 2 3)
 [(list (and xs (? number?)) ...) xs])
```

This checks that every element of the list is a number, and binds `xs` to a *list* of numbers (or fails to match). The `...` suffix functions similarly to the Kleene closure operator in regular expressions. In fact, in conjunction with `or` and `and` patterns `match` patterns can express many regular languages.

Compilation Compiling patterns with `...` requires a straightforward extension to the traditional pattern matching compilation algorithm. A pattern clause such as `[(list p ...) expr]` is compiled using two clauses. The first is `[(list) expr]`, matching empty list and the second is `[(cons p rest) (loop rest)]` where `loop` is a recursive function that repeats the matching again on its argument. Of course, all of the variables bound by `p` must be passed along in the loop so that they are accumulated as a list.

3.2 `app` Patterns

The simplest form of extensible pattern matching is the `app` pattern. A pattern of the form `(app f pat)` applies the (user-specified) function `f` to the value being matched, and then matches `pat` against the *result* of `f`. For example:

```
> (match 16
    [(app sqrt (and (? integer?) x))
     (format "perfect square: ~a" x)]
    [_ "not perfect"])
"perfect square: 4"
```

Implementing the app pattern is straightforward. We simply apply the function to the value being matched, and continue pattern matching with the new result and the sub-pattern. This fits straightforwardly into the Augustsson-style matching algorithm. Currently, we do not attempt to coalesce multiple uses of the same expression, however this would be a straightforward addition. As a result of the backtracking algorithm, the function may be called multiple times; the use of side-effects in this function is therefore discouraged.

app patterns are already very expressive—Peyton Jones [20] gives many examples of their use under the name *view patterns*. Below is one simple demonstration of their use:

```
(define (map f l)
  (match l
    ['() '()]
    [(cons (app f x) (app (curry map f) xs))
     (cons x xs)]))

> (map add1 (list 1 2 3 4 5))
'(2 3 4 5 6)
```

In combination with the ability to define new pattern forms, app patterns allow almost arbitrary extensions to patterns.

3.3 User-defined Structures

Racket supports the definition of new user-specified structures:

```
(struct point (x y))
(define p1 (make-point 1 2))


> (point-x p1)
1
```

Of course, they should also be supported in pattern matching:

```
> (match p1
   [(point a b) a])
1
```

To accomplish this, the `struct` form, which is also implemented as a language extension, must communicate with `match` *at expansion time*, so that `match` can produce code using the `point?` predicate and `point-x` selector.

Static Binding The simplest form of communication between two portions of the program is binding. To take advantage of this at expansion time, we can bind arbitrary values with the `define-syntax` form, not just macros.

```
(define-syntax just-5 5)
```

Now, `just-5` is bound to `5` in the static environment. We can access this environment with the `syntax-local-value` function, rewriting the `always-5` macro thus:

```
(define-syntax (always-5 stx)
  (to-syntax (syntax-local-value #'just-5)))


> (always-5)
reference to undefined identifier: to-syntax
```

`syntax-local-value` produces the value to which an identifier such as `#'just-5` is bound. We then convert that value to a piece of syntax with `to-syntax`.

Static Structure Information We can take advantage of this static binding mechanism to record statically known information about structure definitions. The `(struct point (x y))` form expands to definitions of the runtime values:

```
(define point-x —) (define point-y —) (define make-point —)
```

as well as *compile-time* static information about the structure.

```
(define-syntax point
  (list #'make-point #'point? #'point-x #'point-y))
```

Thus, the identifier `point` contains information about how to create `point`s, test for them, and extract their components. Using this facility, we can now extend our simple pattern matcher from Figure 2 to test for structures. We add a new clause to the `parse-pat` function:

```
[(struct sname)
  (let* ([static-info (syntax-local-value #'sname)]
         [predicate (second static-info)])
   #`[(#,predicate tmp)                    #,rhs])]
```

In this, we first access the static structure information about the mentioned structure (`static-info`), and select the element naming the structure predicate (`predicate`). Then the result clause simply uses the predicate to test `tmp`.

In Racket, of course, the `struct` pattern also takes patterns to match against each field of the structure. Compiling these patterns uses the field accessors names provided by the static information.

4 Extensible Patterns

Using the techniques of Section 3.3, we now add extensibility to `match`.

4.1 `match` Expanders

The first task is to define the API for extending `match`. The fundamental mechanism is a procedure that consumes the syntax for a pattern using the extension and produces syntax for a new pattern. For example, the following procedure always produces a pattern matching numbers:

```
(λ (stx) #'(? number?))
```

Of course, more interesting functions are possible. This example combines the `and` and `?` pattern to make them more useful together than the ones presented in our original grammar in Figure 1:[3]

```
(λ (stx)
  (syntax-parse stx
    [(?? pred pat) #'(and (? pred) pat)]))
```

To inform `match` that this is to be used as an extension, we introduce the `define-match-expander` form. This form expects a procedure for transforming patterns, and binds it statically. We could simply use `define-syntax`, but this has several drawbacks: (a) it prevents us from distinguishing `match` expanders from other forms and (b) it prevents us from adding additional information to `match` expanders. For example, the Racket `match` implementation supports an old-style syntax for backwards-compatibility, and the distinction between `match` expanders and other static binding enables the definition of expanders that work with both syntaxes.

We can therefore define our trivial `match` expander for numbers with

```
(define-match-expander num
  (λ (stx) #'(? number?)))
```

which is equivalent to

```
(define-syntax num
  (make-match-expander (λ (stx) #'(? number?))))
```

This simply wraps the given procedure in a structure that `match` will later recognize. We can now use the `match` expander in a pattern as follows:

```
> (match 7
    [(num) 'yes]
    [_ 'no])
'yes
```

[3] Racket's version of `match` includes this functionality in the `?` pattern, as demonstrated in the first section.

4.2 Parsing `match` expanders

Extending the `parse-pat` function of Figure 2 to handle `match` expanders is surprisingly straightforward. We add one new clause:

```
[(expander . rest)
 ; (1) look up the match expander
 (let ([val (syntax-local-value #'expander)])
   (if (match-expander? val)
       (let (; (2) extract the transformer function
             [transformer (match-expander-transformer val)]
             ; (3) transform the original pattern
             [new-pat (transformer #'(expander . rest))])
         ; (4) recur on the new pattern
         (parse-pat new-pat rhs))
       (error "not an expander")))]
```

There are four parts to this new clause. In step 1, we look up the value bound to the name `expander`. If this is a `match-expander` value, then we extract the transformer function in step 2, which we defined to be a function from the syntax of a pattern to syntax for a new pattern. In step 3, we apply this transformer to the original pattern, producing a new pattern. Finally, in step 4, we recur with the `parse-pat` function, since we now have a new pattern to be handled.

This loop executes the same steps as the basic loop of macro expansion as presented by Dybvig et al. [9, Figure 4] and by Culpepper and Felleisen [6, Figure 2]. This is unsurprising, since we have extended the syntax of an embedded language, that of `match` patterns, with transformer functions, just as macros extend the syntax of the expression language with transformers. Based on this insight, we can use the same strategies to ensure *hygiene*, a key correctness criterion for macro systems, that are applied in existing systems. Racket provides an API to these facilities, but their use is beyond the scope of this paper.

These 8 lines of code are the key to extensible pattern matching. They provide a simple facility for defining arbitrary syntactic transformations on patterns. In combination with expressive patterns such as `app`, `and`, and `?`, new pattern abstractions and new uses of pattern matching can be created.

4.3 Using `match` expanders

In combination with the `app` pattern, we can now define the `polar` example from the introduction. The implementation is given in Figure 3. This expander makes use of the `or` pattern to handle either polar or Cartesian coordinates, and the `app` pattern to transform the Cartesian coordinates into polar ones.

`match` expanders have been put to numerous other uses in Racket. For example, they are used to create a domain-specific matching language for specifying rewrite rules in the PLT web server [17]. They are used to provide abstract accessors to complicated data structures in numerous systems. A library for defining Wadler-style "views" using `match` expanders is available for Racket as well.

```
(define-match-expander polar
  (λ (stx)
    (syntax-parse stx
      [(_ r pats ...)
       #'(or (list 'polar r (? real? pats) ...)
             (cons 'cart
                   (app (λ (x)
                          (cons (sqrt (apply + (map sqr x)))
                                ... compute angles ...))
                        (list r (? real? pats) ...))))])))
```

Figure 3: A match expander

4.4 Further Extensions

Based on this framework, we can add additional features to `match` expanders, making them more useful in real-world applications. Above, we discuss the ability to specify a separate transformer for legacy pattern-matching syntax, allowing the same `match` expander to be used with both syntaxes. Additionally, since Racket supports creating structures which can be used as procedures[12, Section 3.17], `define-match-expander` supports creating `match` expanders which can be also used in expressions, allowing the same name to be both a constructor and a pattern matching form.

5 History and Related Work

5.1 A History of Pattern Matching in Scheme

To the best of the author's knowledge, the first pattern matcher in Scheme was written by Matthias Felleisen in February or March of 1984, inspired by the pattern matching features of Prolog [4]. This original system was written in, or ported to, Kohlbecker et al's `extend-syntax` macro system [14], and went by the name of `match`, which was also the name of the primary macro. The system was then maintained by Bruce Duba until 1991, when Andrew Wright began maintaining it. At this point, it was ported to a number of other macro systems, including Common Lisp-style `defmacro` [22], Clinger and Rees's explicit renaming system [2], and Dybvig et al's expander-passing style [8]. In 1995, Wright and Duba prepared an unpublished manuscript [26][4] describing the features of `match`, including the grammar of patterns. This matcher generated decision trees rather than automata.

At this point, the development of the `match` library, usually known as the "Wright matcher", stagnated. Wright's implementation was ported to `syntax-case` [9] by Bruce Hauman in Racket, but this implementation was not used by any other Scheme implementation. Hauman also added a number of features not found in other versions of

[4] Although this is often cited as a Rice University technical report, it was never published as such.

Wright's library. Hauman's implementation was then maintained by the author, and extended with `match` expanders as described in Section 4. The author subsequently created a new implementation in 2008 using backtracking automata, which is currently distributed with Racket.

In 1990, Quinnec [21] presented a pattern matcher for S-expressions as well as a formal semantics for his matcher. However, we know of no Scheme implementation which distributes his implementation.

Eisenberg, Clinger and Hartheimer also included a pattern matcher in their introductory book, *Programming in MacScheme* [10]. This matcher was implemented as a procedure rather than a macro, functioned on quoted lists as patterns, and did not bind variables.

Of course, Scheme macro systems, beginning with Kohlbecker and Wand's *Macro by Example* [15] have included sophisticated pattern matchers, functioning entirely on S-expressions. These systems introduced the `...` notation for sequence patterns, later adopted by Wright's matcher among others and seen in the second example in the introduction. However, these have typically not been integrated into the rest of the programming language and have not made use of sophisticated pattern compilation techniques. An exception is Culpepper et al.'s `syntax-parse` system [5], which uses a novel compilation strategy to support sophisticated error reporting.

Finally, numerous Scheme and Lisp systems have implemented their own simple pattern matching libraries, too many to list here.

5.2 Other Extensible Pattern Matchers

Numerous other proposals for extensible pattern matching in functional languages have been presented. Here, we survey only the most significant.

The original proposal for extensible pattern matching is Wadler's "views" [25], presented in the context of Haskell. In this system, a view is a pair of an injection and projection from an abstract type to an algebraic datatype. Views are intended to support data abstraction in conjunction with pattern matching.

Expressing views using `match` expanders is straightforward, as is seen with the example of `cart` and `polar`, which form a simple view on tagged lists. More complex examples are also expressible, including the use of views as injections, which uses the extensions discussed in Section 4.4. Cobbe [3] provides a library which implements view definitions as a layer on top of `match` expanders.

Also in Haskell, Peyton Jones presents view patterns [20] as an extension to the GHC compiler and gives a wide range of motivating examples. These view patterns are almost identical to `app` patterns in `match`, with the exception that Peyton Jones suggests using typeclasses to implicitly provide the function when it is not supplied. However, this extension is not implemented in GHC. Peyton Jones also lists 5 desirable properties of pattern matching extensions, all of which are provided by `match` with `app` patterns and `match` expanders.

Active patterns, originally proposed by Erwig [11] and subsequently extended and implemented in F# by Syme et al. [23], as well as Scala extractors [**?**], provide more expressive extensions. Users can define both partial and total patterns. Partial patterns can be implemented as an abstraction over `app` and `?` patterns, using a helper function.

Total patterns require the definition of several `match` expanders, each using such an abstraction, with only one helper function. Using both pattern abstraction and abstraction over definitions provided by macros, such extensions can be specified in Racket just as in F#, but again without the need to modify the base language. Since `match` does not check exhaustiveness of pattern matching, total active patterns cannot be verified to match completely.

6 Conclusion

Pattern matching is an invaluable tool for programmers to write concise, maintainable, and performant code. However, it does not usually support the abstraction facilities that functional programmers expect in other parts of the language. In this paper, we describe a syntactic abstraction facility that allows arbitrary rewriting of patterns at compilation time. Furthermore, this is provided in a pattern matching system that is implemented as a syntactic extension in Racket. The implementation reveals a striking similarity between the base language extension mechanism and extensions defined in higher-level domain-specific languages such as pattern matching.

Acknowledgments

Ryan Culpepper provided invaluable assistance in the development of `match`, and devised the algorithm used for handling sequence patterns. Matthias Felleisen shared his knowledge of the early history of pattern matching in Scheme. Stevie Strickland and Vincent St-Amour provided feedback on earlier drafts of the paper. The author is supported by a grant from the Mozilla Foundation.

References

1. Lennart Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag.
2. William Clinger. Hygienic macros through explicit renaming. *LISP Pointers*, 4(4), 1991.
3. Richard Cobbe. Views, 2007. `http://planet.racket-lang.org/`.
4. Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367, New York, NY, USA, 1996. ACM.
5. Ryan Culepper and Matthias Felleisen. Fortifying macros. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.
6. Ryan Culpepper and Matthias Felleisen. Debugging hygienic macros. *Sci. Comput. Program.*, 75(7):496–515, 2010.
7. Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proc. 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701*, pages 1–13, 2007.
8. R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, January 1988.
9. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

10. Michael Eisenberg, William Clinger, and Anne Hartheimer. *Programming in MacScheme*. MIT Press, Cambridge, MA, USA, 1990.
11. Martin Erwig. Active patterns. In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 21–40, London, UK, 1997. Springer-Verlag.
12. Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.
13. Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 2008.
14. Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LFP '86: Proc. 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.
15. Eugene E. Kohlbecker and Mitchell Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.
16. Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Lulu.com, January 2007.
17. Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher Order Symbol. Comput.*, 20(4):431–460, 2007.
18. 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.
19. Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM Workshop on ML*, pages 35–46, 2008.
20. Simon Peyton Jones. View patterns: lightweight views for Haskell, 2007. `http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`.
21. Christian Queinnec. Compilation of non-linear, second order patterns on S-expressions. In *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 340–357, London, UK, 1990. Springer-Verlag.
22. Guy Lewis Steele Jr. *Common Lisp—The Language*. Digital Press, Bedford, MA, 1984.
23. Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 29–40, New York, NY, USA, 2007. ACM.
24. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. 35th Symposium on Principles of Programming Languages*, pages 395–406. ACM Press, 2008.
25. P. Wadler. 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*, pages 307–313, New York, NY, USA, 1987. ACM.
26. Andrew Wright and Bruce Duba. Pattern matching for Scheme, 1995.