6.945 Final Project: Match Definitions and Cases in MIT Scheme

Tom Boning, Eli Davis, and Jenny Ramseyer
{tboning,ebdavis,ramseyer}@mit.edu

Project Github: https://github.com/orinatic/Match-Define

Overview

Our project aims to implement case matching and match definitions in MIT Scheme. Currently we support all of these in the default Scheme interpreter.

Matching

Match definitions are definitions (and let, named-let, let*, and letrec) of the form (match-define '(1 2 3) '((? a) (? b) (? c))) which set a = 1, b = 2, and c = 3. The general form of these matches is (match-FOO object pattern . body) where object is the object the match should be run against. The matcher itself can be redefined by the user. We chose to use the matcher from Problem Set 4.

First we rewrote our matcher code from Problem Set 4, as we needed a cleaner implementation. See matcher.scm for our new implementation. We then attempted to write macros for define, let, let*, letrec, and named-let which worked to varying extents. We then moved on to doing the actual assignment.

First Macro Strategy

Before we got a working version, we had several failed implementations. Our first failed implementation used macros. These macros mostly work: Define can define things at the correct scope level, depending where it is in a function, and the let's are all scoped correctly. We have define, let, letrec, named-let, and let*. The basic form for all of them was something like:

```
(define-syntax match-let
  (sc-macro-transformer
   (lambda (exp env)
     (let* ((body (cddr exp))
            (dict (*run-match* (car vals) (cadr vals)))
       `(let ( ,@(map (lambda(entry)
                        `(,(car entry) ,(cadr entry)))
                      dict))
          (begin ,@(map (lambda (statement)
                          statement)
```

```
                    body)))))))
```

(The full code for these macros is in the “previous-attempts/previous-attempts.scm”).

These macros work fine if they are given something like

```
(match-define '((? a) (? b)) '((4 2)))
```

Unfortunately, there was a key bug in these macro definitions: we were running the match within the macro. And macros don't have access to the current environment, so we don't catch other existing bindings in the environment. So the following code wouldn’t work:

```
(define d 4)
(match-define '((? a) 4) '((1 d)))
```

because the matcher would see the symbol d, not the value 4. Another, more dire consequence is that we can’t do expressions like the following:

```
(define p '((? a) (? b)))
(match-define p '((4 2)))
```

or

```
(define o '((4 2)))
(match-define `((? a) (? b)) o)
```

Instead of substituting in the value of p that p is defined to be, this attempt at match-define would use the symbol "p". Specifically, with this implementation, case matching is impossible - case matching requires that the thing being matched against to be passed in. We scrapped this version as unworkable.

Eval Strategy

After we figured out that macros wouldn't work, we switched to an eval/environment strategy. Our first idea was to try to edit the environment directly, with something like the following:

```
(define-syntax dict-let
  (sc-macro-transformer
   (lambda (exp env)
     (let ((dict (cadr exp))
           (body (caddr exp)))
       `(lambda ()
          (define (empty a) 'nothing)
          (let ((our-env (procedure-environment empty)))
            (let dict-define-loop ((todo ,dict))
              (if (null? todo)
                  'done
                  (let ((var (caar todo))
                        (val (cadar todo)))
                    (environment-define our-env var val)
                    (dict-define-loop (cdr todo)))))
            (eval ,body our-env)))))))
```

Unfortunately, we had issues with environment scoping. We originally tried using the top level environment, but any assignments to that were permanent. In addition, attempting to assign and then unassign within our function body might break if we use continuations or if exceptions are thrown. The version above creates an empty environment, which prevents bindings from hopping into the global scope. However, as all its bindings only apply inside the scope of the function, we don't have access to global bindings. Thus, when we try to eval something, our code breaks (see the code in previous-attempts/previous-attempts.scm).

We also tried to create our desired string and then eval it, as below:

```
(define (dict-let dict body)
  (let ((keys (dict->keys dict))
        (vals (dict->vals dict)))
    (eval
      `((lambda (,(car keys))
          ,(car keys))
        (car vals))) ??? ...
```

(this code will break for other reasons as well, but it's a small case that shows the idea)

This generates the (mostly) correct string, but we have no way to evaluate it. We run into the same problem as above. Eval takes an environment, and we found no way to get the current environment. As far as we can tell, (the-environment) only works at top-level; it can't be used in the middle of a let statement. Unfortunately, Scheme doesn't allow the user to get the current environment outside of the top level. (While inconvenient to the user, this step allows the compiler to have better guarantees and thus better optimize the code).

Identifiers and Interpreters

In the process of writing our previous version, we learned that Scheme's assignment macros require all of the identifiers present in the code to be statically named, hence our problems with macros. There are two ways to fulfill this requirement. One way is to write a sub interpreter where we can partially evaluate the code in order to get the full values of the pattern and the object. We can then evaluate them to get the identifiers. Another way is to place restrictions on what constitutes an "acceptable" pattern, such that all patterns are self contained. Imposing the second requirement would mean that each identifier is present in the macro, letting us access them to correctly set their assignments.

We tried both approaches, and decided that requiring a complete pattern in each match statement is the more useful approach. We wrote both a sub interpreter and a macro based system for general Scheme, to see which one would work best. While a sub-interpreter would suffice, sub-interpreters are less extensible than the original REPL, so we chose to use the initial REPL in order to allow for more compatibility with other systems, and for future additions to our work.

Sub Interpreter

By using a sub interpreter, we gained a lot of power and control over the what code we executed. This effectively and easily solved our issues with environments and partial evaluation. Sub interpreters easily allow access to the environment as a matter of course, as it is needed to bind symbols, which is normally handled by the compiler. Compilers do not allow outside access to this environment, as it is much more difficult for compilers to optimize while allowing such fine grain access.

In addition to allowing fine grained access to the environment, sub interpreters allow partial evaluation. What this allows is code to be evaluated to the dictionary of values, effectively allowing the symbols and values to be gained from code by the interpreter and properly bound. This is related to access to the environment. Since there is fine grained access, it is easy to peek during evaluation and see the current state of environments and values. This even allows us to have the pattern we match against be a symbol or evaluated code, such as this:

```
(let ((vars '(seq (? x) (?? xs)))
      (vals '(seq 1 2 3 4 5)))
 (match-let vars vals (pp (cons x xs)) (pp 'herewego) (pp x) (pp
xs)))
;(1 2 3 4 5)
;herewego
;1
;(2 3 4 5)
;#!unspecific
```

Of course, this comes with an associated performance cost. Sub interpreters do little optimization and effectively have another layer of code evaluation, which decreases speed of code execution. Furthermore, sub interpreters have a critical problem: they are quite difficult to extend, and require reimplementing existing features. For instance, we ran into issues with the sub interpreter being unable to interpret common scheme features, such as variable arity functions, without reimplementing the feature. For these reasons, we were unsatisfied with a sub interpreter based solution. We instead redoubled our efforts outside a sub interpreter, using macros and functions.

Current System

The final system we decided on was another macro-based one. Our first macro-based system only worked if the object and the pattern were in the text of the procedure call. Thus,

```
(match-let `((? x) (? y)) `(1 2)
      (+ x y))
```

would work, but

```
(define pattern `((? x) (? y)))
(match-let pattern `(1 2)
      (+ x y))
```

or

```
(define object `(1 2))
(match-let `((? x) (? y)) object
      (+ x y))
```

would not.

Originally, we had declared such limitations completely unworkable, (which they were) and set about trying to allow both of the above constructions. As mentioned above, however, Scheme's assignment macros require all of the identifiers present in the code to be statically named. The first solution we thought of would be to write a sub-interpreter. But, halfway through writing our sub-interpreter, we looked at some Scala and Python pattern-matching code and came to an interesting realization--while the analogous

```
(define object '(1 2))
(match-let '((? x) (? y)) object
      (+ x y))
```

would work in both of those languages,

```
(define pattern '((? x) (? y)))
(match-let pattern '(1 2)
      (+ x y))
```

would not. The languages we were trying to emulate, like Scheme, require that the variables we intend to bind are present in the binding statement. From that realization, we realized that we could write new macros and move back to the primary REPL. Instead of trying to run the match in the macro, as our old macros did, our new macros figure out what variables are going to be match-assigned, wrap the body in a lambda to assign those variables, and spit out a procedure which will run the match. These macros can be found in match-let-final.scm and match-case-final.scm in the GitHub repository. They can be nested correctly, with each other and with normal assignment statements, and, like Python and Scala, the object can be passed in.

It was only after giving our presentation that Professor Sussman pointed out that he had already implemented something similar to this in the last section of Problem Set 4. In retrospect, we should have reread through all of the Problem Set 4 code before we started working, but, alas, we did not. So, we ended up doing a lot of re-inventing the wheel. However, during our re-inventing, we learned a lot about Scheme's internals, and still managed to add new extensions to Scheme.

Extensibility

As always, there are still some features that we would like to add to our system. One feature which generic let supports which our match-let does not is multiple assignments per let statement. Generic let will allow the user to do something of the form

```
(let ((a 1)
      (b 2))
     body)
```

However, our match-let only allows the user to have a single assignment per match-let. Thus, in order to match and assign multiple statements, we need to nest calls to match-let, as follows:

```
(match-let pattern1 query1
      (match-let pattern2 query2
            body))
```

Nesting calls to match-let is easy for the user to do, but a bit inconvenient. For a cleaner system, we would like to implement a new version of match-let which would allow the user to do multiple assignments per match-let. (One of our earlier failed versions allowed multiple assignments, so it is possible, but we have not implemented it yet).

Another feature which would be nice to add would be to allow match-let to take assign statements of the form found in generic let as well as our special matcher assignments. Thus, we could do something of the form:

```
(match-let pattern query ;;setting the query variables to be
the result of the match
      (a 1) ;;setting a to be 1
      body)
```

The exact syntax on how we would make that work has not been established. Again, this extension is not strictly necessary, as the users can achieve the same effect by nesting calls to match-let and let, but having an integrated match-let would give a more convenient and easy-to-use API.

Another feature that would continue this system would be to hook into Scheme's exception system through the unbound variable exception. This would allow us to have a dynamic dictionary that we could check variables against if they were unbound, effectively extending the environment. This addition would not be as extensible with other unbound variable handlers, as it has the possibility to trigger fallthrough issues and other subtle bugs like improper shadowing of variables.

Since Scheme is a highly-extensible language, we would like to run some more checks to make sure that our code meshes well with other extensions to MIT Scheme. Since our code runs in the base REPL, it should be quite easy to add to other Scheme extensions. It can even be used with other matcher implementations. The only restriction that our system puts on the other matcher systems is that the matcher must output an alist of results. To switch matcher systems, the user will also have to redefine our "find-variables" function (found in "match-let-final.scm") in order for our system to find the variables defined in the match results. As our system does not overwrite the base definition of let, match-let and match-case can be used in combination with let and other functions, without causing problems. Thus, our code should work well with other extensions to Scheme.

Matcher Syntax

(Similar to Problem Set 4)

<literal-value> will match that literal value

ex: 2 will match 2

(? x) will match a single value or a function
ex: (? x) will match `(1), but not `(1 2)
ex: (? x) will match `(cos)
ex: `((? x) (? y)) will match `((1 2)), with x = 1 and y = 2

(?? x) will match an arbitrary number of values or expressions
ex: (?? x) will match `(1 2 3 4) or `((cos 4) (sin z))
ex: (?? x) will match an empty list
ex: (j (?? x) k (?? y) j (?? x) z) will match
(j b b b k c c c j b b b z) or (j b b b k b b b j b b b z) or (j k j z) (or many other things)

(match-query match-query match-query) works and will match a list of these match-queries, allowing us to chain queries together.
ex: ((? x) (?:choice 2 (? y)) (? z)) will match `(1 2 3)

(?:choice pattern pattern ...) will match any of the patterns passed into choice
ex: (?:choice x y) will match `(x) or will match `(y)
ex: (?:choice x (? y)) will match `(z) with y = z or will match `((k)) with y = `(k)

(? var ,pred?) will match var to something that satisfies the predicate
ex: (? x ,string?) will match x to anything that is a string
ex: (? y ,symbol?) will match y to anything that is a symbol
Note that we cannot use ,pred? with ??

(?:ref query-name) allows us to make a reference to some other query which has already been defined. It is only used in ?:pletrec (see below)

(?:pletrec ((query-name query)
(other-query-name other-query)
(...))) allows us to recursively define queries in terms of each other.
ex:
(?:pletrec ((simple-case (?:choice * (1 (?:ref simple-case)))))
(?:ref simple-case)) will match '((1 *)) '(), or a variety of other things.

API

special-form **match-let** *pattern query body*

query is a query which follows 6.945-matcher-syntax, described earlier in this document.

pattern is a list which query will be matched against.

query will be matched against *pattern*. If there is a match, the each matched variable will have its matched value bound to it, and the expressions will be evaluated, returning the last expression. If there is no match, the entire thing will return #f
ex: (match-let '(1 2) '((? a) (? b)) (+ a b)) sets a = 1, b = 2, and returns 3.

We allow multi-line body statements in the match-let body.
ex:

```
(let ((vals '(seq 1 2 3 4 5)))
 (match-let vals '(seq (? x) (?? xs))
       (pp `(,x and ,xs have been set))
       (cons x xs)))
```

will print "(1 and (2 3 4 5) have been set)" and then return (1 2 3 4 5)

In the present system, we only allow one match statement per match-let (unlike in generic let). If one wants to have multiple match statements, simply nest calls to match-let.
ex:

```
(match-let '(1 2 3 4) '((? a) (? b) (? c) 4)
           (match-let '(5 6) '((? a) (? d))
                      (+ a b c d)))
```

will return 16.

For more extensive test cases, please see "match-let-final.scm".

special-form **match-case** *query (pattern body) (pattern body)...*

match-case takes a query and a series of cases to match that query against. Each of these cases has a body which, if the case is matched, will execute. This body can be a multi-line or single-line statement.
ex:

```
(match-case '(harold 4 5 3333 33 3333)
         (`(harold (? a) (? b ,number?) (?? c)) (pp `(harold is ,a ,b ,c)))
         (`(cora (? a ,number?) (? b) (?? c)) (pp "cora"))
```

```
        (((?? c) chesterworth (? d)) (pp "chesterworth")))
will return (harold is 4 5 (3333 33 3333))
```

For more extensive test cases, please see “match-case-final.scm”.