C. Generalized Pattern Matcher

This appendix gives a definition of dmatch that is more general in several ways than Oleg Kiselyov’s dmatch, which appeared in Byrd and Friedman (2007). It improves error reporting, since now it is possible to associate a name with each appearance of dmatch, as in the use of example in h below. We get more generality by not handling quote specially, which allows for certain common patterns to be specified that were previously not possible. Finally, there is no else clause and the order of the clauses is arbitrary, but only one pattern (plus guard) can succeed for each invocation of dmatch. Here is an example of dmatch using guards.

\[
(\text{define} \ h

(\text{lambda} \ (x \ y)

(\text{dmatch} \ h \ (x \ y) \ \text{example}

\((, a, b)\)

\((\text{guard} \ (\text{number? a}) \ (\text{number? b}))\)

\((+ a b)\)

\((, (a, b), c)\)

\((\text{guard} \ (\text{number? a}) \ (\text{number? b}) \ (\text{number? c}))\)

\((+ a b c)\))\))\))

\((\text{list} \ (h \ 1 \ 2) \ \text{(apply} \ h \ '(1 \ 3 \ 4))) \Rightarrow (3 \ 8)\)
\]

In this example, a dotted pair is matched against two different kinds of patterns. In the first pattern, the value of x is lexically bound to a and the value of y is lexically bound to b. Before the pattern match succeeds, however, an optional guard (no side-effects allowed in guards) is run within the scope of a and b. The guard succeeds only if x and y are numbers; if so, then the sum of x and y is returned.

The second pattern matches against a pair (a three-element list), provided that the optional guard succeeds. The value of x is 1 and the value of y is (3 4). Then a matches against 1, b matches against 3, and c matches against 4. They are all numbers, so both calls to h succeed.

The overall syntax of dmatch looks like this:

\[
\begin{align*}
\text{match} & := (\text{dmatch} \ \text{exp} \ \text{clause} \ \ldots) \\
& \quad | (\text{dmatch} \ \text{exp} \ \text{name} \ \text{clause} \ \ldots) \\
\text{clause} & := (\text{pattern} \ \text{guard} \ \text{exp} \ \ldots) \\
\text{guard} & := (\text{guard} \ \text{boolean-exp} \ \ldots) | \varepsilon \\
\text{pattern} & := \varepsilon | \text{var} \\
& \quad | (\text{pattern}_1 \ \text{pattern}_2 \ \ldots) \\
& \quad | (\text{pattern}_1 \ . \ \text{pattern}_2)
\end{align*}
\]

Now we examine the implementation of dmatch. The main dmatch macro simply handles the optional name that we can provide, and passes off control to the auxiliary helpers which do most of the extra work. Our auxiliary macros will give us a package list which is then processed by the run-a-thunk procedure.

\[
(\text{define-syntax} \ \text{dmatch} \ (\text{syntax-rules} ()

(\text{(let} \ ((\text{v} \ (\text{e} \ \ldots) \ \ldots))

(\text{dmatch-remexp} \ v \ (\text{e} \ \ldots) \ \ldots)))

(\text{(run-a-thunk} \ v \ v \ \text{f} \ \text{PKG})))

(\text{(let} \ ((\text{v} \ \text{name} \ (\text{e} \ \ldots) \ \ldots))

(\text{dmatch-remexp} \ v \ (\text{e} \ \ldots) \ \ldots)))

(\text{(run-a-thunk} \ v \ v \ \text{name} \ \text{PKG})))\))
\]

In our case we want to represent a package comprising the clause and a thunk. We use the following for our package abstraction.

\[
\begin{align*}
\text{(define} \ \text{pk} & := (\text{lambda} \ (\text{cls} \ \text{thk}) \ (\text{cons} \ \text{cls} \ \text{thk}))) \\
\text{(define} \ \text{pk-clause} & := (\text{lambda} \ (\text{pkg}) \ (\text{car} \ \text{pkg}))) \\
\text{(define} \ \text{pk-thunk} & := (\text{lambda} \ (\text{pkg}) \ (\text{cdr} \ \text{pkg})))
\end{align*}
\]

The first step in processing a dmatch syntax is to ensure that we only evaluate the input expression once, which is what the dmatch-remexp ensures.

\[
\begin{align*}
\text{(define-syntax} \ \text{dmatch-remexp} \ (\text{syntax-rules} ()

(\text{(let} \ ((\text{rator} \ \text{rand} \ \ldots) \ \text{cls} \ \ldots))

(\text{let} \ ((\text{v} \ (\text{rator} \ \text{rand} \ \ldots)))

(\text{dmatch-aux} \ v \ \text{cls} \ \ldots))\))

(\text{(let} \ ((\text{v} \ \text{cls} \ \ldots) \ (\text{dmatch-aux} \ v \ \text{cls} \ \ldots)))\))
\end{align*}
\]

At each expansion of the dmatch-aux macro, we want to create a package list of some type. We have three cases: two recursive cases and a single base case. If we have a pattern without a guard and the pattern matches, we want to add its clause along with its thunk to the package list. In the case where we have a guard, we want to conditionally add the clause and thunk to the package list only if the guard also succeeds.

\[
\begin{align*}
\text{(define-syntax} \ \text{dmatch-aux} \ (\text{syntax-rules} \ (\text{guard})

(\text{(let} \ ((\text{pk} \ \text{v}) \ (\text{dmatch-aux} \ v \ \text{cs} \ \ldots)))

(\text{p} \ \text{pat} \ \text{v} \ \text{pat}

(\text{(if} \ (\text{not} \ (\text{and} \ g \ \ldots))

(\text{f} k)

(\text{cons} \ \text{PKG} \ \text{(pat} \ (\text{guard} \ g \ \ldots) \ e_0 \ e \ \ldots))

(\text{lambda} \ (e_0 \ e \ \ldots))

(\text{f} k)))\))

(\text{(let} \ ((\text{f} k)))

(\text{(let} \ ((\text{f} k)))\))

(\text{(let} \ ((\text{f} k)))\))

(\text{(let} \ ((\text{f} k)))\))\))
\)
\]

To do the heavy lifting, we abstract the actual pattern matching into another helper macro ppat that does the check on the pattern and then expands into one of two forms. The consequent expression is the result of the expansion of ppat if the pattern matches, and the alternate expression otherwise. In all cases, the alternate is just another dmatch-aux macro that drops the first pattern and continues the recursive expansion. To encode the alternative, we build a thunk, which avoids expanding the same expression multiple times.

Now we consider how matching occurs using ppat, and leverage the syntax-rules pattern matcher to do most of the work. We need to do a bit of tree recursion on our expansion in the pair case to match the car and cdr cases. Since we may have vectors or other data we want to handle, we use equal? instead of eq?.

\[
\begin{align*}
\text{(define-syntax} \ \text{ppat} \ (\text{syntax-rules} \ (\text{unquote})

(\text{(let} \ ((\text{v} \ \text{unquote} \ \text{var} \ \text{kt} \ \text{ kf})) \ (\text{let} \ ((\text{var} \ v) \ \text{kt}))

(\text{(let} \ ((\text{v} \ (\text{car} \ v)) \ (\text{vy} \ (\text{cdr} \ v)\))))
\end{align*}
\]
If there is no match, the error is reported using `no-matching-pattern`. If there is an overlap between two or more patterns/guards, then we report this error using `overlapping-patterns/guards`. Otherwise, if there is no overlap, then we invoke the thunk in the singleton package list.

```scheme
(define run-a-thunk
  (lambda ((v-expr v name pkg ⇤))
    (cond
      ((null? pkg ⇤) (no-matching-pattern name v-expr v))
      ((null? (cdr pkg ⇤)) ((pkg-thunk (car pkg ⇤))))
      (else (ambiguous-pattern/guard name v-expr v pkg ⇤))))

(define no-matching-pattern
  (lambda ((name v-expr v))
    (if name
      (printf "dmatch ˜d failed˜n˜d ˜d˜n" name v-expr v)
      (printf "dmatch failed˜n˜d ˜d˜n" v-expr v))
    (error 'dmatch "match failed")))

(define overlapping-patterns/guards
  (lambda ((name v-expr v pkg ⇤))
    (if name
      (printf "dmatch ˜d overlapping matching clauses˜n" name)
      (printf "dmatch overlapping matching clauses"ˆn")
      (printf "with ˜d evaluating to ˜d˜n ˜d˜n" v-expr v)
      (for-each pretty-print (map pkg-clause pkg ⇤)))))
```

Here is the definition of \(	exttt{h}\) (without the second clause) after macro expansion.

```scheme
(lambda (x y)
  (let ((pkgs
        (let ((v (cons x y)))
          (let ((fk (lambda () . . .)))
            (if (pair? v)
                (let ((vx (car v)) (vy (cdr v)))
                  (let ((a vx))
                    (let ((b vy))
                      (if (not (if (number? a) (number? b) #f))
                          (fk)
                          (cons (pkg
                                   (guard (number? a))
                                           (number? b))
                                 (+ a b)))
                          (lambda () (+ a b)))
                     (fk)))))
        (run-a-thunk '((x . y) (cons x y) 'example pkgs))))
```

There are two kinds of improvements that should be resolved by the compiler. First, \(vx\) and \(vy\) are not needed, so they should not get bindings. The lexical variable \(a\) and \(b\) could have replaced \(vx\) and \(vy\), respectively. Second, \(a\) and \(b\) should be parallel \texttt{let} bindings.