Pattern matching with constraints

Matching pattern under constraints: Pattern matcher takes frame containing some (or no) constraints and produces an output frame with possibly more constraints, or fails.

Frame to pattern matcher containing:
Input constraint on variables: $x = a$

$(? x ?y ?y ?x) \text{ match } (a \ b \ b \ a)$

Frame from pattern matcher contains what
Output constraints on variables?

Frame to pattern matcher containing:
Input constraint on variables: $y = a$

$(? x ?y ?y ?x) \text{ match } (a \ b \ b \ a)$

Frame from pattern matcher contains what
Output constraints on variables?

Pattern Matcher

```scheme
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat) (cdr dat)
                        (pattern-match (car pat) (car dat)
                                      frame))))
        (else 'failed)))
```

```scheme
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
      (pattern-match (binding-value binding) dat frame)
      (extend var dat frame)))))
```

; ADT for binding: variable . value
```scheme
(define (binding-variable binding)
  (car binding))
(define (binding-value binding)
  (cdr binding))
```

; Frame is list of bindings
```scheme
(define (binding-in-frame variable frame)
  (assoc variable frame))
(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

Pattern Matching

Somewhere in evaluator there is a pattern matcher.

```scheme
(a ?x c)
```

```scheme
(job ?x (computer ?y))
```

```scheme
(job ?x (computer . ?y))
```

```scheme
(a ?x ?x)
```

```scheme
(?x ?y ?x ?y)
```

```scheme
(a ?x)
```
Abstractions

Any query takes a stream of database elements and a stream of frames and produces a stream of frames.

To process query

(boss (bitdiddle ben) computer)

1. Create frame(s) from matching conclusion of rule
2. Use them as stream of frames to query which is rule body

| ?z = (bitdiddle ben) | ===> | (and (job ?x (?d . ?y)) | ===> frame(s) |
| ?d = computer      |       | (supervisor (?x ?z))) |

database

(eval and apply

Query language:

To apply a rule
Evaluate the rule body relative to the environment formed by unifying the rule conclusion with the given query.

Scheme:

To apply a procedure
Evaluate the procedure body relative to the environment formed by binding the procedure parameters to the arguments.

High level design

Compound queries use same structure with fancier contents in the query box:

More compound queries (combinations)
Logic programming: useful for rule based systems. Include in your set of tools.

Remember logic programming is not exactly logic: need to understand how language works, not just how logic works.

Implementation: still uses eval / apply mutual recursion to support using abstractions.

Unification / pattern matching: know them.

Match portions of term with variables in both directions:

\[
\text{unify } (?x \ ?x) \\
\text{with } ((a \ ?y c) (a \ b \ ?z))
\]

?x = ...
?y = ...
?z = ...

How about

\[
\text{unify } (?x \ ?x) \\
\text{with } ((?y a \ ?w) (b \ ?v \ ?z))
\]

?y = ...
?v = ...
?w = ...
?x = ...

created frame contains a variable!

Unification algorithm

\[
\begin{align*}
\text{(define (unify-match p1 p2 frame)} \\
&\quad (\text{cond ((eq? frame 'failed) 'failed)} \\
&\quad ((\text{equal? p1 p2) frame)} \\
&\quad (\text{((var? p1) (extend-if-possible p1 p2 frame)})) \\
&\quad (\text{((var? p2) (extend-if-possible p2 p1 frame)) ; \{\} \\
&\quad (\text{(and (pair? p1) (pair? p2)) \\
&\quad (\text{(unify-match (cdr p1) \\
&\quad (cdr p2) \\
&\quad (\text{(unify-match (car p1) \\
&\quad (car p2) \\
&\quad frame))}})) \\
&\quad (\text{else 'failed)})))) \\
\end{align*}
\]

\[
\begin{align*}
\text{(define (extend-if-possible var val frame)} \\
&\quad (\text{let ((binding (binding-in-frame var frame))}) \\
&\quad (\text{cond (binding \\
&\quad (\text{(unify-match \\
&\quad (binding-value binding) val frame)})) \\
&\quad (\text{(var? val) ; \{\em ; ***}}) \\
&\quad (\text{(let ((binding (binding-in-frame val frame))}) \\
&\quad (\text{(if binding \\
&\quad (\text{(unify-match \\
&\quad var (binding-value binding) frame) \\
&\quad (extend var val frame)}))))) \\
&\quad ((\text{depends-on? val var frame) ; \{\em ; ***} \\
&\quad 'failed) \\
&\quad (\text{else (extend var val frame)}))})
\end{align*}
\]