Symbolic Pattern Matching in Clojure

S.C.Lynch
Teesside University
Middlesbrough
UK, TS1 3BA
(+44) 1642 218121
s.c.lynch@tees.ac.uk

ABSTRACT
This paper presents a symbolic pattern matcher developed for Clojure. The matcher provides new types of function definition, new conditional forms and new iterative structures. We argue that pattern matching and unification differ in significant ways that give them different semantics, both useful, and show that matcher capability is enhanced by allowing patterns to be dynamically created or embedded in data structures like rules and state-changing operators. We evaluate the matcher by experimentation, demonstrating that it can be used to simplify the specification of inference mechanisms as well as other types of code.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *control structures, patterns.*

General Terms
Design, Experimentation, Languages.

Keywords
Clojure, Pattern Matching, Rules, Inference.

1. BACKGROUND
We can consider four different types of matching:

- regular expression matching
- structural matching
- matching against types
- symbolic pattern matching

Regular expression matching operates at the level of strings and characters within strings and is a facility provided by most modern programming languages. Structural matching, offered by many languages (including Prolog, Haskell and Clojure), allow variables to be bound to data based on structural correspondence. In Clojure for example the variables X, Y and Z would be bound to 1, 2 and 3 respectively by the *let* expression:

```
(let [ [[X Y] Z] [[1 2] 3] ] …)
```

Structural matching is a feature of many of the more recent functional languages. Matching against types is also offered by some languages (Scala for example) where matching specifies type information and only succeeds if types correspond.

Symbolic pattern matching operates at the level of symbols and the (nested) structures within which they are contained. A distinction we make here is that *symbolic* matching permits literal values to be specified in patterns as well as variables. This allows patterns to specialise on data according to both its shape and its contents. The following patterns, for example, each bind variables X and Y but match against tuples describing different relations:

```
(on X Y)         ;; X is on top of Y
(next-to X Y)    ;; X is next to Y
(holds X Y)      ;; X is holding Y
```

Despite its long history, few programming languages provide symbolic pattern matching. In the early years of Artificial Intelligence, many systems, built in various dialects of Lisp, often used some form of symbolic matching. Typically this was built on an ad hoc "as needed" basis and, eventhough matchers would often deliver similar features and symbolic matching provided the core functionality for some systems, no standard emerged.

Examples of early systems based on symbolic pattern matching include SIR and STUDENT, perhaps culminating with Eliza and SHRDLU [1]; SIR used a small set of simple patterns to extract numeric and equality relations from simple English statements, STUDENT used patterns specifying conditional matching to process algebraic problems described textually (e.g. "If Joe has 4 times as many oranges as Mary…. How many oranges does Joe have"). Eliza (1966) engaged in dialogue, behaving as a non-directive psychotherapist but, while it produced some interesting conversation, did no semantic analysis. SHRDLU (1971) used patterns to parse English statements identifying commands to move blocks around in a simple world. Its capabilities exceeded those of many systems at that time but it targeted a very small micro-world of discourse and problem solving. Post Eliza and SHRDLU, the A.I. community considered that achieving more sophisticated results would not be accomplished by systems based exclusively on pattern matching and subsequently discussion of symbolic pattern matching, either as a basis for A.I. or as a topic in its own right, largely disappeared from academic literature.

Many modern languages provide regular expression matching and there is an increasing trend to provide capabilities associated with destructuring and variable assignment. Scheme and Racket provide macro-extensible matching but, outside the Lisp world, symbolic pattern matching is less common. POP-11 (a little dated now) is a notable exception, providing rich matching capabilities including match-iterators as well as destructuring [5] and more recently Scala has provided matching capability that facilitates some symbol matching [7].

Despite the lack of standardised *Symbolic* pattern matchers in modern programming languages they are often implicitly present in some software systems (e.g. expert systems and PDDL planning systems [4, 9]). We argue that a well featured symbolic pattern matcher provides many opportunities to simplify code; that appropriate matcher facilities allow inference engines and other systems to be constructed more concisely and with elegant code. We demonstrate this by example in the evaluation section of this

paper. The following sections outline different approaches to matching and present the key capabilities of the pattern matcher we have developed for Clojure.

2. INTRODUCTION

Broadly we aim to provide matching capabilities to simplify program code, to allow the definition of new types of functions which specialise on the structure of their arguments and can repeatedly apply patterns over larger data sets. We choose Clojure for this due to the nature of our applications (A.I. inference tools) which benefit from its semantics and its ability to integrate with Java. Clojure provides regular expression matching and some structural matching, there are Clojure libraries which provide tailored matching capabilities for specialised application areas (core.logic and core.unify [2]) and a partially specified matcher offering conditionals and function definition (matchure [10]). The matcher presented here is in part motivated and informed by these and other works but intentionally takes a different approach thereby offering alternative facilities. These, we suggest, can form the basis of a generalised symbolic matcher for Clojure.

2.1 Matching vs. Unification

While the difference between regular expression matching and symbolic matching is clear (regular expressions match at the level of strings and characters, symbolic matchers operate at the level of symbols and structures) the difference between matching and unification is more nuanced. Online forums suggest the difference between pattern matching and unification is only that unification is necessary/occurs if variables are allowed on both sides of a match expression. Here we accept there is some progression from simple pattern matching through to full unification but we consider the term "unification" to imply logical substitution. Specifically that a successful outcome of unification may leave variables unresolved in the sense that one or more variables may have more than one possible value or even an infinite set of possible values. For a wider discussion of unification see [3, 8].

We consider any process which simply associates one variable with one value to be matching. If variables are only permitted on only one side of a matching expression we term this "uni-directional" if variables are allowed on both sides we consider this "bi-directional".

We also consider examples where matching is uni-directional but still requires some level of unification (so unification does not fundamentally require variables on both sides of an expression). One example occurs when there are multiple patterns, with shared variables, which all need to be consistently satisfied.

A key aspect of any matching/unification algorithm is its policy for binding matched variables. The rest of this section explores some of the options. We assume that a function **f** takes two arguments and performs some matching or unification process on those arguments. **f** handles variables (denoted using a "?" prefix) and produces a mapping of variables to values if it succeeds.

An example of **f** using uni-directional matching:

```
f( [a ?x c], [a b c] ) → {?x ↦ b}
```

Bi-directional matching:

```
f( [a ?x c], [a b ?y] ) → {?x ↦ b, ?y → c}
```

The semantics of uni-directional matching are clear even when variables are bound more than once:

```
f( [a ?x ?x], [a b b] ) → {?x ↦ b}
f( [a ?x ?x], [a b c] ) → fail
```

However the semantics of bi-directional matching can become closer to some form of unification:

```
f( [a ?x c], [a ?y ?y] ) → {?x ↦ c, ?y ↦ c}
```

In this example there are two partial mappings (i) {?x ↦ ?y, ?y ↦ ?x} (ii) {?y ↦ c} which could unify in the following ways depending on the matching/unification algorithm:

(i)
```
{?x ↦ ?y, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ ?y, ?y ↦ c} → {?x ↦ c, ?y ↦ c}
```

or:

(ii)
```
{?x ↦ ?y, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ c, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ c, ?y ↦ c}
```

Further considerations are necessary where values cannot be fully resolved during a single application of a pattern, this can occur for different reasons. For example, if variables can bind to 1 or more values (implied by the use of "??"):

```
f( [a ??x], [??x a] ) → {?x ↦ a...a}
```

Other variables may be unresolved or undefined:

```
f( [a ?x c], [a [?p ?q] c] )
→ {?x ↦ [?p ?q], ?p ↦ #\, ?q ↦ #\}
```

where #\ represents an undefined binding.

We could allow undefined bindings to propagate through to later expressions which would either successfully become unified or result in failure. The expectation in this case is that a fully developed (logic based) unification mechanism would be employed which would handle backtracking as necessary. This approach has its uses but can also present some limitations.

Consider a rule application mechanism which accepts rules of the form:

```
[Rule 5 (hairy ?x) => (mammal ?x)]
```

The rule application uses patterns in two stages (i) to deconstruct a rule and (ii) to work with its antecedent-consequent parts. In the first stage matching could be specified as follows:

```
f( [Rule ?id ?antecedent => ?consequent],
   [Rule 5 (hairy ?x) => (mammal ?x)] )

→ {?id ↦ 5, ?antecedent ↦ (hairy ?x),
   ?consequent ↦ (mammal ?x), ?x ↦ #\}
```

We then expect the ?x variable to be bound as part of the second stage. It is reasonable to expect that the rule application mechanism and the rules themselves are developed by different people and it is obvious practice to avoid any coupling between these specifications. However, when using the approach above, a small change in one of the patterns can have unwanted results because the same variable, ?x, is used on both sides:

```
f( [Rule ?x ?antecedent => ?consequent],
   [Rule 5 (hairy ?x) => (mammal ?x)] )

→ {?x ↦ 5, ?antecedent ↦ (hairy 5),
   ?consequent ↦ (mammal 5)}
```

This results in incorrect variable bindings for the second (rule application) phase. The situation could be avoided by requiring additional syntax for matching expressions, but adding syntactic notations has an impact on the usability of representations which is better avoided. In addition, while some of the matching forms (described below) use literally specified patterns and data, others allow patterns/data to be dynamically produced, e.g.

```
f( make-pattern1(),make-pattern2() )
```

The matcher could insist that all dynamically created patterns use some kind of name generator (gensym) for any dynamically created patterns but this approach has other drawbacks (it is harder to prime patterns with variables which are intended for sharing across pattern generators for example). Both cases above can be dealt with appropriately using a uni-directional approach (i.e.: assuming match variables are only used on one side of a match expression).

Even with uni-directional matching there may be a requirement for some level of unification, notably when multiple patterns, with shared variables, are to be consistently applied across data sets – a scenario which may also generate multiple possible matches. Consider matching a set of patterns $\{p_0, p_1 \ldots p_n\}$ over data $\{d_0, d_1 \ldots d_m\}$ where all variables need consistent values, for example:

```
f( { [?x ?y], [?y ?z] },
   { [a b], [q r], [c d], [m n], [p q] }

→ {?x ↦ a, ?y ↦ b, ?z ↦ c},
  {?x ↦ p, ?y ↦ q, ?z ↦ r}
```

In this case there are two valid mappings of matcher variables. While the function **f** may simply return these mappings we consider two other behaviors which may be preferred from a more fully developed matcher:

- to use any one of the matches found;
- to use each of the matches – either as arguments to some specified function or in some block of code which is called repeatedly for each valid match.

2.2 Design Principles

With the considerations outlined above, we aim to develop a matcher that provides the following:

- clean, unambiguous semantics which allow integration and nesting of different matcher forms (macros and functions) while consistently preserving their semantics and furthermore allows matcher forms to integrate and nest with other Clojure forms without disrupting the semantics of either;
- high-level matcher forms which abstract out the details of the matching processes themselves;
- pattern forms (unencumbered by unnecessary syntax) which allow matching to occur over nested lists, vectors and maps;
- a suitable mix of forms which take literally specified patterns (patterns specified in program text) and others which allow patterns to be dynamically specified (so patterns may be read, constructed or extracted from data at run-time);
- the ability to specify pattern groups with shared variables which implicitly require some type of unification (and by implication may require backtracking – see mfind* and mfor*;
- a namespace which (i) will extend (shadow) into lexically nested matcher forms and unwind out of these forms and (ii) may be captured in a data structure – to allow the state of successful matching to be saved and reinstated or provided (e.g. as an argument) to other functions and subsystems.

A key distinction between this matcher and those acknowledged earlier is that other matchers tend to operate only with static patterns and use normal (native) variable bindings. While this can provide some opportunity to improve performance it restricts the ability to store patterns as data or dynamically create them. The matcher presented below allows dynamic construction of patterns, an approach which significantly effects the matcher utility (as we demonstrate in the evaluation section).

3. MATCHER MACROS AND FUNCTIONS

This section describes key functions and macros developed to provide a symbolic pattern matcher for Clojure which addresses the issues discussed above. The matcher takes symbolic data structures and matches them against structured patterns. Patterns operate at the level of symbols and structures; they may contain literals and match variables. Match variables are prefixed with a "?" (or "??" – see later), symbols without a "?" prefix are literals so the pattern (?x ?y end) will match with any three element structure which contains 'end as its third element (binding match variables x and y to the first and second elements of the data). The pattern ((a b) {n ?x m ?y}) matches a nested structure binding the variables x and y to values for n and m held in a two-element map, nested within the data.

The most primitive form of matcher expression provided for general use is *mlet* (matcher-let), it is structured as follows:

```
(mlet [ pattern datum ] ...body... )
```

mlet operates as follows: if the pattern matches the datum, binding (zero or more) matcher variables as part of the matching process then *mlet* evaluates its body in the context of these bindings. If the pattern and datum do not match, *mlet* returns nil.

In the following example, the pattern (?x ?y ?z) matches the datum (cat dog bat) binding match variables "x", "y", "z" to 'cat, 'dog, 'bat respectively. The expression (? y) in the body of *mlet* retrieves the value of the match variable "y" from (pseudo) matcher name space.

```
(mlet ['(?x ?y ?z) '(cat dog bat)]
  (? y))
→ dog
```

mout (matcher-out) is a convenience form to build structured output from a mixture of literals and bound match variables:

```
(mlet ['(?x ?y ?z) '(cat dog bat)]
    (mout '(a ?x (a ?y) and a ?z)))
→ (a cat (a dog) and a bat)
```

mlet returns nil if matches fail:

```
(mlet ['(?x ?y ?z) '(cat dog bat frog)]
    (mout '(a ?x a ?y and a ?z)))
→ nil
```

Unbound matcher variables also have nil values as does the anonymous match variable "?_" which will always match with a piece of data but does not retain the data it matches against:

```
(mlet ['(?_ ?x) '(cat dog)]
    (list (? _) (? x) (? y)))
→ (nil dog nil)
```

Matcher variables are immutable so, once bound, a match variable cannot be implicitly re-bound and whilst the pattern (?x dog ?x) matches (cat dog cat) it will not match (cat dog bat) because this would result in an inconsistent/ambiguous binding for "?x". This approach also holds true with nested matcher forms, so given the data (dog bat) the following expression will return (cat dog bat) but with data (rat bat) it will return 'inner-match-failed:

```
(defn foo [data]
  (mlet ['(?x ?y) '(cat dog)]
    (or (mlet ['(?y ?z) data]
          (mout '(?x ?y ?z)))
      'inner-match-failed)
    ))

(foo '(dog bat)) → (cat dog bat)
(foo '(rat bat)) → inner-match-failed
```

In addition to *single element match directives* (prefixed with "?") the matcher supports *multiple match directives* which match against zero or more elements of data (these are prefixed with "??"). Multiple directives may also be used in matcher-out expressions, in which case their value is appended into the resulting structure:

```
(mlet ['(??pre x ??post)
       '(mango melon x apple pear berry)]
  (mout '(pre= ?pre post= ??post)))

→ (pre= (mango melon)
   post= apple pear berry)
```

All patterns may be structured, containing sequences, subsequences (and maps within sequences within maps within sequences, etc.), so it is possible to use patterns to extract data from nested data structures. The pattern used in the following example extracts the value from a *quantification* slot nested within an *actor* slot (which is also nested in the enclosing data structure)…

```
(mlet ['(??_ (actor ??_ [quant ?q] ??_) ??_)
       semantics ]
  (? q))
```

mlet has its uses but other forms (constructed on top of *mlet*) provide greater functionality. These other forms can be grouped into three families (i) switching and specialisation (ii) searching and selection (iii) iteration and collection.

3.1 Switching and Specialisation

mcond is the most general of the switching/specialisation forms, it can be used to specify a series of pattern based rules as follows:

```
(mcond [exp]
    ((?x plus ?y)  (+ (? x) (? y)))
    ((?x minus ?y) (- (? x) (? y)))
    )
```

The *mcond* form will attempt to match the data it is given (the value of *exp* in the example above) to the first pattern in its sequence of rules (?x plus ?y) then its second (?x minus ?y) until it finds a rule which matches; it then evaluates the body of that rule and returns the result. As with other matcher forms, *mcond* returns nil if it fails to find a match. The *mcond* form above will return 9 if exp has a value of (5 plus 4) or 1 if exp has a value of (5 minus 4). Note that *mcond* (and other forms) can optionally use additional symbols to make their rule-based structure more explicit, we recommend using ":=>" for example:

```
(mcond [exp]
    ((?x plus ?y)  :=> (+ (? x) (? y)))
    ((?x minus ?y) :=> (- (? x) (? y)))
    )
```

defmatch is similar in structure to *mcond*, wrapping an implicit *mcond* form with a function definition:

```
(defmatch math1 []
  ((?x plus ?y)  :=> (+ (? x) (? y)))
  ((?x minus ?y) :=> (- (? x) (? y)))
  )

(math1 '(4 plus 5))  → 9
(math1 '(4 minus 5)) → -1
(math1 '(4 times 5)) → nil
```

defmatch forms can take explicit arguments in addition to their implicit matched-data argument. The example below illustrates this and additionally uses an anonymous match variable to handle default cases:

```
(defmatch math2 [x]
  ((add ?y)  :=> (+ x (? y)))
  ((subt ?y) :=> (- x (? y)))
  ( ?_       :=> x)
  )

(math2 '(add 7) 12) → 19
(math2 '(subt 7) 12) → 5
(math2 '(times 7) 12) → 12
```

Due to the way patterns may be specified at the symbol level, *defmatch* forms can be used to specialise on keywords and thereby resemble some kind of dispatch, e.g.

```
(defmatch calcd [x y]
  (:add  :=> (+ x y))
  (:subt :=> (- x y))
  (:mult :=> (* x y))
  )

(calcd :add 5 4)  → 9
(calcd :mult 5 4) → 20
```

3.2 Searching and Selection

The searching and selection mechanisms apply patterns across collections of data, returning the first match found. These matcher forms are called *mfind* (which matches one pattern across a collection of data) and *mfind** (which consistently matches a

group of patterns across a collection of data). This is illustrated using the following data:

```
(def food
 '([isa cherry  fruit]    [isa cabbage veg]
   [isa chilli  veg]      [isa apple   fruit]
   [isa radish veg]       [isa leek    veg]
   [color leek  green]    [color chilli  red]
   [color apple green]    [color cherry  red]
   [color cabbage green]  [color radish red]
  ))
```

Note that in this example we use vectors in our data, this is perhaps idiomatic but we sometimes prefer wrapping tuples as vectors (rather than as lists) and the matcher deals with either vectors or lists (or maps).

mfind takes one pattern, *mfind** takes multiple patterns:

```
(mfind ['[isa ?f veg] food] (? f))
→ cabbage

 (mfind* ['([isa ?f veg] [color ?f red])
          food]
    (? f))
→ chilli
```

3.3 Iteration and Collection

The matcher supports two forms to provide iteration and collection, these are called *mfor* and *mfor**. They iterate over sets of data using one pattern (*mfor*) or multiple patterns (*mfor**). The following examples use the food data presented above:

```
(mfor ['[isa ?f veg] food] (? f))
→ (cabbage chilli radish leek)

(mfor* ['([isa ?f veg] [color ?f red]) food]
    (? f))
→ (chilli radish)
```

3.4 Matcher Name Space

A *pseudo* matcher name space is maintained. This is not a Clojure name space but is a map (called *mvars*) which associates named matcher variables with their values. *mvars* is a lexically bound Clojure symbol accessible within the body of all matcher expressions.

with-mvars provides a simple way to inject variables into the matcher name space or shadow existing values, for example:

```
(with-mvars {'a (+ 2 3), 'b (- 3 4)}
  (println mvars)
  (with-mvars {'b 'bb, 'd 'xx, 'e 'yy}
    (println "  " mvars)
    (mlet ['(?a ?b ?d ?c) '(5 bb xx spam)]
      (println "    " mvars))
    (println "  " mvars))
  (println mvars))
```

output:

```
{b -1, a 5}
   {e yy, d xx, b bb, a 5}
     {c spam, :pat (?a ?b ?d ?c),
        :it (5 bb xx spam),
        e yy, d xx, b bb, a 5}
   {e yy, d xx, b bb, a 5}
{b -1, a 5}
nil
```

Note that the matcher adds the last datum that was match (called :it) and the last pattern :it matched against into the name space.

While direct reference to *mvars* is generally unnecessary, it is useful for writing new macros and it allows the results of successful matching operations to be saved for later processing or passed to other functions (in cases where the lexical scoping of matcher variables is found restrictive).

3.5 Implementation Notes

There are few basic building blocks to the matcher. The first is the core matches function which, in the context of any existing matcher variable bindings, performs the essential pattern matching process and builds a map of bindings for new matcher variables, e.g.

```
(matches '(a ?x ?y) '(a b c))
→ {y c, x b, :pat (a ?x ?y), :it (a b c)}

(with-mvars {'p 'ppp, 'q 'qqq}
    (matches '(a ?x ?y) '(a b c)))
→ {y c, x b, :pat (a ?x ?y),
     :it (a b c), p ppp, q qqq}
```

Other building blocks (*with-mvars* and *mlet*) use "*let*" forms to set up new lexical closures to shadow matcher name space when matching is successful which, in effect, provides lexical scope for matcher variables.

mcond (a macro) is specified as a series of *mlet* expressions and *defmatch* (also a macro) is specified in terms of *mcond*. *mfor*, *mfor**, *mfind* and *mfind** are also all specified as macros. *mfor* uses a function which recurses through an *mlet* form and *mfor** is specified in terms of *mfor*. *mfind* (like *mfor*) uses its own function to recurse through its own *mlet* form and *mfind** recurses through *mlet*.

In this way the expansion of nested matcher forms (defined as macros) produces a cascade of nested let forms where the pseudo matcher name space (*mvars*) is populated with match variables created by successful matches.

4. EVALUATION

We have evaluated the matcher from three different perspectives:

(i) an objective examination to assess whether matcher functions and macros operate as intended; whether they are semantically consistent when nested/interleaved with other matcher expressions and Clojure forms;

(ii) from the subjective view of Clojure programmers are the semantics of matcher forms appropriate and do their names (*mfind*, *mfor*, etc.) mnemonically suggest their semantics?

(iii) is the matcher useful? Does it simplify the construction and readability of Clojure code? Specifically (since this is the nature of much of our work) we are interested in simplifying the construction of inference engines – typically based on the application of rules and state changing operations.

The first approach to evaluation (above) is not described here, the matcher was constructed using a strict *test driven development* approach. The matcher presented here satisfies all tests.

User acceptance evaluation has been conducted using the matcher as a basis for student assessments and projects and also as a build tool for developing larger subsystems. Feedback from these user groups has influenced (i) the choice of names for matcher forms (macros and functions), (ii) the syntactic conventions used to specify macros and patterns and (iii) cases where the matcher semantics needed to be more clearly specified. Detailed analysis of this process is not discussed here, instead we focus on the third style of evaluation which considers the utility of the matcher as a tool for code construction.

4.1 Searching Sets of Tuples

For the first example we consider searching for objects in a set of tuples which describe the state of a micro-world. To put this in context: we receive object descriptions (and other forms) from language processing subsystem so, for example, the noun-phrase "red fruit" would produce:

```
(obj
  (quantifier all)
  (desc ((color red) (isa fruit))))
```

The phrase "a large red fruit" would produce:

```
(obj
  (quantifier any)
  (desc
    ((size large) (color red) (isa fruit))))
```

We store state information in the following form:

```
(def food
  '#{[isa chilli veg]     [isa cherry fruit]
     [isa radish veg]     [isa apple fruit]
     [isa leek veg]       [isa kiwi fruit]
     [color chilli red]   [color cherry red]
     [color radish red]   [color apple green]
     [color leek green]   [color kiwi green]
     [on chilli table]    [on cherry table]
     [on leek table]
     })
```

Our aim is to write code which, using the type of object descriptions from the language processing subsystem, can retrieve the relevant object names. Given the matcher facilities described in preceding sections we can use *mfor* to find the names of objects for a single type of fact/tuple. For example, the following form returns the names of all cubes:

```
(mfor ['(isa ?obj veg) food]
    (? obj))
→ (chilli leek radish)
```

It is possible to dynamically construct a suitable pattern for an mfor expression from the type of [relation value] pairs provided by the language processing subsystem. A match function provides a convenient way to extract the components of a [relation value] pair which can then be used in the mfor expression:

```
(defmatch find-all [tuples]
  ([?reln ?val]
    (mfor ['(?reln ?obj ?val) tuples]
      (? obj)
      )))

(find-all '(isa veg) food)
→ (chilli leek radish)
```

If the results of multiple find-all expressions are converted to sets multiple (relation value) pairs can be handled using set operators. So to find red vegetable from the food data:

```
(find-all '(isa veg) food)
→ (chilli leek radish)

(find-all '(color red) food)
→ (chilli radish cherry)

(intersection
    (set '(chilli leek radish))
    (set '(chilli radish cherry)))
→ #{radish chilli}
```

This processing can be captured in a function as follows:

```
(defn query
  [reduction pairs tuples]
  (reduce reduction
    (map #(set (find-all % tuples)) pairs))
  )

(query intersection
    '((isa veg)(color red)) food)
→ #{radish chilli}
```

The query function may also be used with union to return "or" combinations:

```
(query union
    '((isa veg)(color red)) food)
→ #{cherry radish chilli leek}
```

To satisfy our initial aim we therefore need the following:

```
(defmatch find-all [tuples]
  ([?reln ?val]
    (mfor ['(?reln ?obj ?val) tuples]
      (? obj)
      )))

(defn query
  [reduction pairs tuples]
  (reduce reduction
    (map #(set (find-all % tuples)) pairs))
  )
```

4.2 Application of Rules

The second example considers a rule-based, fact deduction or forward chaining mechanism. Facts are held as tuples and rules have antecedents and consequents. Some introductory texts for Artificial Intelligence provide example rules like:

```
IF (has fido hair) THEN (isa fido mammal)
```

While these serve to illustrate their discussion of rule-based inference, rules like this are of limited use because they are specific to object names ("fido" in this case) and take only a single antecedent and consequent. For practical purposes we need to extend this rule syntax – to allow rules to be flexible about the length of their antecedents/consequents and the objects they describe. Specifying rules in terms of match variables and writing a flexible rule application mechanism addresses this. For example:

```
(rule 15 (parent ?a ?b) (parent ?b ?c)
      => (grandparent ?a ?c))
```

can match against tuples like:

```
(def family
 '((parent Sarah Tom) (parent Steve Joe)
   (parent Sally Sam) (parent Ellen Sarah)
   (parent Emma  Bill)(parent Rob   Sally)))
```

A suitable rule application mechanism needs to split the rule into its constituent parts, search for all consistent sets of antecedents, ripple any antecedent variable bindings through to consequents and collect evaluated consequents for each rule every time it fires. In practice these requirements can be satisfied by using a match function to pull a rule apart, *mfor** to satisfy all possible antecedent combinations and *mout* to bind variables into consequents. This can be specified as follows:

```
(defmatch apply-rule [facts]
  ((rule ?n ??antecedents => ??consequents)
    :=> (mfor* [(? antecedents) facts]
          (mout (? consequents)))))

(apply-rule
  '(rule 15 (parent ?a ?b) (parent ?b ?c)
            => (grandparent ?a ?c))
  family)

→ ((grandparent Ellen Tom)
   (grandparent Rob Sam))
```

Notice that while the pattern for *defmatch* is literally specified, the patterns for *mfor** and *mout* must, necessarily, be generated dynamically. Furthermore these dynamically generated patterns are embedded in the rule structure pulled apart by *defmatch*'s literal pattern.

To investigate this rule deduction example further we use a richer set of facts and rules where the consequences of some rules trigger the antecedents of others (we choose a "toy" example to illustrate this).

```
(def facts
  '((mineral pebble)  (small pebble)
    (mineral boulder) (large boulder)
    (small daisy)     (light daisy)
    (on boulder daisy)
     ))

(def rules1
  '((rule 0
      (dangerous ?x)(fragile ?y)(on ?x ?y)
          => (broken ?y))
     (rule 1 (heavy ?x) => (dangerous ?x))
     (rule 2 (large ?x) => (heavy ?x))
     (rule 3 (small ?x)(light ?x)
          => (portable ?x)(fragile ?x))
     ))
```

Given these definitions it is possible to develop a function to apply all rules once:

```
(defn apply-all [rules facts]
  (reduce concat
    (map #(apply-rule % facts) rules)
    ))

(apply-all rules facts)
→ ((hard pebble)    (hard boulder)
   (fragile daisy) (portable daisy))
```

For simplicity in combining the output of rules we use sets which necessitates modifying the apply-all function to:

```
(defn apply-all [rules facts]
  (set (reduce concat
         (map #(apply-rule % facts) rules)
         )))
```

A forward chaining/fact deduction function which continues to operate while it is generating new facts can then be defined:

```
(defn fwd-chain [rules facts]
  (let [new-facts (apply-all rules facts)]
    (if (subset? new-facts facts)
      facts
      (recur rules (union facts new-facts))
      )))

(fwd-chain rules (set facts))
→ #{(light daisy)  (mineral boulder)
    (hard boulder) (fragile daisy)
    (small daisy)  (heavy boulder)
    (broken daisy) (small pebble)
    (mineral pebble) (hard pebble)
    (portable daisy) (on boulder daisy)
    (large boulder)}
```

As with the previous example, the matcher performs most of the processing (in this case using a *defmatch* construct and *mfor** in apply-rule) while other functions collate results, etc.

```
(defmatch apply-rule [facts]
  ((rule ?n ??antecedents => ??consequents)
    :=> (mfor* [(? antecedents) facts]
          (mout (? consequents)))))

(defn apply-all [rules facts]
  (reduce concat
    (map #(apply-rule % facts) rules)
    ))

(defn fwd-chain [rules facts]
  (let [new-facts (apply-all rules facts)]
    (if (subset? new-facts facts)
      facts
      (recur rules (union facts new-facts))
      )))
```

4.3 Application of Operators

In this example we consider how to apply the kind of state changing operators that are used in some planning systems. Broadly we adapt a representation borrowed from PDDL [4, 9] for use with a STRIPS [6] style solver. The operators are specified in terms of their preconditions and their effects. As with the earlier examples, we use tuples to capture state information. The following tuples, for example, describe a simple state in which some (animated) agent (R) is at a table, holding nothing and a book is on the table.

```
#{(at R table)         (on book table)
  (holds R nil)        (path table bench)
  (manipulable book)   (agent R) }
```

In order to generalise an operator (so it can be used with different agents, objects and in various locations) it is necessary to specify it using variables, in this case matcher variables. An operator which describes a "pickup" activity for an agent and which can be

used to produce a new state (new tuples) can be described as follows:

```
{:pre ((agent ?agent)
       (manipulable ?obj)
       (at ?agent ?place)
       (on ?obj   ?place)
       (holds ?agent nil)
       )
 :add ((holds ?agent ?obj))
 :del ((on ?obj   ?place)
       (holds ?agent nil))
 }
```

The operator is a map with three components (i) a set of preconditions which must be satisfied in order for the operator to be used (ii) a set of tuples to add to an existing state when producing a new state and (iii) a set of tuples to delete from an existing state.

To apply this kind of operator specification we extract patterns from the operator then use *mfind**

```
(defn apply-op
  [state {:keys [pre add del]}]
  (mfind* [pre state]
    (union (mout add)
      (difference state (mout del))
      )))

 (apply-op state1 ('pickup ops))
 →  #{(agent R) (holds R book)
      (manipulable book)
      (path table bench) (at R table)}
```

As with the previous examples, the patterns used by *mfind** are provided dynamically when apply-op is called. Furthermore, in this example, the patterns themselves define the semantics of the operators.

Collections of operators are conveniently held in a map and ordered sequences of operator applications can be formed by chaining apply-op calls, e.g.

```
(def ops
  '{pickup {:pre ((agent ?agent)
                  (manipulable ?obj)
                  (at ?agent ?place)
                  (on ?obj   ?place)
                  (holds ?agent nil)
                  )
            :add ((holds ?agent ?obj))
            :del ((on ?obj   ?place)
                  (holds ?agent nil))
            }
    drop    {:pre ((at ?agent ?place)
                   (holds ?agent ?obj))
             :add ((holds ?agent nil)
                   (on ?obj   ?place))
             :del ((holds ?agent ?obj))
             }
    move    {:pre ((agent ?agent)
                   (at ?agent ?p1)
                   (path ?p1 ?p2)
                   )
             :add ((at ?agent ?p2))
             :del ((at ?agent ?p1))
    }})
```

```
(-> state1  (apply-op ('pickup ops))
            (apply-op ('move   ops))
            (apply-op ('drop   ops)))

→ #{(agent R) (manipulable book)
    (on book bench) (holds R nil)
    (at R bench) (path table bench)}
```

We can further develop this example so apply-op (or some similar function) works with a search process or a STRIPS-style planner to generate sequences of moves in order to reach a goal state.

5. SUMMARY & CONCLUSION

This paper has argued that pattern matching can be used to simplify some programming tasks, facilitating the production of concise, well-formed code with precise semantics. We have presented a symbolic pattern matcher (now available under "clojure resources" at www.agent-domain.org) which binds immutable match variables and provides matcher functions/macros to support pattern-based conditional statements, function definitions and iterative/mapping forms. The matcher has some forms which take literal patterns but, importantly, has others which allow their patterns to be retrieved from data structures or to be constructed at run-time. This provides increased flexibility in pattern production and use; allowing some types of rules and state-change operators to have their semantics described in terms of patterns. These in turn facilitate the construction of inference engines which apply these structures. In the evaluation section we have presented three sample problems (searching tuples, applying rules and using operators) and demonstrated how the matcher can be employed to solve these problems.

6. REFERENCES

[1] Barr, A., Feigenbaum, E.A., The Handbook of Artificial Intelligence, (1981) vol 1, II F

[2] Fogus, M., Houser, C., Joy of Clojure (2nd ed.) ch. 16, (2014)

[3] Franz, B. and Snyder. W., "Unification Theory." *Handbook of automated reasoning 1* (2001): 445-532.

[4] Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., ... & Weld, D. (1998). PDDL-the planning domain definition language.

[5] Jones, T., "Artificial Intelligence: a systems approach" Jones & Bartlett, ISBN 978-0-7637-7337-3 (2009)

[6] Lekavý, M., Návrat, P. Expressivity of STRIPS-Like and HTN-Like Planning, Agent and Multi-Agent Systems: Technologies and Applications (2007), LNCS 4496, 121-130

[7] Odersky, M.: The Scala Language Specification (2008)

[8] Oliart, A., and Snyder, W., "Fast Algorithms for Uniform Semi-Unification," Journal of Symbolic Computation 37 (2004) 455-484.

[9] De Weerdt, M., T. Mors, A.T., Witteveen, C., Multi-agent planning: An introduction to planning and coordination In: Handouts of the European Agent Summer (2005), pp. 1-32

[10] Younger, B., "Matchure", (2013) https://clojars.org/brendanyounger/matchure