The Pattern Matcher

From OpenCog

Contents

Theory

In the last lesson we added Atoms to the Atomspace using the scheme shell. We also used a function named cog-execute! to use the Atoms to add two numbers. In this lesson, we will look at some other functions to interact with Atomspace.

What is the Pattern Matcher(PM)?

The pattern matcher is the query engine that is used to find Atoms in Atomspace that fit into a certain template or pattern. The PM can be used from c++ or scheme. Here we will use it from scheme. A pattern here means a hypergraph consisting of Nodes and Links of several types. One type of Node is the VariableNode. If the pattern has some VariableNodes then it can be 'grounded'. Grounding means looking for other patterns in the Atomspace that match exactly with the pattern to be grounded except at the VariableNodes. One can think of grounding like filling the blanks in a sentence by looking for similar sentences in a passage.

The scheme interface to the PM is implemented by the functions: cog-satisfy, cog-satisfying-set and cog-bind (see The Simplified API). Both cog-satisfy and cog-satisfying-set can be used to ground patterns specified inside SatisfactionLinks. The difference between cog-satisfy and cog-satisfying-set is that the former returns TruthValues and the later returns Atoms. We will look at what these things mean in following examples.

The cog-bind function is used to rewrite the hypergraphs in the Atomspace. The patterns are specified for cog-bind function inside a BindLink. This function is used to look for possible groundings of a pattern and add new hypergraphs into the Atomspace based on those findings.

Practice

A Background in Scheme

Scheme is a dialect of Lisp; the scheme shell allows scheme code to manipulate the contents of an OpenCog AtomSpace.

- If you don't have a background in Scheme, it might be worth going through the Scheme documentation.

Initial setup

We need to add some Atoms in the Atomspace so that we can apply the above mentioned functions. We make a file helloPM.scm:

```
vim helloPM.scm
```

note you can't create or edit files within the scheme commandline, open another terminal or exit to bash

This script will just add some Atoms to the Atomspace and exit. Add the following script to the helloPM.scm file:

```
;Boilerplate code for loading the opencog modules
;You should skip these if you have already put these in ~/.guile

(use-modules (ice-9 readline))
(activate-readline)
(add-to-load-path "/usr/local/share/opencog/scm")
(add-to-load-path ".")
(use-modules (opencog))
(use-modules (opencog query))
(use-modules (opencog exec))
;END Boilerplate code

(display "--------------------------------------------------------------------")
(newline)(newline)
;Utility function to create InheritanceLinks
(define (typedef type instance)
    (InheritanceLink
        (ConceptNode instance)
        type
    )
)

;Types of entities
(define color
    (ConceptNode "Color")
)

(define animal
    (ConceptNode "Animal")
)

;Some instances of entities
(typedef color "Blue")
(typedef color "Green")
(typedef color "Red")

(typedef animal "fish")
(typedef animal "dog")
(typedef animal "cat")
```

Run the script on the bash shell:

```
$ guile helloPM.scm
```

Or within the scheme commandline:

```
$ (load "helloPM.scm")
```

Now we can start with the Pattern Matcher (PM) functions.

SatisfactionLink: cog-satisfy and cog-satisfying-set

As said earlier, patterns can be grounded *if* they have VariableNodes - we use the cog-satisfy function to ground patterns. So, we will first *specify a pattern* and then *ground it* using the *cog-satisfy* function.

Specify a pattern: Suppose we want to find out what colors there are in the Atomspace. For this we can specify a pattern as follows (all of these snippets are to be appended to the end of helloPM.scm.)

```
;Define a pattern that is satisfiable by colors
(define colornode
    (SatisfactionLink
        ;Declare varibales [optional]
        (VariableNode "$color")
        ;The pattern that the variable must satisfy
        (InheritanceLink
            (VariableNode "$color")
            (ConceptNode "Color")
        )
    )
)
```

Ground the pattern: Now, we can look for all Atoms in the Atomspace that can be substituted in place of (VariableNode "$color") in the pattern above. All such atoms must be linked to the (ConceptNode "Color") via an InheritanceLink to be valid matches. Lets call the two functions of this section to find such atoms.

- Ground the pattern using *cog-satisfy*: `(display (cog-satisfy colornode))(newline)`

The result of running *cog-satisfy* is a TruthValue. It is printed to the screen as (stv 1 1). This means that atoms were found that satisfied the pattern.

- Ground the pattern using *cog-satisfying-set*: `(display (cog-satisfying-set colornode))(newline)`

The result of running *cog-satisfying-set* will be a SetLink, that is connecting all Atoms that matched the pattern. This will be printed to the screen as:

```
(SetLink
   (ConceptNode "Blue")
   (ConceptNode "Green")
   (ConceptNode "Red")
)
```

(Note the text above is not code)

BindLink: cog-bind

The cog-bind function can be used to deliver graph rewrite queries to Atomspace. This is accomplished in conjunction with BindLink. Let us look at an example.

What we are doing here is defining a *graph-rewrite* called *rewrite* (not that imaginative I know) The query looks for notes of type Animal, and then label these nodes as pets. Code:

```
(define rewrite
    (BindLink
        ;Declare the variables [optional]
        (VariableNode "$denizen")
        ;Declare the pattern used to ground the variables
        (InheritanceLink
            (VariableNode "$denizen")
            (ConceptNode "Animal")
        )
        ;If a match is found for the pattern then we want
        ;to add the following hypergraph ot the Atomspace
        (InheritanceLink
            (VariableNode "$denizen")
            (ConceptNode "Pet")
        )
    )
)
```

Now we execute the *graph-rewrite* query (unimaginatively called 'rewrite')using cog-bind. Code: `(display (cog-bind rewrite))`

The *output* of this code will be:

```
(SetLink
   (InheritanceLink
      (ConceptNode "fish")
      (ConceptNode "Pet")
   )
   (InheritanceLink
      (ConceptNode "dog")
      (ConceptNode "Pet")
   )
   (InheritanceLink
      (ConceptNode "cat")
      (ConceptNode "Pet")
   )
)
```

Now, you can use another pattern and the cog-satisfying-set function to check that the *Pet* nodes are indeed added in the Atomspace. This will be done just like we did it for the colors above.

GetLink: cog-satisfying-set and cog-execute!

The GetLink is just like the SatisfactionLink, except that it can also be executed with the cog-execute! function. Let us use SatisfactionLink (with cog-satisfying-set) and GetLink (with both cog-satisying-set and cog-execute!) to find the pets in the Atomspace.

```
;Get the list of pets in the Atomspace
(define petnode
    (SatisfactionLink
        ;Declare varibales
        ;This is how you specify that the VariableNode "$animal"
        ;should only be grounded by a ConceptNode. We are constraining
        ;the type of the VariableNode to a ConceptNode.
        (TypedVariableLink
            (VariableNode "$animal")
            (TypeNode "ConceptNode")
        )
        ;The pattern that the variable must satisfy
        (InheritanceLink
            (VariableNode "$animal")
            (ConceptNode "Pet")
        )
    )
)
(display "SatisfactionLink with cog-satisfying-set")(newline)
(display (cog-satisfying-set petnode))

;GetLink is just like the SatisfactionLink except that it can also
;be executed using cog-execute
(define executablepetnode
    (GetLink
        ;Declare varibales [optional]
        (TypedVariableLink
            (VariableNode "$animal")
            (TypeNode "ConceptNode")
        )
        ;The pattern that the variable must satisfy
        (InheritanceLink
            (VariableNode "$animal")
            (ConceptNode "Pet")
        )
    )
)
(display "GetLink with cog-satisfying-set")(newline)
(display (cog-satisfying-set executablepetnode))
(display "GetLink with cog-execute!")(newline)
(display (cog-execute! executablepetnode))
```

PutLink: cog-execute!

PutLink provides a way to execute write queries in Atomspace using the cog-execute! function. This is the second method to execute write queries after the BindLink method. You can obviously directly write Nodes and Links into Atomspace as we did in the Initial setup section of this chapter. Using the PutLink enables the use of VariableNodes. First we will look at how PutLink can be used to write hypergraphs into Atomspace and then we will see how we can combine it with GetLink.

Writing new nodes using PutLink: We specify a pattern in the PutLink that is to be written to Atomspace, and we also provide a list of VariableNodes that are to be used to ground the pattern before it is written.

```
;Write with PutLink
(define writequery
    (PutLink
                ;The pattern to write into Atomspace
        (InheritanceLink
            (VariableNode "$x")
            (ConceptNode "PrimaryColor")
        )
                ;The nodes used to ground the pattern
        (SetLink
            (ConceptNode "Red")
                        (ConceptNode "Green")
                        (ConceptNode "Blue")
        )
    )
)
(display (cog-execute! writequery))

;Check that the node was written
(define primarycolors
    (GetLink
        (TypedVariableLink
            (VariableNode "$color")
            (TypeNode "ConceptNode")
        )
        ;The pattern that the variable must satisfy
        (InheritanceLink
            (VariableNode "$color")
            (ConceptNode "PrimaryColor")
        )
    )
)
(display (cog-execute! primarycolors))
```

You would see the following output, meaning that the primary color was written to the Atomspace.

```
(SetLink
   (ConceptNode "Red")
)
```

Combining with GetLink: Now we use GetLink to get the Nodes that are to be used to ground the pattern in PutLink from the Atomspace. Used this way, the query is equvalent to those provided by BindLink.

```
;Combining PutLink and GetLink together
(define writequery
    (PutLink
        ;The pattern to be written to Atomspace
        (InheritanceLink
            (VariableNode "$x")
            (ConceptNode "PrimaryColor")
        )
        ;The GetLink to search the Atomspace for grounding nodes
        (GetLink
            ;Variable declaration
            (TypedVariableLink
                (VariableNode "$color")
                (TypeNode "ConceptNode")
            )
            ;Pattern
            (InheritanceLink
                (VariableNode "$color")
                (ConceptNode "Color")
            )
        )
    )
)
(display (cog-execute! writequery))

;Check that the node was written
(define primarycolors
    (GetLink
        (TypedVariableLink
            (VariableNode "$color")
            (TypeNode "ConceptNode")
        )
        ;The pattern that the variable must satisfy
        (InheritanceLink
            (VariableNode "$color")
            (ConceptNode "PrimaryColor")
        )
    )
)
(display (cog-execute! primarycolors))
```

You should now see the output:

```
(SetLink
   (ConceptNode "100")
   (ConceptNode "010")
   (ConceptNode "001")
   (ConceptNode "Red")
)
```

ChoiceLink, TypeChoice

These link is unlike the others above. This cannot help to execute any queries on its own. The use of ChoiceLink is to create complex patterns for grounding. The use of TypeChoice is similar. Let us look at an example.

```
;Find all nodes that are either primarycolors or colors
(define getcolors
    (GetLink
        ;Variables
        (TypedVariableLink
            (VariableNode "$obj")
            ;The TypeChoice link can be used to constrain the
            ;type of a node to two or more types.
            (TypeChoice
                (TypeNode "VariableNode")
                (TypeNode "ConceptNode")
            )
        )
        ;Pattern: Nodes satisfying any of the choices of patterns
        ;will be returned
        (ChoiceLink
            ;Choice1
            (InheritanceLink
                (VariableNode "$obj")
                (ConceptNode "Color")
            )
            ;Choice2
            (InheritanceLink
                (VariableNode "$obj")
                (ConceptNode "PrimaryColor")
            )
        )
    )
)
(display "All colors\n")
(display (cog-execute! getcolors))
```

Reading this snippet may give the impression that the ChoiceLink is like the logical OR for patterns. This is not the case. The logical OR operator for building groundable patterns is provided through OrLink.

Quiz

1. What is the Pattern Matcher?

- ○ an interface to ground nodes for consumption by MOSES.
- ○ the query engine that is used to find Atoms in Atomspace that fit into a certain template or pattern.
- ○ within the atomspace, it's the agent-factory that cyclically produces the correct XML that glues relevant concepts together.
- ○ the section of the opencog sql query engine which interprets WHERE and HAVING clauses.
- ○ the opencog fork of regex (based on perl-like syntax).

2. In the context of the opencog Pattern Matcher, what is a pattern?

- ○ a cluster of interesting or surprising configurations of nodes and links as opposed to just noisy/chaotic groupings of nodes and links
- ○ a collection of nodes and links of a variety of types (each link may span any number of nodes)
- ○ a collection of atoms (in MOSES)
- ○ it can be anything - every material is just made of patterns of atoms

3. Grounding a pattern requires?

- ○ clear and sound logic.
- ○ the successful tests of empirically verifiable (and ideally falsifiable) hypotheses to justify a strong correspondence between the pattern in atomspace and the observed pattern in reality.
- ○ at least one VariableNode.
- ○ filling the blanks in a pattern by looking for similar patterns in atomspace.
- ○ asking the underpants gnomes to press the ground-it button.

4. cog-satisfy...

- ☐ is used to rewrite the hypergraphs in the Atomspace
- ☐ can be used to invoke Moses
- ☐ is used to look for possible groundings of a pattern and add new hypergraphs into the Atomspace

based on given findings
- ☐ can be used to ground patterns specified inside SatisfactionLinks
- ☐ returns TruthValues
- ☐ returns Atoms
- ☐ returns Hypergraphs

5. cog-satisfying-set...

- ☐ returns Hypergraphs
- ☐ is used to look for possible groundings of a pattern and add new hypergraphs into the Atomspace based on given findings
- ☐ returns Atoms
- ☐ can be used to invoke Moses
- ☐ is used to rewrite the hypergraphs in the Atomspace
- ☐ can be used to ground patterns specified inside SatisfactionLinks
- ☐ returns TruthValues

6. cog-bind...

- ☐ returns Atoms
- ☐ is used to look for possible groundings of a pattern and add new hypergraphs into the Atomspace based on given findings
- ☐ can be used to invoke Moses
- ☐ can be used to ground patterns specified inside SatisfactionLinks
- ☐ returns Hypergraphs
- ☐ returns TruthValues
- ☐ is used to rewrite the hypergraphs in the Atomspace

7. Based on the examples above, what will be the output of `(display (cog-satisfy colornode))(newline)`?

- ○ (stv 1 1)
- ○ yes
- ○ 1
- ○ (SetLink
  (ConceptNode "Blue")
  (ConceptNode "Green")
  (ConceptNode "Red")
)
- ○ 0

8. Based on the examples above, what will be the output of `(display (cog-satisfying-set colornode))(newline)`?

- ○ (stv 1 1)
- ○ yes

- ○ 1
- ○ (SetLink
  (ConceptNode "Blue")
  (ConceptNode "Green")
  (ConceptNode "Red")
)
- ○ 0

Submit

Further Reading

The links below provide detailed descriptions of all topics in this chapter. This tutorial has been severely restricted selection from these pages. One should explore these links to gain a deeper understanding of the topics introduced here.

- The Simplified API
- Examples with GetLink and PutLink
- Examples with ChoiceLink

Notes

Linus is the Pattern Matcher Master!