Predicate Calculus and Jess

Wolfgang Laun
Thales Rail Signalling GesmbH
Vienna, Austria

5 February 2009

1 Introduction

Occasionally Jess novices wonder how the objects of Jess relate to more theoretical concepts they've encountered while studying such topics as Propositional Logic or First Order Predicate Calculus. Moreover, Jess pattern operators such as `forall` or terms such as "binding" are eponymous with constituents from these mathematical domains without being their straightforward implementations.

This article discusses a few elementary sentences of predicate calculus and how they are reflected by Jess constructs.

2 Propositions

"A proposition is a statement of some alleged fact which is either true or false." How does this definition which uses the Jess term *fact* relate to Jess and its facts? That's simple: any proposition (as they are used in Propositional Logic) can be represented as a Jess fact. Here, for instance, is one Jess representation of the facts that Italy is a country and Rome is a city, moreover the capital of Italy:

```
(Country Italy)
(City    Rome)
(Capital (country Italy)(city Rome))
```

Nothing keeps you from representing another "fact" such as, e.g.:

```
(Capital (country France)(city Brussels))
```

but any real world application would, of course, be based on propositions that are true by real world standards. From now on, we'll simply assume that Jess facts are true propositions.

The propositions $country(x)$ and $city(x)$ are represented as unordered facts, whereas the the proposition $capital(x, y)$ is a compound proposition, which are more conveniently represented by an ordered fact. This corresponds very well to the familiar concept of an object with its attributes.

A very simple Jess rule would answer the question whether a simple proposition is true or false. Here is a rule testing the truth of a proposition:

```
(defrule CityRome
  (City Rome)
=>
  (printout t "Yes, Rome is a city." crlf)
)
```

We can also write a rule that responds to a false proposition. Well, actually, it responds to a true proposition since we have to negate the false one.

```
(defrule notCityFoo
  (not (City Foo))
=>
  (printout t "Yup, Foo isn't a city." crlf)
)
```

The obvious question is: what do I do if I don't know what to expect and want to have an answer, yes or no? To get a reaction in either case we'll have to provide both patterns and join them with the disjunctive operator "or".

```
(defrule CityOrNoCityBar    ; not very useful
  (or      (City Bar)
      (not (City Bar)))
=>
   ...
)
```

Now the question is: is Bar a city or isn't Bar a city? The answer's going to be yes in either case, so we're none the wiser if the right hand side fires. (Actually this rule fires once after a city "Bar" has been asserted, and once after it has been sacked, but we wouldn't know which.) We'll have to find a way to determine which term of the conjunction was responsible for the overall match. Usually we can determine what was matched by adding a variable binding, but this won't work in a "not" conditional element. But we can extend both terms of the conjunction with a pattern for one out of two different existing facts. Then we'll have a bound variable in any case so that we can assess the situation by inspecting the value of the bound variable.

```
(defrule CityOrNoCityBar-1
  (or (and      (City Bar) (Boolean TRUE  & ?b))
      (and (not (City Bar))(Boolean FALSE & ?b)))
=>
  (if (eq ?b TRUE) then
    (printout t "Yes, Bar is a city." crlf)
  else
    (printout t "No, Bar isn't a city." crlf)
  )
)
```

As an alternative, we can also bind the `City` fact in the first term and check, on the right hand side, whether this variable is bound to a fact.

```
(defrule  CityOrNotCityFoo
  (or ?city <- (City Foo)
          (not (City Foo)))
=>
  (if (factp "city") then
    (printout t "Yes, Foo is a city." crlf)
  else
    (printout t "No, Foo isn't a city." crlf)
  )
)
```

The function `(factp)` isn't predefined, but it's simple to write. (Notice that this technique isn't absolutely safe because the variable might have been bound to a fact in the global context.)

```
(deffunction factp (?name)
  (if ((context) isVariableDefined ?name) then
    (bind ?value  ((context) getVariable ?name))
```

```
    (return (eq ((?value getClass) getName) "jess.Fact"))
  else
    (return FALSE)
  )
)
```

A simple equivalence of Propositional Calculus with a proposition P is

$$P \wedge P \equiv P$$

Since Jess facts F_1, F_2,... are all true, keeping two identical facts is useless. This is incorporated in the Jess mechanics: it's impossible to add a fact to the Jess collection of facts that is identical to an already existing fact.

3 Predicates

3.1 Quantifiers

In the process of reasoning about propositions, more expressive power is provided by *predicates*. A predicate states a specific property of an object, or lets us conveniently express relations between two or more objects. Here are some predicates:

$$\begin{aligned} city(x) \\ capitalOf(x, y) \end{aligned}$$

The variables x and y can be instantiated with objects

$$city(Rome)$$

resulting in a simple proposition. More interesting constructs are obtained by *quantification*, using the symbols $\forall$ (pronounced "for all") and $\exists$ ("exists"):

$$\begin{aligned} \forall\, x \bullet P(x) \\ \exists\, x \bullet P(x) \end{aligned}$$

This creates *quantified* propositions, with the first one stating that the proposition P holds for *all* objects x, and the second one asserting that proposition P holds for em at least one object x. It's obvious that usually some constraint on the domain of the variable is useful which can be given in the construct as shown below.

$$\forall\, x : \mathbb{N} \bullet P(x)$$

This is just a more readable form for

$$\forall\, x \bullet x \in \mathbb{N} \wedge P(x)$$

where the restriction on the variable is introduced by an additional predicate.

It should be fairly obvious that these two quantifications are merely a more convenient notation for disjunctions and conjunctions:

$$\begin{aligned} \forall\, x \bullet P(x) \equiv P(x_1) \wedge P(x_2) \wedge ... \\ \exists\, x \bullet P(x) \vee P(x_1) \vee P(x_2) \wedge ... \end{aligned}$$

Using these equivalences and applying de Morgan's laws, you can obtain the following equivalences:

$$\neg \forall\, x \bullet P(x) \equiv \exists\, x \bullet \neg P(x) \quad (1)$$

$$\neg \exists\, x \bullet P(x) \equiv \forall\, x \bullet \neg P(x) \quad (2)$$

3.2 Simple Rule Patterns

How does this relate to Jess? Let's assume that we have lots of facts about countries and cities. Then the simple rule

```
(defrule print-cities
  (City (name ?x))
=>
  (printout t ?x crlf)
)
```

prints all cities which we have in our working memory. Apparently, something has been done "for all City facts". The $\forall$ quantifier can be used to describe the effects:

$$\forall\, c : City \bullet printed(c)$$

The predicate $printed(c)$ is meant to describe the post-condition that we have a printed line with the name of some city.

Logicians refer to x in a term such as $\forall x \bullet P(x)$ as a *bound* variable, meaning that it is bound to the semantic meaning of the quantifier. It is, for instance, not possible to replace a bound variable with an object when a predicate is to be turned into a proposition. In Jess, we also use the term "bound" to describe that a variable is used as a (read only) handle to refer to a value from one slot from a single fact in the process of one firing of a rule.

The "for all x in X" is applied for any conditional expression in a rule. Here is a (not very useful) rule with two conditional expressions.

```
(defrule cc
  (City    (name ?x))
  (Country (name ?y))
=>
  (printout t ?x "/" ?y crlf)
)
```

The predicate describing the situation after the rule has stopped working can be written as

$$\forall\, x : City \bullet \forall\, y : Country \bullet printed(x, y)$$

We should note that a similar rule can be written where the "for all x in X" is used twice with the same domain.

```
(defrule move
  (City (name ?x))
  (City (name ?y))
=>
  (printout t ?x "-" ?y crlf)
)
```

This is the predicate sentence describing the resulting situation:

$$\forall\, x : City \bullet \forall\, y : City \bullet printed(x, y)$$

The predicate term tells us that we have printed ordered pairs of city names. There is no restriction with respect to the way the pairs are formed. Thus, we'll find pairs of pairs with swapped cities, and pairs with both cities being equal.

In a way, the firing of these rules is also a proof for the sentences where $\forall$ is replaced by $\exists$. But for all practical purposes we would prefer to have a simpler way of establishing the existence of any city than to receive a list of all cities!

3.3 Testing the Existence

Jess provides an easy way for evaluating the predicate $\exists\, x : X \bullet P(x)$.

```
(defrule ExistsCity
  (exists (City ?))
=>
  (printout t "There is a city." crlf)
)
```

The question mark is necessary to match a `City` fact with exactly one value in its multislot. If we test for the presence of a fact from an unordered template, the template name alone is sufficient.

Now, let's try the negation of existence, i.e., the absence of any fact from a domain. One might be tempted to try it like this:

```
(defrule NotExistsShangriLa
  (not (exists (Country ShangriLa)))   ; Not recommended!
=>
  (printout t "There is no ShangriLa." crlf)
)
```

Although this works, it is not the best way. Remember that we were using `not` in combination with some simple pattern to construct a match for the absence of facts of some kind – and that's just what the negated existence means. Hence, the absence of any "Shangri La" is nicely confirmed by the rule shown below.

```
(defrule NotShangriLa
  (not (Country ShangriLa))
=>
  (printout t "There is no ShangriLa." crlf)
)
```

3.4 For All Objects in a Domain

We have seen that the patterns of a rule's left hand side (without `exists` or `not` are implicitly matched against all the facts. We have seen that the combined results of all the firings can be described with a predicate using the $\forall$ operator. That, however, is not quite the same as a $\forall$-predicate that is true because it is true for *all* objects of the domain of the bound variable.

Let's assume that we want to make processing dependent upon the consistency of our collection of capitals. The slots `city` and `country` ought to have entries that are indeed `City` or `Country` facts, respectively. We can find the misfits by a rule like this:

```
(defrule NotCapitalOK-1
  (Capital (city ?city) (country ?country))
  (or (not (City    ?city))
      (not (Country ?country)))
=>
  (printout t "Capital " ?city " of " ?country " not OK" crlf)
)
```

The post-condition of this rule is expressed by the sentence given below where we use the tuple notation $\langle a, b, ...\rangle$ for ordered Facts with slot values a, b,...

$$\forall\, \langle x, y\rangle : Capital \bullet \neg\, \exists\, City(x) \vee \neg\, \exists\, Country(y) \bullet printed(x, y)$$

The problem that cannot be solved with the patterns we have seen so far is to avoid firing of a rule unless *all* objects of a domain meet a specific condition. This is not what the following rule does, which will fire for *any* valid Capital fact.

```
(defrule ValidCapital
  (Capital (city ?city) (country ?country))
  (City    ?city)
  (Country ?country)
=>
  (printout t ?city " is the capital of " ?country "." crlf)
)
```

We might try to negate the entire pattern from `NotCapitalOK-1`, as shown below:

```
(defrule AllCapitalsOK-X
  (not (                          ; ### ERROR ###
    (Capital (city ?city) (country ?country))
    (or (not (City    ?city))
        (not (Country ?country))))
=>
  (printout t "All Capital facts are OK." crlf)
)
```

But this is not permitted because `not` can only be wrapped around a *single* pattern.

Jess provides the conditional element `forall` to solve this particular problem . It is used, as shown below, to request that the entire pattern it contains must match *for all objects of the first domain.*

```
(defrule AllCapitalsOK
  (forall (Capital (city ?city) (country ?country))
          (City    ?city)
          (Country ?country))
=>
  (printout t "All Capital facts are OK." crlf)
)
```

Again, we can formulate the predicate describing the situation after the rule has fired.

$$\forall \langle x, y \rangle : Capital \bullet \exists City(x) \wedge \exists Country(y) \bullet printed(m)$$

We might try to derive the negation – the confirmation that we have some bad capital – by predicate calculus. We begin with a sentence that we are interested in the existence of a *Capital* with a non-existent city or a non-existent country.

$$\begin{aligned} \exists \langle x, y \rangle : Capital \bullet \neg \exists City(x) \vee \neg \exists Country(y) &= \\ \exists \langle x, y \rangle : Capital \bullet \neg (\exists City(x) \wedge \exists Country(y)) &= \quad ... \end{aligned}$$

Now we can apply equation (1) to extract the negation from the inner predicate, obtaining

$$\neg \forall \langle x, y \rangle : Capital \bullet \exists City(x) \wedge \exists Country(y)$$

Trusting the mathematical result we come up with this rule, which indeed does what we expect it to.

```
(defrule ExistsCapitalNotOK
  (not (forall (Capital (city ?city) (country ?country))
               (City    ?city)
               (Country ?country)))
=>
  (printout t "Some Capital fact is not OK." crlf)
)
```

4 The Jess Code

```
(clear)

(deffunction factp (?name)
  (if ((context) isVariableDefined ?name) then
    (bind ?value  ((context) getVariable ?name))
    (return (eq ((?value getClass) getName) "jess.Fact"))
  else
    (return FALSE)
  )
)

(deftemplate Boolean (declare (ordered TRUE)))

(deftemplate City      (declare (ordered TRUE)))
(deftemplate Country   (declare (ordered TRUE)))
(deftemplate Capital   (slot country)(slot city))

(deffacts ccc
  (Boolean TRUE)
  (Boolean FALSE)
  (City Rome)
  (City Paris)
  (City London)
  (City Berlin)
  (Country Italy)
  (Country France)
  (Country England)
  (Country Germany)
  (Capital (city Rome)  (country Italy))
  (Capital (city Paris) (country France))
  (Capital (city London)(country England))
  (Capital (city Berlon)(country Germany))  ;; a bad Capital fact
)

; city(Rome)
;
(defrule CityRome
  (City Rome)
=>
  (printout t "Right, Rome is a city." crlf)
)

(defrule notCityBar
  (not (City Bar))
=>
  (printout t "Right, Bar isn't a city." crlf)
)

(defrule  CityOrNotCityFoo
  (or ?city <- (City Foo)
  (not (City Foo)))
=>
```

```
  (if (factp "city") then
    (printout t "Yes, Foo is a city." crlf)
  else
    (printout t "No, Foo isn't a city." crlf)
  )
)

(defrule CityOrNoCityBar
  (or (and      (City Bar) (Boolean TRUE  & ?b))
      (and (not (City Bar))(Boolean FALSE & ?b)))
=>
  (if (eq ?b TRUE) then
    (printout t "Yes, Bar is a city." crlf)
  else
    (printout t "No, Bar isn't a city." crlf)
  )
)


(defrule ExistsCity
  (exists (City ?))
=>
  (printout t "There is a city." crlf)
)

(defrule ExistsCapital
  (exists (Capital))
=>
  (printout t "There is a capital." crlf)
)

(defrule NotExistsShangriLa
  (not (exists (Country ShangriLa)))
=>
  (printout t "There is no ShangriLa." crlf)
)

(defrule NotShangriLa
  (not (Country ShangriLa))
=>
  (printout t "There is no ShangriLa." crlf)
)

(defrule NotCapitalOK-1
  (Capital (city ?city) (country ?country))
  (or (not (City    ?city))
      (not (Country ?country)))
=>
  (printout t "Capital " ?city " of " ?country " not OK" crlf)
)

(defrule AllCapitalsOK-2
  (forall (Capital (city ?city) (country ?country))
          (City    ?city)
```

```
          (Country ?country))
=>
  (printout t "All Capital facts are OK." crlf)
)

(defrule ExistsCapitalNotOK
  (not (forall (Capital (city ?city) (country ?country))
          (City    ?city)
          (Country ?country)))
=>
  (printout t "Some Capital fact is not OK." crlf)
)

(reset)
(run)
```