Query Evaluation

In the previous chapter, we learned how to create fact bases and store logical facts. A fact base on its own is just a collection of knowledge — to actually use it, we need to ask questions. This is where queries come in. A query lets us specify a pattern and see which facts match it. In this tutorial, we'll learn how to write queries in DLGP syntax, evaluate them against a fact base, and understand the answers that come back. We'll also look at how variables behave in queries and explore a special kind of query: the boolean query.

All code examples shown here can be found in the accompanying Python file: ex2_query_evaluation.py.

Creating a query

Now that we know how to create a fact base, the next step is to ask questions about it. A fact base on its own is just a collection of facts — to actually use it, we need queries.

A query is essentially a pattern we want to match against the facts in the base. In DLGP syntax, a query has two parts:

- the head, which lists the variables we want to see in the answers
- the body, which describes the conditions the facts must satisfy

The head and body are separated by `:-`, and like all DLGP objects, the query ends with a period.

For example:

- `?(X) :- human(X).` → asks for all X such that X is human
- `?(X, Y) :- parent(X, Y), human(Y).` → asks for all pairs X,Y where X is a parent of Y and Y is human

Creating a query in Python

Just like fact bases, queries are created from DLGP strings. Here's a minimal example:

```
# creation of graal instance and factbase, as seen in ex1
from py4graal.graal import Graal
graal = Graal()
factbase = graal.create_factbase("human(socrates). human(plato).")
# creation of the query
query = graal.create_query("?(X) :- human(X).")
```

> **Note:** A query object in Python works much like a fact base object: it's a handle to an object managed by the Java server. Printing it will export it back to DLGP syntax.

Query evaluation

Once we have a query, we can evaluate it against a fact base to see which facts match the pattern. The result of this evaluation is a set of **substitutions**: they tell us which constants can replace the query variables to make the body true in the fact base.

For example:

```
answers = query.evaluate(factbase)
```

Calling `evaluate()` doesn't immediately give you a list — it returns an `Answers` generator. You can view it as a list, or iterate over it, to get Python dictionaries where each key is a variable from the query head, and each value is the constant that matched.

```
for answer in answers:
    print(answer["X"])
```

Possible output:

```
plato
socrates
```

> **Note:** The order of answers is not guaranteed; it depends on the evaluation engine. In the same way, unicity of answers is not guaranteed either.

Queries with Multiple Atoms

The same process applies when the query body contains several atoms. For example:

```
factbase = graal.create_factbase("human(socrates), human(sophroniscus),
parent(sophroniscus, socrates).")
query = graal.create_query("?(X, Y) :- parent(X, Y), human(Y).")

print(list(query.evaluate(factbase)))
```

Output:

```
[{'X': 'sophroniscus', 'Y': 'socrates'}]
```

Here the answer tells us that substituting X = `sophroniscus` and Y = `socrates` makes both atoms in the query body true in the fact base.

On variables as substitutions values

In the previous examples, all answers were constants. But what happens when the facts themselves contain variables?

By default, query evaluation only returns substitutions that bind variables in the query to constants. If a match uses a variable from the fact base, it is ignored and not included in the answers.

```
factbase_dlgp = "human(socrates), human(sophroniscus), parent(sophroniscus,
socrates), parent(X, sophroniscus)."
factbase = graal.create_factbase(factbase_dlgp)
query = graal.create_query("?(X ,Y) :- parent(X, Y), human(Y).")

for answer in query.evaluate(factbase):
    print(answer)
```

Output :

```
{'X': 'sophroniscus', 'Y': 'socrates'}
```

Here, the second parent fact contains a variable X. The match where X stays a variable is ignored by default, so we only get the constant substitution.

Returning variables with `Images.ALL`

If we want to include variables from the fact base as well, we can use the `images` parameter in `evaluate()` and set it to `Images.ALL`:

```
from py4graal.bridge.params import Images

factbase_dlgp = "human(socrates), human(sophroniscus), parent(sophroniscus,
socrates), parent(X, sophroniscus)."
factbase = graal.create_factbase(factbase_dlgp)
query = graal.create_query("?(X ,Y) :- parent(X, Y), human(Y).")

for answer in query.evaluate(factbase, images=Images.ALL):
    print(answer)
```

Output :

```
{'X': 'sophroniscus', 'Y': 'socrates'}
{'X': 'Graal:EE2', 'Y': 'sophroniscus'}
```

The second answer now appears: it shows that the query can also match facts containing variables. Here, X is mapped to a variable instead of a constant. `Graal:EE2` is a generated name ensuring that the variable is unique inside this fact base (see ex1).

Boolean queries

Sometimes, we don't want to retrieve values — we just want to check whether a pattern exists in the fact base. A boolean query does exactly that: its head has no variables, so it only returns true or false information.

For example, the query `?() :- human(X).` asks: *"Does there exist an X such that human(X) is true in the fact base?"*

```
factbase = graal.create_factbase("human(socrates). human(plato).")
query = graal.create_query("?() :- human(X).")
print(list(query.evaluate(factbase)))
```

Output:

```
[{}]
```

The result `[{}]` means that the query matched at least once. The dictionary is empty because there are no variables to return.

If the pattern is not found in the fact base, the evaluation returns an empty list:

```
[]
```

Using boolean queries in Python

Because `query.evaluate()` can produce a list, we can use it directly as a truth test in Python:

```
if list(query.evaluate(factbase)):
    print("The answer is True.")
else:
    print("The answer is False.")
```

This works because a non-empty list evaluates to `True` and an empty list evaluates to `False`.