# Rules saturation



In the previous chapters, we created fact bases to store knowledge and queries to extract it. Now we add a new ingredient: rules.

Rules allow you to infer new facts automatically based on patterns in the fact base. In this tutorial, you'll also learn how to saturate a fact base — a process where rules are applied until all possible conclusions have been added.

All code examples shown here can be found in the accompanying Python file: ex3_rules_saturation.py.

## Creating a rulebase

A rule base stores a list of rules. Each rule has two parts separated by `:-` : a **head** and a **body**. Both head and body can contain one or more atoms.

Intuitively: *If the atoms in the body are true for some values of the variables, then the atoms in the head also become true for those same values.*

For example:

`mortal(X) :- human(X).` means that every human is mortal.

You can combine multiple rules in one DLGP string by separating them with periods to obtain a rulebase:

```
from py4graal.graal import Graal

graal = Graal()

rulebase_dlgp = "mortal(X) :- human(X). mortal(X) :- cat(X)."
rulebase = graal.create_rulebase(rulebase_dlgp)
```

A rulebase has the same properties as a fact base, described in ex1_objects_creation.

## Saturating a factbase

If we create a fact base of humans and a rule that says "all humans are mortal," what happens when we query for mortal(X)?

```
factbase = graal.create_factbase("human(socrates). human(plato).
human(aristotle).")
rulebase = graal.create_rulebase("mortal(X) :- human(X).")

query = graal.create_query("?(X) :- mortal(X).")

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

Output:

```
[]
```

Nothing is returned because `evaluate()` only considers the facts in the fact base. By default, the rule base is not applied during query evaluation.

### What is saturation ?

One way to use rules is to **saturate** the fact base. Saturation means applying the rules to the existing facts to generate all possible new facts, and then adding those inferred facts to the fact base. Once saturation is complete, the fact base contains both the original facts and everything that can be deduced from the rules.

```
factbase = graal.create_factbase("human(socrates). human(plato).
human(aristotle).")
rulebase = graal.create_rulebase("mortal(X) :- human(X).")

factbase.saturate(rulebase)
print(factbase)
```

Output:

```
human(aristotle), human(socrates), human(plato), mortal(socrates),
mortal(plato), mortal(aristotle).
```

Now that the fact base “knows” the new facts, evaluating the query gives the expected result:

```
query = graal.create_query("?(X) :- mortal(X).")
print(list(query.evaluate(factbase)))
```

Output:

```
[{'X': 'aristotle'}, {'X': 'socrates'}, {'X': 'plato'}]
```

## The case of existential variables

When a variable is present in the head of a rule and not in the body, this variable is an **existential variable**. When we apply such a rule during the saturation, an existential variables produces a "new unknown".

For instance, in the rule `parent(Y, X) :- human(X).`, the variable `Y` is existential. The rule thus means *"every human has a parent"*, or, more formally, *"for every human X, there exists some (unknown) Y that is the parent of X"*.

To implement this generation of "new unknowns", the saturation mechanism replaces existential variables by fresh ones, that are of the form `Graal_EE1`, ensuring their uniqueness inside the fact base.

```
factbase = graal.create_factbase("human(socrates). human(plato).
human(aristotle).")
rulebase = graal.create_rulebase("parent(Y, X) :- human(X).")
print(factbase.saturate(rulebase))
```

Possible output:

```
parent(Graal_EE1, aristotle), parent(Graal_EE2, socrates), parent(Graal_EE3,
plato), human(aristotle), human(socrates), human(plato).
```

If all inferred facts reused the same existential variable, the rule `parent(Y, X) :- human(X).` would imply that every human shares the same parent — clearly not what we want!

## Struggling against redundancies

In the presence of existential variables, saturating a fact base can sometimes introduce redundant facts.

```
factbase = graal.create_factbase("human(socrates), parent(sophroniscus,
socrates).")
rulebase = graal.create_rulebase("parent(Y, X) :- human(X).")
print(factbase.saturate(rulebase))
```

Possible output:

```
human(socrates), parent(sophroniscus, socrates), parent(Graal_EE4,
socrates).
```

Here, the rule inferred parent(Graal_EE4, socrates) ("*Socrates has a parent*") even though the fact base already contained parent(sophroniscus, socrates) ("*Socrates has a parent, and it is Sophroniscus*").

The new fact doesn't add information: it expresses knowledge that was already entailed by the existing fact. In other words, saturation added a redundancy.

In order to reduce the redundancies in the factbase, one can try, whenever we want to insert `parent(Y, X)` in the factbase according to some substitution of `X` (in our case, `{X: socrates}`), to check if there is a substitution of `parent(Y, X)` in the factbase such that `X = socrates`. Such a substitution is called a *folding*. In that case, the insertion of `parent(Graal_EE7, socrates)` in the factbase would only add redundancy,

and we should avoid to do it. A saturation that checks foldings before insertion is called *restricted*, and we can enforce that behavior by adding the parameter `Checker.RESTRICTED` to the `saturate()` call.

```
from py4graal.bridge.params import Checker

factbase = graal.create_factbase("human(socrates), parent(sophroniscus,
socrates).")
rulebase = graal.create_rulebase("parent(Y, X) :- human(X).")
print(factbase.saturate(rulebase, checker=Checker.RESTRICTED))
```

Output:

```
parent(sophroniscus, socrates), human(socrates).
```

> **Note:** By setting the parameter `checker` to `Checker.CORE`, we can even ensure that a (finite) saturation will produce the smallest possible factbase (the *core*), effectively removing all redundancies. Computation of this smallest equivalent factbase, even if only after each saturation step (see `RANK` below) can be costly though, and this is why `Checker.CORE` is seldom used in the case of large factbases.

## Using `rank` to fight infinite saturation

In some cases, the saturation process may not terminate (and the Python program will wait until the java server crashes with `java.lang.OutOfMemoryError`, that Py4J will wrap in a `py4j.protocol.Py4JJavaError`).

Consider, for instance, the fact base `human(socrates).` and the rule base `parent(Y, X), human(Y) :- human(X).`. The saturation of that fact base will endlessly generate new humans (the parent of Socrates, then the parent of his parent, ...).

A *saturation step* on some factbase F is done by applying *non recursively* the rules on F (if applying a rule creates a new application of another rule, it will be done in the next saturation step).

The parameter `rank` of `saturate()` allows to limit the number of steps of saturation. By setting `rank =3`, Integraal only performs 3 saturation steps, ensuring finiteness of saturation, but losing completeness of results (the resulting fact base may not contain all facts that could be inferred without a rank limit).

Here we call `saturate()` three times with `rank=1`. Since saturation updates the fact base in place, this is equivalent to performing one saturation step at a time and lets us observe the intermediate steps (1, 2, 3).

```
factbase =  graal.create_factbase("human(socrates).")
rulebase = graal.create_rulebase("parent(Y, X), human(Y) :- human(X).")

for i in range(3) :
    print(str(factbase.saturate(rulebase, checker=Checker.RESTRICTED, rank=1)))
```

Possible output:

```
human(Graal_EE21), human(socrates), parent(Graal_EE21, socrates).
human(Graal_EE21), human(Graal_EE38), human(socrates), parent(Graal_EE38,
Graal_EE21), parent(Graal_EE21, socrates).
human(Graal_EE21), human(Graal_EE38), human(socrates), human(Graal_EE55),
parent(Graal_EE38, Graal_EE21), parent(Graal_EE55, Graal_EE38),
parent(Graal_EE21, socrates).
```

Limiting the rank is a practical safeguard against infinite derivations in recursive rule sets. It is particularly useful during experimentation, debugging, or when working with existential rules that can easily create infinite chains.

> **Note:** The saturation would have been uglier without Checker.RESTRICTED. Can you see why ?