Compilation

In the previous chapter, we introduced query rewriting as an alternative to saturating the fact base when applying rules. While rewriting is powerful, it can quickly become expensive: for some rule bases and queries, the number of generated query patterns can grow explosively.

To address this, compilation preprocesses the rule base into a form that allows rewriting to be performed more efficiently, without changing the final answers. This approach, based upon Mélanie König's work, separates heavy computation into an initial compilation step, making subsequent queries faster and easier to manage.

All code examples shown here can be found in the accompanying Python file: ex5_compilation.py.

A motivational example

Consider this simple rule base and query. The rules express that both fathers and mothers are parents, while the query asks whether Socrates has any great-grandparent:

```
from py4graal import *

rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y)."
rulebase = graal.create_rulebase(rbstr)
qstr = "?() :- parent(X,Y), parent(Y, Z), parent(Z, socrates)."
query = graal.create_query(qstr)

print(query.rewrite(rulebase))
```

Output:

```
?() :- father(X, Y), parent(Z, socrates), parent(Y, Z).
?() :- parent(X, Y), father(Z, socrates), parent(Y, Z).
?() :- father(Y, Z), parent(X, Y), parent(Z, socrates).
?() :- parent(X, Y), parent(Y, Z), parent(Z, socrates).
?() :- mother(Y, Z), father(X, Y), parent(Z, socrates).
?() :- father(X, Y), mother(Z, socrates), parent(Y, Z).
?() :- mother(X, Y), father(Z, socrates), parent(Y, Z).
?() :- mother(Y, Z), parent(X, Y), father(Z, socrates).
?() :- mother(X, Y), father(Y, Z), parent(Z, socrates).
?() :- father(Y, Z), mother(Z, socrates), parent(X, Y).
?() :- mother(Y, Z), father(X, Y), mother(Z, socrates).
?() :- mother(Y, Z), mother(X, Y), father(Z, socrates).
?() :- mother(X, Y), father(Y, Z), mother(Z, socrates).
?() :- father(Y, Z), father(X, Y), father(Z, socrates).
?() :- mother(X, Y), father(Y, Z), father(Z, socrates).
?() :- mother(Y, Z), father(X, Y), father(Z, socrates).
```

```
?() :- father(Y, Z), father(X, Y), mother(Z, socrates).
?() :- father(Y, Z), parent(X, Y), father(Z, socrates).
?() :- father(X, Y), father(Z, socrates), parent(Y, Z).
?() :- father(Y, Z), father(X, Y), parent(Z, socrates).
?() :- mother(X, Y), mother(Z, socrates), parent(Y, Z).
?() :- mother(Y, Z), mother(Z, socrates), parent(X, Y).
?() :- mother(Y, Z), mother(X, Y), parent(Z, socrates).
?() :- mother(X, Y), mother(Y, Z), mother(Z, socrates).
?() :- mother(Z, socrates), parent(X, Y), parent(Y, Z).
?() :- mother(X, Y), parent(Z, socrates), parent(Y, Z).
?() :- mother(Y, Z), parent(X, Y), parent(Z, socrates).
```

Here, each of the three atoms in the query can be rewritten in three different ways (parent, father, mother), producing 3x3x3 = 27 different queries in total. This explosion of combinations is typical when rewriting queries against even modest rule bases.

However, if we look at the result, all these queries essentially mean:

```
couldBeParent(X, Y), couldBeParent(Y, Z), couldBeParent(Z, socrates).
```

where `couldBeParent` is defined as `parent OR father OR mother`.

This redundancy is the motivation behind compilation: certain rules do not change the shape of the query and can be factored out to avoid generating an exponential number of rewritings.

Compilable rules

Compiling a rulebase will mainly split the rulebase in two disjoint sets of rules:

- *compilable rules* are very simple rules that, when used in a rewriting, *do not change the structure of the query*. This is the case for the two rules in our example.
- *noncompilable rules*: all the other rules

Building a Compilation in `py4graal`

The two examples below show how to build a `Compilation` instance from a rulebase, and extract from it the two rulebases `compiled` and `uncompiled`.

Example 1: all rules are compilable

The example below shows that py4graal recognizes all rules of the previous example as compilable rules (the parameter `compile` of the compilation returns the compilable rules)

```
from py4graal import *
graal = Graal()
rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y)."
rulebase = graal.create_rulebase(rbstr)
```

```
compilation = rulebase.compile()

print(compilation.compiled)
```

Output:

```
[1753994612277_43] parent(X, Y) :- father(X, Y).
[R:1753994612277_44] parent(X, Y) :- mother(X, Y).
```

Example 2: Mix of compilable and non compilable rules

Of course, not all rules are compilable, as shown by the next example.

```
from py4graal import *
graal = Graal()
rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y). parent(X, Y),
human(Y) :- human(X)."
rulebase = graal.create_rulebase(rbstr)
compilation = rulebase.compile()
print(compilation.uncompiled)
```

Output:

```
[R:1753994746237_49] parent(X, Y), human(Y) :- human(X).
```

Notes :

- Compilation not only splits the rules into compilable and non-compilable sets, it also builds an internal index structure that speeds up reasoning steps involving the compilable rules.
- With `CompilationType.HIERARCHICAL` (the default parameter), rules of the form `p(X1, ..., Xk) :- p(X1, ..., Xk).` are always treated as compilable. These usually encode class hierarchies or taxonomies.
- `With CompilationType.ID`, the compiler is more aggressive and classifies additional rules as compilable (for instance, it admits reordering of the body variables). This often produces smaller rewritten queries, but at the cost of a slower compilation step to build the optimized structure.

Rewriting a query using a `Compilation`

When using a `Compilation` to rewrite a query, only the non-compilable rules are actively applied during the rewriting process.

The compilable rules are not discardedt, they are treated as *background knowledge*.

For example, if a `Compilation` contains a compilable rule `parent(X,Y) :- father(X,Y).`, then during rewriting a `parent` atom in the query is implicitly understood to unify with a `father` atom in the head of a rule, even though the rule itself is not explicitly used at this stage.

Example 1 (continued): all rules are compilable

Let's revisit our initial example, this time performing a compiled rewriting of the query *"does Socrates have great-grandparents"* using a `Compilation`:

```
from py4graal import *
graal = Graal()

rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y)."
rulebase = graal.create_rulebase(rbstr)
compilation = rulebase.compile()

query = graal.create_query("?() :- parent(X, Y), parent(Y, Z), parent(Z,
socrates).")
query.compiled_rewrite(compilation)
print(query)
```

Output:

```
?() :- parent(X, Y), parent(Y, Z), parent(Z, socrates).
```

Here, all rules are compilable, so the compiled rewriting leaves the query unchanged, instead of generating the 27 queries of the standard rewriting.

Example 2 (continued): Mix of compilable and non compilable rules

We extend example 2 by considering the query *"search all great-grandparents of a human"*, and rewrite it using the compiled rulebase.

```
from py4graal import *
graal = Graal()
rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y). parent(X, Y),
human(Y) :- human(X)."
rulebase = graal.create_rulebase(rbstr)
compilation = rulebase.compile()

qstr = "?(X) :- parent(X,Y), parent(Y, Z), parent(Z, A), human(A)."
query = graal.create_query(qstr)
print(query.compiled_rewrite(compilation))
```

Output:

```
?(X) :- parent(X, Y), human(Y).
?(X) :- human(X).
?(X) :- parent(X, Y), parent(Y, Z), parent(Z, A), human(A).
?(X) :- parent(Y, Z), parent(X, Y), human(Z).
```

Only the uncompiled rule has been actively used for the rewriting, allowing to generate a rewriting of size 4. Please note that, without compilation, the size of the rewriting would have been 3+1+27+9 = 40.

Evaluating the obtained rewritten query

Let us consider again the rewritten query `?() :- parent(X, Y), parent(Y, Z), parent(Z, socrates).` generated in **example 1 (continued)**. Let us now consider the factbase `father(sophroniscus, socrates), mother(X, sophroniscus), father(Y, X).` We can see that there is no answer to our rewritten query in the factbase, unless we use the rules in some way.

A key result from Mélanie König's thesis is that, at this step, it is *only necessary to use compiled rules* to obtain all desired results. Concretely, if we consider a `factbase`, a `rulebase` and its `compilation`, and a `query`, the following five evaluation methods are guaranteed to produce **exactly the same answers**:

- 1. `query.evaluate(factbase.saturate(rulebase))` the **saturation** approach is discussed in ex3 Rules Saturation.
- 2. `query.rewrite(rulebase).evaluate(factbase)` the **rewriting** approach is the method discussed in ex4 Query Rewriting.
- 3. `queryompiled_rewrite(compilation).evaluate(factbase.saturate(compilation.compiled))` the **compiled rewriting + saturation with compiled** approach.
- 4. `query.compiled_rewrite(compilation).rewrite(compilation.compiled).evaluate(factbase)` the **compiled rewriting + rewriting with compiled** approach.
- 5. `query.compiled_rewrite(compilation).evaluate(factbase, compilation)` the **compiled rewriting + evaluation with compiled** approach. This is a planned optimization in Integraal, and will soon be available in Py4Graal.

Discussion about the three "new" evaluation approaches

Thanks to the special nature of compilable rules, both saturation and rewriting are guaranteed to terminate when applied to them. Thus termination of the two available new approaches only depends upon the termination of the compiled rewriting. So why choose one instead of the other?

compiled rewriting + rewriting with compiled

At first sight, using `query.compiled_rewrite(compilation).rewrite(compilation.compiled)` is just a more complicated way to write `query.rewrite(rulebase)`, since both produce exactly the same queries. However, the compilation version does it *faster*, and requires *less memory*, as shown by Mélanie König's experiments.

compiled rewriting + saturation with compiled

One might want to avoid the exponential blowup induced by rewriting again (with the compilable rules) the rewritten query. This can be a huge source of inefficiency when using an ontology where lots of rules express

the `subClassOf` or `subPredicateOf` properties. In that case, it can be useful to trade the exponential blowup of the query with a *linear* blowup of the factbase (a consequence of saturating with hierarchical rules).

compiled rewriting + evaluation with compiled

Sadly not yet an option. Would avoid both the exponential blowup of the query and the linear blowup of the factbase. The impact on the query evaluation speed is still to be studied, but we are kind of confident:

- the evaluation would run on a smaller factbase
- the optimized index structure should mitigate the evaluation cost of the more complex atomic queries.

Finalizing the examples

As a conclusion, let us finalize the two ongoing examples by summarizing all possible steps to evaluate a query in a factbase, given the compilation of a rulebase.

Example 1 (continued) compiled rewriting + saturation with compiled

```
from py4graal import *
graal = Graal()

rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y)."
compilation = graal.create_rulebase(rbstr).compile()

qstr = "?() :- parent(X, Y), parent(Y, Z), parent(Z, socrates)."
query = graal.create_query(qstr).compiled_rewrite(compilation)

fbstr = "father(sophroniscus, socrates), mother(X, sophroniscus), father(Y, X)."
factbase = graal.create_factbase(fbstr)
print(list(query.evaluate(factbase.saturate(compilation.compiled))))
```

Output:

```
[{}]
```

Example 2 (continued) compiled rewriting + rewriting with compiled

```
from py4graal import *
graal = Graal()
rbstr = "parent(X, Y) :- father(X, Y). parent(X, Y) :- mother(X, Y). parent(X, Y),
human(Y) :- human(X)."
rulebase = graal.create_rulebase(rbstr)
compilation = graal.create_rulebase(rbstr).compile()

qstr = "?(X) :- parent(X,Y), parent(Y, Z), parent(Z, A), human(A)."
query = graal.create_query(qstr).compiled_rewrite(compilation)
```

```
fbstr = "human(socrates), father(sophroniscus, socrates)."
factbase = graal.create_factbase(fbstr)

print(list(query.compiled_rewrite(compilation).rewrite(compilation.compiled).evalu
ate(factbase)))
```

Output:

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