rules in practical applications. RIF also supports the combined use of rules, OWL, and RDF. While the case of OWL is closely related to our discussion in Section 6.3, the specification of a combined semantics for RDF and rules requires us to encode RDF entailments in first-order logic. This last aspect, which is interesting in its own right, is found in Section 6.4.6 and can be read separately. We close the chapter with a short summary, exercises, and notes on further reading.

6.1 What Is a Rule?

It has been mentioned that rules of any type should consist at least of a premise and a conclusion, with the intuitive meaning that in any situation where the premise applies the conclusion must also hold. Such a general description obviously comprises some, if not all, OWL axioms. Consider, e.g., the "rule" that, if a person is the author of a book then she is a (member of the class) book author. This can surely be expressed in OWL DL: using the more concise description logic syntax introduced in Chapter 5 we could write

$$\texttt{Person} \sqcap \exists \texttt{authorOf}.\texttt{Book} \sqsubseteq \texttt{Bookauthor}.$$

It has also been explained in Section 5.2.2 that OWL DL can be considered as a sublanguage of first-order predicate logic. Using this knowledge, we can equivalently write the above statement as a predicate logic formula:

$$\forall x.(\texttt{Person}(x) \wedge \exists y.(\texttt{authorOf}(x,y) \wedge \texttt{Book}(y)) \rightarrow \texttt{Bookauthor}(x)).$$

Using standard semantic equivalences of predicate logic, we can make the existential quantifier disappear (exercise: how exactly?):

$$\forall x \forall y.(\texttt{Person}(x) \wedge \texttt{authorOf}(x,y) \wedge \texttt{Book}(y) \rightarrow \texttt{Bookauthor}(x)).$$

This formula is a logical implication with universally quantified variables, hence it comes close to our vague idea of a "rule." The universal quantifiers express the fact that the implication is applicable to all individuals that satisfy the premise. So could we indeed consider "predicate logic rules" to

mean simply predicate logic formulae that are implications? It turns out that this alone would not say much, simply because every predicate logic formula can be rewritten to fit that syntactic form (exercise: how?). Yet, using the term "rule" as a synonym for "first-order implication" has become common practice in connection with the Semantic Web, as witnessed by formalisms such as the *Semantic Web Rule Language*, *Description Logic Rules*, *DL-safe Rules*, and the *Rule Interchange Format* (RIF-Core), all of which essentially comprise certain kinds of first-order implications. These approaches further restrict us to particular kinds of implications that no longer encompass all possible formulae of predicate logic, and which are thus interesting in their own right. Indeed, it turns out that many such rule languages have expressive and computational properties that distinguish them from first-order logic, thus making them adequate for different application scenarios.

Before going into the details of the Semantic Web rule languages mentioned above, it should be noted that there are a number of rather different interpretations of the term "rule" outside of first-order logic. Among the most popular rule formalisms in Computer Science is certainly logic programming, which is closely associated with the Prolog programming language and its various derivatives and extensions. At first glance, Prolog rules appear to be very similar to first-order logic implications that merely use a slightly different syntax, putting the precondition to the right of the rule. The example above would read as follows in Prolog:

$$\texttt{Bookauthor}(X) \texttt{ :- } \texttt{Person}(X), \texttt{authorOf}(X,Y), \texttt{Book}(Y).$$

Basic Prolog indeed has the same expressivity as first-order logic, and can equivalently be interpreted under a predicate logic semantics. But there are many extensions of Prolog that introduce features beyond first-order logic, such as operational plug-ins (e.g., for arithmetic functions) and so-called *non-monotonic* inferences which derive new results from the fact that something else can *not* be derived. Logic programming in this form, as the name suggests, has been conceived as a way of specifying and controlling powerful computations, and not as an ontology language for direct interchange on the Web. Two ontologies from different sources can usually be merged simply by taking the union of their axioms (meaningful or not), whereas two independent Prolog programs can hardly be combined without carefully checking manually that the result is still a program that can be successfully executed by the employed logic programming engine. The use of logic programming in combination with ontologies can still be quite useful, but the research that has been conducted in this field is beyond the scope of this book.

Yet another kind of rules that is very relevant in practice is known as *production rules*, such as *Event Condition Action Rules* or *business rules*. Rule languages of this type apply a more *operational* interpretation of rules, i.e.

they view rules as program statements that can be *executed* actively. For ontology languages like OWL, the semantics of an ontology is not affected by the order in which ontological axioms are considered. In contrast, for rules with an operational semantics it can be crucial to know which rule is executed first, and part of the semantics of production rules is concerned with the question of precedence between rules. Many different kinds of production rule engines are used in practice and many rule engines implement their own customized semantic interpretations of rules that do not follow a shared published semantics. As such, production rules again are hard to interchange between different systems, and the ongoing work on the W3C Rule Interchange Format is among the first efforts to allow for the kind of interoperability that a common semantic standard can offer. Yet it is currently unclear how production rule engines should best be combined with ontology-based systems, and we shall not pursue this endeavor in the remainder of this book.

Besides the interpretation of "rule" in these diverse approaches, the term can also have an even more general meaning in the context of (onto)logical knowledge representation. In particular, a "deduction rule" or "rule of inference" is sometimes understood as an instruction of how to derive additional conclusions from a knowledge base. In this sense, the rule is not part of the encoded knowledge, but rather a component of algorithms that are used to process this knowledge. A case in point is the deduction rules for RDF(S) that were discussed in Section 3.3. As we will see in Section 6.4.6, the essence of these rules can actually be captured by first-order logic implications, so that the distinction between deduction rules and rule-like logical formulae is blurred here. More generally, the deduction rules of virtually any calculus could be expressed as logical rules of some suitable logic. But this logic is typically required to be very expressive, making it difficult or impossible to implement general-purpose reasoners that can process the logical theory that was derived from a set of deduction rules. Since we are interested in semantic technologies that represent knowledge in a machine-processable way, the topic of this chapter is rules in the earlier sense, i.e. axioms for representing ontological knowledge in the form of a rule.

Moreover, all rule languages that we consider here can be viewed as fragments of first-order logic. This allows for a close semantic integration with both OWL and RDF, which in turn has helped to advance standardization, implementation, and practical usage of these types of rules. Additionally, simple first-order rule languages can also be extended with non-monotonic features inspired by logic programming. Discussing the various types of extensions that have been considered in this line of research is beyond this book, but the material covered herein is still useful as a basis for further reading (see Section 6.7). The W3C Rule Interchange Format, introduced in Section 6.4, is expected to support non-monotonic features at some future stage, possibly leading to a greater impact of such rules in the field of semantic technologies.

6.2 Datalog as a First-Order Rule Language

We now turn to concrete types of rules that, in essence, are implications of first-order logic. The advantage of this interpretation of rules is that the semantics of first-order predicate logic can be naturally combined with the semantics of OWL DL, since the latter can also be defined as a sublanguage of first-order logic. The rule language that we consider here is known as datalog, and was originally developed in the field of deductive databases.

6.2.1 Introduction to Datalog

In a nutshell, a datalog rule is a logical implication that may only contain conjunctions, constant symbols, and universally quantified variables, but no disjunctions, negations, existential quantifiers, or function symbols. We always consider datalog as a sublanguage of first-order logic to which the classical semantics applies. Both syntax and semantics will be explained in more precise terms below in a fully self-contained way. Some background knowledge in first-order logic can still be handy for understanding datalog (see Appendix C).

Before going into further details, it is worth mentioning that datalog was originally developed for querying databases. Rules and queries indeed have much in common: our example rule from Section 6.1, e.g., is in fact a datalog rule which can also be interpreted as a means of *querying* a given database for all book authors:

$$\forall x \forall y.(\texttt{Person}(x) \wedge \texttt{authorOf}(x, y) \wedge \texttt{Book}(y) \rightarrow \texttt{Bookauthor}(x)).$$

In this case, one would assume information about `Person`, `authorOf`, and `Book` to be stored in a database, while `Bookauthor` is derived from this data as a "query result." It is thus always possible to regard single rules as descriptions of relevant "views" on the data. Much work on datalog is related to the use of rules in this sense, and we will return to the topic of querying later in Chapter 7.

When considering datalog as a rule language, however, we also want to allow rules to be applied *recursively*. This means that the result of a rule can again be used by other rules to derive further conclusions, continuing until no further conclusions can be obtained from any rule. This use of recursion has been an important topic in the area of *deductive databases* as well, and semantic technologies can build on the results that were obtained in this field.

Further references on deductive databases are given at the end of this chapter.[2]

Let us now consider the syntax of datalog rules and the intuitive semantics of such rules. Besides logical operators, a datalog rule can feature three kinds of symbols:

- *Constant symbols* are used as names to refer to certain elements of the domain of interest.
- *Variables* are used as place holders for (arbitrary) domain elements to which rules might apply.
- *Predicate symbols*, or simply *predicates*, are used to denote relations between domain elements.

Constant symbols play essentially the role of individual names in OWL or description logics, as explained in Chapters 4 and 5. Predicate symbols may take an arbitrary number of arguments: predicates with one argument are similar to OWL classes, just like `Person` in our earlier example; predicates with two arguments resemble OWL property names, as in the case of `authorOf` above. But datalog also allows for predicates that have three or more, or even zero arguments. It is usually assumed that the number of arguments for each predicate symbol is fixed, and this number is then called the *arity* of this predicate symbol.

Summing up, the syntax of datalog depends on three sets of symbols: a set C of constant symbols, a set V of variable symbols, and a set P of predicate symbols each of which has a fixed natural number as its arity. Together, the sets (C, V, P) are called a *signature* of datalog, and every set of datalog rules is based on some such signature. The sets C and P are usually assumed to be finite, containing only the symbols required for an application. In contrast, one often assumes that there is an arbitrary supply of variables, i.e. that the set V is (countably) infinite. It is common to denote variables by letters x, y, and z, possibly with subscripts.

Now given such a signature we can build datalog rules as follows:

- A datalog *term* is a constant symbol or a variable.
- A datalog *atom* is a formula of the form $p(t_1, \ldots, t_n)$ given that $p \in P$ is a predicate of arity n, and $t_1, \ldots, t_n$ are terms.

[2] A notable difference to our treatment is that many database-related applications define datalog based on a logic programming semantics or with certain "closure axioms." This is useful for achieving a *closed-world* semantics that is desirable for a database: if a fact is not in the database, it should be concluded that it is false. Such *non-monotonic* behavior, however, is only obtained when extending datalog with further features, especially with non-monotonic negation. We do not consider any form of non-monotonicity in this chapter. For plain datalog, our definitions lead to exactly the same deductions as the closed-world approach. See [AHV94, Chapter 12] for a discussion and comparison of both approaches.

(1) $\texttt{Vegetarian}(x) \wedge \texttt{FishProduct}(y) \rightarrow \texttt{dislikes}(x, y)$
(2) $\texttt{orderedDish}(x, y) \wedge \texttt{dislikes}(x, y) \rightarrow \texttt{Unhappy}(x)$
(3) $\texttt{orderedDish}(x, y) \rightarrow \texttt{Dish}(y)$
(4) $\texttt{dislikes}(x, z) \wedge \texttt{Dish}(y) \wedge \texttt{contains}(y, z) \rightarrow \texttt{dislikes}(x, y)$
(5) $\texttt{Happy}(x) \wedge \texttt{Unhappy}(x) \rightarrow$
(6) $\rightarrow \texttt{Vegetarian}(\texttt{zarkus})$

FIGURE 6.1: Example datalog program

- A *datalog rule* is a formula of the form

$$\forall x_1 \ldots \forall x_m . (B_1 \wedge \ldots \wedge B_k \rightarrow H),$$

where $B_1, \ldots, B_k$ and H are datalog atoms, and $x_1, \ldots, x_m$ are exactly the variables that occur within these atoms.

Since all variables in datalog are always universally quantified at the level of rules, it is common to omit the $\forall$ quantifiers from datalog rules. We adopt this simplification for the rest of this chapter. The premise of a datalog rule is called the *rule body* while the conclusion is called the *rule head*. A set of datalog rules is sometimes called a *datalog program* which hints at the relationship to logic programming.

Figure 6.1 gives an example of a datalog program based on a datalog signature with set of constant symbols $C = \{\texttt{zarkus}\}$ and set of predicate symbols $P = \{\texttt{Dish}, \texttt{Vegetarian}, \texttt{FishProduct}, \texttt{Happy}, \texttt{Unhappy}, \texttt{dislikes}, \texttt{orderedDish}\}$. Adopting the convention introduced for OWL, we use capital letters for predicates of arity 1 ("class names"); the other predicates are all of arity 2. It is not hard to read the intended meaning from such a set of datalog rules:

(1) "Every vegetarian dislikes all fish products."[3]

(2) "Anyone who ordered a dish that he or she dislikes is unhappy." This rule shows that not all variables occurring in a rule body need to appear in the rule head.

(3) "Everything that can be ordered as a dish actually is a dish."

(4) "If someone dislikes something that is contained in a certain dish, then this person will also dislike the whole dish."

[3] Some "pesco-vegetarians" might disagree. We follow the historic definition of the Vegetarian Society here.

(5) "Markus is a vegetarian." Empty rule bodies impose no conditions, i.e. they are always true. This is the reason why a rule that consists only of a rule head is also called a *fact*. The implication arrow is sometimes omitted in this case.

(6) "Nobody can be happy and unhappy at the same time." Empty rule heads cannot be concluded, i.e. they are always false. Hence a rule without a head describes a condition that must never occur, and such rules therefore are sometimes called *integrity constraints*.

Note that some of the rules might be more widely applicable than desired. For example, rule (2) does not require that it was a person who ordered the dish. In practice, one might add further preconditions to ensure that such implicit assumptions do really hold. For the purposes of this book, however, we prefer a more simple formalization over a more correct one.

This example also illustrates that rules can often be read and understood rather easily, which is one reason why they might sometimes be preferred over other types of ontological axioms. Yet we must be wary when dealing with rules: while the intention of a single rule can seem obvious, there are still many possibly unexpected conclusions that can be drawn from a set of rules. In particular, we must be aware that rules in first-order logic "work in both directions": if a rule body is true then the rule head must of course also be true, but conversely, if a rule head is false, then the rule body must also be false. In logic, this inverse reading of a rule is known as the *contrapositive* of the implication; it is well-known that both forms $p \rightarrow q$ and $\neg q \rightarrow \neg p$ are logically equivalent. Assume, e.g., that the following facts are added to the program of Fig. 6.1 (we assume that the new constant symbols have been added to the signature):

```
Happy(markus)
orderedDish(markus,crêpeSuzette)
FishProduct(worcestershireSauce)
```

With these additional assertions, we might (rightly) conclude that Crêpe Suzette does *not* contain Worcestershire Sauce: Since Markus is happy, he cannot be unhappy (6), and hence he did not order any dish he dislikes (2). Thus, since he ordered Crêpe Suzette, Markus does not dislike this dish. On the other hand, as a vegetarian (5) Markus dislikes Worcestershire Sauce on account of it being a fish product (1). Thus, since Crêpe Suzette is a dish (3), and since Markus does not dislike it, rule (4) ensures us that the crêpe does not contain any Worcestershire Sauce.

Conclusions like the one we have just drawn are often not obvious, and as soon as we deal with larger datalog programs, we certainly would like to leave it to the computer to draw such conclusions for us. To make this possible, we

first need to specify the problem more precisely: which conclusions exactly do we expect the computer to draw? This is the right time to introduce the formal semantics of datalog.

6.2.2 Semantics of Datalog

As mentioned in the previous section, we consider datalog as a sublanguage of first-order logic, and its formal semantics is already determined by this fact. In this section, we give an alternative self-contained presentation of the datalog semantics, which can be slightly simplified due to the fact that function symbols and various first-order logical operators do not need to be addressed. This section can safely be skipped by readers who are familiar with first-order logic or who are content with the intuitive understanding established so far.

As in Chapters 3 and 5, the semantics of datalog is *model-theoretic*, i.e. it is based on defining which "models" a datalog program has. A correct conclusion from a datalog program then is any formula that is satisfied by all models of this program. As usual, a model is a special kind of *interpretation*, one that makes a given datalog program true. Hence we first explain what a datalog interpretation is and what it means for it to satisfy some datalog rule.

A *datalog interpretation* $\mathcal{I}$ consists of an interpretation domain $\Delta^\mathcal{I}$ and an interpretation function $\cdot^\mathcal{I}$. The domain is an arbitrary set that defines the (abstract) world within which all symbols are interpreted, while the interpretation function establishes the mapping from symbols into this domain:

- If a is a constant, then $a^\mathcal{I} \in \Delta^\mathcal{I}$, i.e. a is interpreted as an element of the domain.
- If p is a predicate symbol of arity n, then $p^\mathcal{I} \subseteq (\Delta^\mathcal{I})^n$, i.e. p is interpreted as an n-ary relation over the domain (the *predicate extension*).

In contrast to RDFS and OWL, datalog also contains variables and we must account for the fact that these can take arbitrary values from the domain, even for a fixed interpretation. Therefore they obtain their values from a *variable assignment*. A variable assignment $\mathcal{Z}$ for a datalog interpretation $\mathcal{I}$ is simply a function from the set V of variables to the interpretation domain $\Delta^\mathcal{I}$. For an arbitrary term t we write $t^{\mathcal{I},\mathcal{Z}}$ to mean $t^\mathcal{I}$ if t is a constant, and $\mathcal{Z}(t)$ if t is a variable. With these additional tools, we can define a truth value – *true* or *false* – of a formula by extending $\cdot^\mathcal{I}$:

- For a datalog atom $p(t_1,\ldots,t_n)$, we set $p(t_1,\ldots,t_n)^{\mathcal{I},\mathcal{Z}} = \mathit{true}$ if we find that $(t_1^{\mathcal{I},\mathcal{Z}},\ldots,t_n^{\mathcal{I},\mathcal{Z}}) \in p^\mathcal{I}$. We set $p(t_1,\ldots,t_n)^{\mathcal{I},\mathcal{Z}} = \mathit{false}$ otherwise.
- For a conjunction $B_1 \wedge \ldots \wedge B_n$ of datalog atoms $B_1,\ldots,B_n$, we set $(B_1 \wedge \ldots \wedge B_n)^{\mathcal{I},\mathcal{Z}} = \mathit{true}$ if $B_i^{\mathcal{I},\mathcal{Z}} = \mathit{true}$ for all $i = 1,\ldots,n$. We set $(B_1 \wedge \ldots \wedge B_n)^{\mathcal{I},\mathcal{Z}} = \mathit{false}$ otherwise.

$$\begin{aligned}
\texttt{Vegetarian}^{\mathcal{I}} &= \texttt{Happy}^{\mathcal{I}} = \{\texttt{markus}\} \\
\texttt{FishProduct}^{\mathcal{I}} &= \{\texttt{worcestershireSauce}\} \\
\texttt{dislikes}^{\mathcal{I}} &= \{(\texttt{markus}, \texttt{worcestershireSauce})\} \\
\texttt{orderedDish}^{\mathcal{I}} &= \{(\texttt{markus}, \texttt{crêpeSuzette})\} \\
\texttt{Dish}^{\mathcal{I}} &= \{\texttt{crêpeSuzette}\} \\
\texttt{Unhappy}^{\mathcal{I}} &= \texttt{contains}^{\mathcal{I}} = \emptyset
\end{aligned}$$

FIGURE 6.2: Example datalog interpretation of predicate symbols

- For a datalog rule $B \rightarrow H$, we set $(B \rightarrow H)^{\mathcal{I}} = \mathit{true}$ if, for all possible variable assignments $\mathcal{Z}$ for $\mathcal{I}$, we find that either $B^{\mathcal{I},\mathcal{Z}} = \mathit{false}$ or $H^{\mathcal{I},\mathcal{Z}} = \mathit{true}$. We set $(B \rightarrow H)^{\mathcal{I}} = \mathit{false}$ otherwise. Note that B can be an arbitrary conjunction of datalog atoms in this case.
- For a datalog fact $\rightarrow H$, we set $(\rightarrow H)^{\mathcal{I}} = \mathit{true}$ if, for all possible variable assignments $\mathcal{Z}$ for $\mathcal{I}$, we find that $H^{\mathcal{I},\mathcal{Z}} = \mathit{true}$. Otherwise, we set $(\rightarrow H)^{\mathcal{I}} = \mathit{false}$.
- For a datalog integrity constraint $B \rightarrow$, we set $(B \rightarrow)^{\mathcal{I}} = \mathit{true}$ if, for all possible variable assignments $\mathcal{Z}$ for $\mathcal{I}$, we find that $B^{\mathcal{I},\mathcal{Z}} = \mathit{false}$. Otherwise, we set $(B \rightarrow)^{\mathcal{I}} = \mathit{false}$. Note that B can be an arbitrary conjunction of datalog atoms in this case.

Note that the truth of a rule does not depend on a particular variable assignment, since the (implicit) universal quantifiers bind all variables in all rules. If an interpretation $\mathcal{I}$ maps a datalog rule to *true*, then we say that $\mathcal{I}$ *satisfies* this rule. If $\mathcal{I}$ satisfies all rules of a datalog program, then we say that $\mathcal{I}$ satisfies the program, or that $\mathcal{I}$ is a *model* of that program. A datalog rule is a *conclusion* of a datalog program if the rule is satisfied by all models of the program. Observe that the last sentence includes all types of rules, so in particular it defines in which cases a certain fact is entailed by a datalog program. The entailment of facts is by far the most common reasoning problem for datalog, and many implementations are specifically tailored toward the derivation of facts.

The above finishes the formal definition of the datalog semantics. To illustrate the definitions, we describe a particularly interesting model for the example in Section 6.2.1 (Fig. 6.1 and the related facts on page 219). As a domain of interpretation, we pick the set of constant symbols of the given signature, i.e. $\Delta^{\mathcal{I}} = \{\texttt{markus}, \texttt{crêpeSuzette}, \texttt{worcestershireSauce}\}$. Next, we need to define the mapping $\cdot^{\mathcal{I}}$. On constant symbols, this is very easy to do: we just map every constant symbol to itself, e.g., $\texttt{markus}^{\mathcal{I}} = \texttt{markus}$. The interpretations of the predicate symbols are given in Fig. 6.2. It is straightforward to check that this interpretation is indeed a model for the datalog program we consider. For plain datalog programs that are consistent, it is always possible to construct models in this particularly simple fashion by just taking the constant symbols as interpretation domains, and such models are known as *Herbrand models*.[4] Moreover, it is always possible to find a model that satisfies as few datalog atoms as possible, such that no other model satisfies fewer datalog facts. The existence of such *least Herbrand models* is of great significance and can be exploited for practical implementations – but for our purposes it is enough to note that this feature provides us with a conveniently small example model. Unfortunately, this nice property is lost as soon as we introduce OWL or RDF into the picture.

6.3 Combining Rules with OWL DL

Datalog gives us a basic mechanism for specifying knowledge using rules. Our introduction to datalog so far has intentionally left out technical issues such as a concrete machine-readable syntax for storing and exchanging datalog rules but these details could clearly be added to obtain a full-fledged modeling language that could be used on the Semantic Web. Indeed, the Rule Interchange Format introduced in Section 6.4 achieves this to a certain extent.

The paradigm of rule-based modeling is quite different from the ontological modeling that was introduced with OWL, and it is not obvious how to combine both approaches. Would such a combination of OWL and rules be meaningful at all? Would this combination actually increase the expressive power of either formalism? How difficult would it be to build tools that can process a combination of OWL and rules? We address these questions in this section.

6.3.1 Combined Semantics: Datalog and Description Logics

The first of our initial questions is not hard to answer: a combination of datalog and OWL DL is indeed meaningful. Both languages can be seen as sublanguages of standard first-order logic, so the combination of a datalog program with an OWL DL ontology can always be viewed as a collection of first-order logic formulae with the usual first-order semantics. So, at least conceptually, there are no major problems.[5]

[4] After the French mathematician *Jacques Herbrand*, pronounced /ɛrbrã/ with *H* silent.
[5] The situation for OWL Full is not as clear. A possible combined semantics is proposed in the OWL compatibility document of the Rule Interchange Format; see Section 6.4 for some discussion.

ways in which rules can be applied, especially if they have a great number of variables in their body.

Do these results imply that the combined complexity of OWL DL and datalog is also not harder than NExpTime, the larger of the two individual complexities? The answer is a resounding no. Complexities can in general not be combined in such a naive way, and, in fact, typical reasoning tasks for the combination of datalog and $\mathcal{SHOIN}(D)$ turn out to be *undecidable*. Moreover, this is the case even for much simpler description logics such as $\mathcal{ALC}$. This result might be somewhat disappointing since it assures us that it is impossible to ever devise a software tool that can compute *all* conclusions from *all* possible knowledge bases that consist of a description logic part and a datalog part. But this formulation also hints at two ways of escaping this problem. As a first option, one might be content with a tool that draws at least *some* conclusions which are certain, i.e. an inferencing program that is sound but incomplete. Alternatively, one could try to find reasoning methods that are sound and complete, but that cannot be applied to all possible knowledge bases. In the next sections, we explore these options for two cases that restrict the expressivity of datalog rules to recover decidability: *Description Logic Rules* and *DL-safe Rules*.

6.3.3 Description Logic Rules

We have already noted in the introductory Section 6.1 that some description logic axioms can also be presented as (datalog) rules, and, equivalently, certain datalog rules can be cast into description logic axioms with the same meaning. It is clear that there must still be rules and axioms that cannot be rewritten in this way, or at least that it is not possible to do this rewriting automatically. Otherwise, one could use a rewriting algorithm followed by a standard reasoning algorithm for datalog or description logics, respectively, to obtain a decision procedure for the combined reasoning tasks. Such a procedure cannot exist according to the undecidability result mentioned in the previous section.

In this section, we address the question which datalog rules can be directly represented as description logic axioms, thus deserving the name *Description Logic Rules*. We shall see that the highly expressive description logic $\mathcal{SROIQ}$ (the basis for OWL 2 DL) can express significantly more rules than the description logic $\mathcal{SHOIN}$ (the basis for OWL DL). A comprehensive algorithm for transforming rules into description logic axioms is then provided in Fig. 6.5.

Let us first consider some examples to improve our intuition. The following rule appeared within the introductory section:

$$\texttt{Person}(x) \wedge \texttt{authorOf}(x, y) \wedge \texttt{Book}(y) \rightarrow \texttt{Bookauthor}(x).$$

We noted that it can equivalently be expressed by the description logic axiom $\texttt{Person} \sqcap \exists \texttt{authorOf}.\texttt{Book} \sqsubseteq \texttt{Bookauthor}$. The important difference between both representations is that the latter does not use any variables. So where did the variables go? We have learned in previous sections that class descriptions resemble unary predicates of first-order logic. It is not necessary to state the argument of these unary predicates since it is always the same variable on both sides of a class inclusion axiom. In the above case, e.g., we could mix datalog and description logic syntax to write:

$$(\texttt{Person} \sqcap \exists \texttt{authorOf}.\texttt{Book})(x) \rightarrow \texttt{Bookauthor}(x).$$

This explains the whereabouts of variable x. The variable y in turn appears only in two positions in the rule body. Since it is not referred to in any other part of the rule, it suffices to state that there exists some object with the required relationship to x, so the rule atoms $\texttt{authorOf}(x, y) \wedge \texttt{Book}(y)$ are transformed into $\exists \texttt{authorOf}.\texttt{Book}(x)$. Rewriting atoms as description logic class expressions in this fashion is sometimes called *rolling-up*, since a "branch" of the rule body is rolled-up into a statement about its first variable. This terminology will become more intuitive in light of a graphical representation that we explain below.

We can try to generalize from this example. We have seen that x in the above case is simply an implicit (and necessary) part of the class inclusion axiom. So for any rule that we wish to rewrite as such an axiom, we need to identify some variable x which plays this special role, and find a way to eliminate all other variables from the rule using a rolling-up method as above. This is not always possible, as rule (2) from Fig. 6.1 illustrates:

$$\texttt{orderedDish}(x, y) \wedge \texttt{dislikes}(x, y) \rightarrow \texttt{Unhappy}(x).$$

The conclusion of this rule suggests that the variable y should be eliminated to obtain a class inclusion axiom. But the premise of the rule cannot be rewritten as above. A class expression like $\exists \texttt{orderedDish}.\top \sqcap \exists \texttt{dislikes}.\top$ describes elements with relationships orderedDish and dislikes, but not necessarily to the same element y. Using inverse roles, one could also write $\exists \texttt{orderedDish}.\exists \texttt{dislikes}^-.\top$ to describe some x who ordered something that is disliked by someone – but not necessarily by x. It turns out that there is no way to directly express this relationship in any of the major description

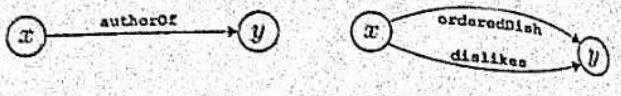

FIGURE 6.4: Examples of simple rule dependency graphs

logics considered in this book.[6] We conclude that rolling-up is only possible if a variable is "reachable" by only a single binary predicate.

We now give a more precise characterization of the rules that can be rewritten as description logic axioms. In order to understand in which cases we can use the rolling-up method, the key is to consider the *dependency graph* of the rule premise. This graph is obtained from the premise by simply taking variables as nodes, and binary predicates as edges between variables. Note that (atoms with) constant symbols do not play a role in this definition; it will be discussed further below why this is desirable. Figure 6.4 shows the dependency graphs of the above example rules, with labels indicating the relationship to binary predicates.

With this visualization in mind, we can speak about "paths" within a rule premise. Intuitively, a path between two nodes is simply a set of edges leading from one node to the other, where we do not care about the direction of the edges. More formally, we can describe a path in some rule premise B as follows:

- if $R(x,y)$ is an atom in B, then $\{R(x,y)\}$ is a path between x and y,
- if p is a path between x and y, then p is also a path between y and x,
- if p is a path between x and y, q is a path between y and z, and no atom occurs both in p and in q, then $p \cup q$ is a path between x and z,

where x, y, and z are all variables. The set $\{\texttt{authorOf}(x,y)\}$, e.g., is a path between x and y, and this is the only path in the first example rule given in this section. In the second example rule, we find the obvious paths of length one, but also the path $\{\texttt{orderedDish}(x,y), \texttt{dislikes}(x,y)\}$ which can be viewed as a path from x to x, or as a path from y to y. Looking at Fig. 6.4, we recognize that paths are really just sets of edges that we can use to get from one node to another. Observe that as a result of defining paths as sets, we are not allowed to use any edge more than once in a single path.

Now a datalog rule can be transformed into a semantically equivalent set of axioms of the description logic $\mathcal{SROIQ}$ if the following conditions hold:

- The rule contains only unary and binary predicates.

[6] The required description logic feature in this case is the *conjunction of roles*; see [RKH08].

- For any two variables x and y, there is at most a single path between x and y in the premise of the rule.

We call such datalog rules *Description Logic Rules*, or *DL Rules* for short. Before providing the complete transformation algorithm for Description Logic Rules in Fig. 6.5, we highlight some important cases that the definition allows.

The second item in the above definition is tantamount to the statement that the rule premise's dependency graph contains no (undirected) cycles of length greater than 1. A cycle of length 1 is an atom of the form $R(x,x)$ - a special case that we can address in $\mathcal{SROIQ}$. Indeed, whenever we encounter an atom of the form $R(x,x)$, we can introduce a new class name C_R which we define with an axiom $C_R \equiv \exists R.\mathrm{Self}$. One can then simply replace any atom $R(x,x)$ by $C_R(x)$, and this does not change the conclusions that can be drawn from the knowledge base, as long as we are only interested in conclusions that do not refer to the new class name C_R.

The attentive reader will already have noticed that our above definition admits further types of rules that we did not consider yet. An example is rule (4) of Fig. 6.1: it contains only unary and binary predicates, and its dependency graph has no loops. Yet its conclusion is a binary atom, and hence can certainly not be expressed as a class inclusion axiom. $\mathcal{SROIQ}$ offers two basic forms of role inclusion axioms that we may try to use:

$$R \sqsubseteq S \qquad \text{and} \qquad R_1 \circ \ldots \circ R_n \sqsubseteq S,$$

where the first can be considered as a special case of the second role composition. But both of these axioms can include only role names, while rule (4) also contains a unary (class) atom $\texttt{Dish}(y)$. As in the above case of rule atoms $R(x,x)$, this problem can be addressed by adding an auxiliary axiom to the knowledge base. This time, a new role name $R_{\texttt{Dish}}$ is introduced together with the class inclusion axiom $\texttt{Dish} \equiv \exists R_{\texttt{Dish}}.\mathrm{Self}$. Intuitively speaking, this defines the class of dishes to be equivalent to the class of those things which have the relationship $R_{\texttt{Dish}}$ to themselves. With this additional axiom, one can rewrite rule (4) as follows:

$$\texttt{dislikes}(x,z) \wedge R_{\texttt{Dish}}(y,y) \wedge \texttt{contains}(y,z) \rightarrow \texttt{dislikes}(x,y).$$

This step is the core of the transformation to $\mathcal{SROIQ}$. Using inverse roles, we can now write the rule premise as a chain:

$$\texttt{dislikes}(x,z) \wedge \texttt{contains}^-(z,y) \wedge R_{\texttt{Dish}}(y,y) \rightarrow \texttt{dislikes}(x,y).$$

This rule can now easily be expressed as a $\mathcal{SROIQ}$ role composition axiom. Together with the auxiliary axiom we have used above, rule (4) is thus represented by the following description logic knowledge base:

$$\texttt{Dish} \equiv \exists R_{\texttt{Dish}}.\mathrm{Self}$$
$$\texttt{dislikes} \circ \texttt{contains}^- \circ R_{\texttt{Dish}} \sqsubseteq \texttt{dislikes}$$

Note that the second axiom no longer contains the requirement that $R_{\texttt{Dish}}$ refers to the same variable in first and second position. The resulting knowledge base therefore is not strictly semantically equivalent to the original rule. Yet, a formula that does not contain the auxiliary name $R_{\texttt{Dish}}$ is entailed by the new knowledge base exactly if it is entailed by the original rule (4). Therefore the transformed knowledge base can be used instead of the original rule for all common reasoning tasks.

While these examples provide us with a significant set of tools for translating rules into axioms, there is still a case that we have not addressed yet. Consider rule (1) of Fig. 6.1. Its dependency graph has no edges and certainly no loops, so it should be possible to transform it. Yet, even if we use the above method for replacing the unary predicates $\texttt{Vegetarian}(x)$ and $\texttt{FishProduct}(y)$ with new auxiliary roles, we only obtain the following rule:

$$R_{\texttt{Vegetarian}}(x,x) \wedge R_{\texttt{FishProduct}}(y,y) \rightarrow \texttt{dislikes}(x,y).$$

But this cannot be rewritten as a role composition axiom, since there is a "gap" between x and y. Another special feature of $\mathcal{SROIQ}$ comes to our aid: the universal role U can be added to the rule without changing the semantics:

$$R_{\texttt{Vegetarian}}(x,x) \wedge U(x,y) \wedge R_{\texttt{FishProduct}}(y,y) \rightarrow \texttt{dislikes}(x,y).$$

Since the relation denoted by U is defined to comprise all pairs of individuals, adding the atom $U(x,y)$ does not impose any restrictions on the applicability of the rule. Yet it helps us to bring the rule into the right syntactic shape for being expressed in $\mathcal{SROIQ}$. Together with the required auxiliary axioms, we thus obtain:

$$\texttt{Vegetarian} \equiv \exists R_{\texttt{Vegetarian}}.\mathrm{Self}$$
$$\texttt{FishProduct} \equiv \exists R_{\texttt{FishProduct}}.\mathrm{Self}$$
$$R_{\texttt{Vegetarian}} \circ U \circ R_{\texttt{FishProduct}} \sqsubseteq \texttt{dislikes}$$

Essentially, this discussion provides us with enough tools for treating all DL Rules in Fig. 6.1. Yet there is one more issue that deserves some attention, even if it does not occur in our example: the definition of DL Rules does not impose any restrictions on the occurrence of constant symbols, since the latter are not taken into account when defining dependency graphs. The reason why this is feasible is that we can replace individual occurrences of constant symbols by arbitrary new variables. If, for example, a rule body has the form $R(x,a) \wedge S(a,y)$, it can equivalently be rewritten as $R(x,z) \wedge \{a\}(z) \wedge S(a,y)$. Here, the new variable z is required to refer to the (unique) member of the nominal class denoted by $\{a\}$, i.e. z must refer to the individual denoted by a. Note that only one occurrence of the constant a has been replaced in this transformation. When replacing the remaining occurrence, we can introduce yet another new variable instead of using z again: $R(x,z) \wedge \{a\}(z) \wedge S(v,y) \wedge \{a\}(v)$. Clearly, this transformation cannot create any new cycles in the rule's dependency graph, so that it is indeed safe to ignore constants when defining DL Rules.

This completes our set of methods for rewriting rules. We sum up our insights in a single transformation algorithm that can be applied to any Description Logic Rule, given in Fig. 6.5. The algorithm is organized in multiple steps, each of which is meant to solve a particular problem that may occur in the shape of the input rule. All but the last step apply to rules with unary and binary head atoms alike, while the last step needs to distinguish between class inclusion axioms and role inclusion axioms. The underlying ideas of steps 2 through 6 have already been explained in the examples above – we only note that the step-wise transformation introduces some description logic syntax into the rule, and that notions like "unary atom" should be assumed to include these additional expressions.

Step 1 has been added to normalize the shape of the rule so as to reduce the cases we need to distinguish in the algorithm. This initial step uses $\top$ and $\bot$ class expressions to normalize rules with empty bodies or heads, i.e. facts and integrity constraints. Moreover, it eliminates constant symbols as discussed above.

Some further characteristics of the transformation algorithm are worth noting. First of all, the algorithm is non-deterministic since there are often multiple ways to complete a step. Step 3, e.g., allows us to pick any pair of unconnected variables to connect them. If the dependency graph consists of two unconnected parts with more than one variable in each, then we can choose any of the occurring variables to be connected. Here and in all other non-deterministic cases, our choice does not influence the correctness of the result, but it might simplify some of the later steps.

Moreover, it must be acknowledged that the transformation algorithm is by far not optimal, and often produces more complicated results than necessary. This is so, since the purpose of the algorithm is to cover all possible cases, and not to yield minimal results whenever possible. As an ex-

Input: A Description Logic Rule $B \to H$
Output: A $\mathcal{SROIQ}$ knowledge base K

Initialize $K := \emptyset$
Repeat each of the following steps until no further changes occur:

Step 1: Normalize rule

- If H is empty, then set $H := \bot(x)$, where x is an arbitrary variable.
- For each variable x in H: If x does not occur in B, then set $B := B \wedge \top(x)$.
- If possible, select a single occurrence of a constant symbol a as a parameter of some predicate of $B \to H$, and pick a variable x not occurring in $B \to H$. Then replace the selected occurrence of a with x, and set $B := B \wedge \{a\}(x)$.

Step 2: Replace reflexive binary predicates

- If possible, select a predicate $R(x,x)$ and replace $R(x,x)$ with $C_R(x)$ where C_R is a new unary predicate symbol. Set $K := K \cup \{C_R \equiv \exists R.\mathrm{Self}\}$.

Step 3: Connect rule premise

- If possible, select two (arbitrary) variables x and y such that there is no path between x and y in B. Then set $B := B \wedge U(x,y)$.

Step 4: Orient binary predicates

- Now H must be of the form $D(z)$ or $S(z,z')$ for some variables z and z'. For every binary predicate $R(x,y)$ in B:
  If the (unique) path from z to y is shorter (has fewer elements) than the path from z to x, then replace $R(x,y)$ in B by $R^-(y,x)$.

Step 5: Roll up side branches

- If B contains an atom $R(x,y)$ or $R^-(x,y)$ such that y does not occur in any other binary atom in B or H then
  - If B contains unary atoms $C_1(y),\ldots,C_n(y)$ that refer to y, then define a new description logic concept $E := C_1 \sqcap \ldots \sqcap C_n$, and delete $C_1(y),\ldots,C_n(y)$ from B. Otherwise define $E := \top$.
  - Replace $R(x,y)$ (or $R^-(x,y)$, respectively) by $\exists R.E(x)$ (or $\exists R^-.E(z)$), where E is the concept just defined.

Step 6a: If H is of the form $D(x)$: Create final class inclusion axiom

- In this case, B must be of the form $C_1(x) \wedge \ldots \wedge C_n(x)$. Set $K := K \cup \{C_1 \sqcap \ldots \sqcap C_n \sqsubseteq D\}$.

Step 6b: If H is of the form $S(x,y)$: Create final role inclusion axiom

- For each unary atom $C(z)$ in B: Replace $C(z)$ by $Rc(z,z)$ where Rc is a new role name, and set $K := K \cup \{C \equiv \exists Rc.\mathrm{Self}\}$.
- Now B contains only one unique path between x and y which is of the form $\{R_1(x,x_2), R_2(x_2,x_3), \ldots, R_n(x_n,y)\}$ (where each of the R_i might be a role name or an inverse role name). Set $K := K \cup \{R_1 \circ \ldots \circ R_n \sqsubseteq S\}$.

FIGURE 6.5: Transforming Description Logic Rules into $\mathcal{SROIQ}$ axioms

treme example, consider the simple fact $\to R(a,b)$ which could directly be expressed as a $\mathcal{SROIQ}$ ABox statement. Instead, the algorithm normalizes the rule to $\{a\}(x) \wedge \{b\}(y) \to R(x,y)$, and then connects both terms to obtain $\{a\}(x) \wedge U(x,y) \wedge \{b\}(y) \to R(x,y)$. Finally, in Step 6b, unary atoms are replaced by auxiliary binary predicates $R_{\{a\}}$ and $R_{\{b\}}$, so that we obtain the following final set of description logic axioms:

$$\{a\} \equiv \exists R_{\{a\}}.\mathrm{Self}$$
$$\{b\} \equiv \exists R_{\{b\}}.\mathrm{Self}$$
$$R_{\{a\}} \circ U \circ R_{\{b\}} \sqsubseteq R$$

While this is clearly not the preferred way of expressing this statement, it still captures the intended semantics. On the other hand, the algorithm also covers cases like $\to R(x,a)$, $C(x) \wedge D(y) \to$, or $R(x,y) \wedge S(a,z) \to T(y,z)$ where a proper transformation might be less obvious. When dealing with such transformations – for instance in the exercises later in this chapter – it is therefore left to the reader to either apply exactly the above algorithm to obtain a correct but possibly lengthy solution, or to use shortcuts for obtaining a simplified yet, hopefully, correct result.

When using DL Rules in practice, we should not forget to take into account that the description logic $\mathcal{SROIQ}$ imposes some further restrictions on its knowledge bases. Two such restrictions have been introduced in Section 5.1.4: regularity of RBoxes and simplicity of roles. To ensure decidability, the corresponding conditions must be checked for the knowledge base as a whole, and not just for single axioms. Hence, when adding DL Rules to $\mathcal{SROIQ}$ knowledge bases, we must take care not to violate any such condition. The following DL Rule, e.g., could be useful in practice:

$$\texttt{Woman}(x) \wedge \texttt{hasChild}(x,y) \to \texttt{motherOf}(x,y).$$

But if, in addition, an axiom $\texttt{motherOf} \sqsubseteq \texttt{hasChild}$ is contained in the knowledge base, then the RBox obtained after translating the DL Rule to $\mathcal{SROIQ}$ contains a cyclic dependency between motherOf and hasChild, and hence is no longer regular. Therefore, either of the two statements can be used, but they cannot be combined in one knowledge base if we want to employ common inference algorithms for reasoning. It is conceivable that the restrictions of current algorithms could be relaxed to accommodate some more of the specific axioms that are obtained from Description Logic Rules. Another possible solution is to resort to other kinds of rules, as introduced in the following section.

(1)	$\texttt{Vegetarian}(x) \wedge \texttt{FishProduct}(y) \rightarrow \texttt{dislikes}(x, y)$
(2)	$\texttt{orderedDish}(x, y) \wedge \texttt{dislikes}(x, y) \rightarrow \texttt{Unhappy}(x)$
(3)	$\texttt{orderedDish}(x, y) \rightarrow \texttt{Dish}(y)$
(4)	$\texttt{dislikes}(x, z) \wedge \texttt{Dish}(y) \wedge \texttt{contains}(y, z) \rightarrow \texttt{dislikes}(x, y)$
(5)	$\rightarrow \texttt{Vegetarian}(\texttt{markus})$
(6)	$\texttt{Happy}(x) \wedge \texttt{Unhappy}(x) \rightarrow$
(7)	$\exists \texttt{orderedDish}.\texttt{ThaiCurry}(\texttt{markus})$
(8)	$\texttt{ThaiCurry} \sqsubseteq \exists \texttt{contains}.\texttt{FishProduct}$

FIGURE 6.6: DL-safety of rules depends on the given description logic axioms

6.3.4 DL-safe Rules

The previous section considered DL Rules as a kind of datalog rules that could also be represented in description logics. The main application of DL Rules therefore is to simplify ontology editing, especially considering that some of the required encodings can be quite complex. This section, in contrast, introduces a different type of rules that add real expressivity which is not available in $\mathcal{SROIQ}$ yet. These datalog rules are called *DL-safe*, and they are based on the idea of limiting the interaction between datalog and description logics to a "safe" amount that does not endanger decidability.[7]

The restrictions that DL-safe rules impose on datalog to preserve decidability can be viewed from two perspectives. On the one hand, one can give syntactic "safety" conditions that ensure the desired behavior. This corresponds to the original definition of DL-safe rules. On the other hand, one can modify the semantics of datalog rules so as to ensure that every rule is implicitly restricted to allow only "safe" interactions with description logic knowledge bases. This approach has become very common in practice, since it is indeed always possible to evaluate arbitrary datalog rules in a DL-safe way, without requiring the user to adhere to specific syntactic restrictions. We begin with the original definition and explain the second perspective afterwards.

To define DL-safety, we need to consider a concrete description logic knowledge base K. We call a datalog atom a *DL-atom* if its predicate symbol is used as a role name or class name in K, and we call all other datalog atoms *non-DL-atoms*. Then a datalog rule $B \rightarrow H$ is *DL-safe for* K if all variables occurring in $B \rightarrow H$ also occur in a non-DL-atom in the body B. Note that, as before, we use B to abbreviate an arbitrary conjunction of datalog atoms. A set of datalog rules is DL-safe for K if all of its rules are DL-safe for K. Consider, e.g., the datalog rules and description logic axioms in Fig. 6.6 that we already know from earlier examples. The predicates `orderedDish`, `contains`, `ThaiCurry`, and `FishProduct` are used in the description logic part. Therefore, rule (1) is not DL-safe since y is used only in the DL-atom $\texttt{FishProduct}(y)$. Rule (3) is not allowed for similar reasons, but all other rules are indeed DL-safe.

DL-safety therefore is rather easy to recognize: we only need to check whether there are enough non-DL-atoms in each rule premise. Some care must still be taken since DL-safety is not an intrinsic feature that a datalog rule may have. Rather, it depends on the accompanying description logic knowledge base and the predicates used therein. To see this, we can take a different perspective on the rules of Fig. 6.6. As we have seen in Section 6.3.3, rule (1) and rules (3) to (6) could similarly be considered as Description Logic Rules, while rule (2) does not meet the requirements. Using DL Rules and DL-safe rules together is no problem since the former are merely a syntactic shortcut for description logic axioms. We just have to treat all predicates that occur in DL Rules as if they were used in the description logic part of the knowledge base. Rule (1) and rule (3), which we found not to be DL-safe above, could thus also be considered as DL Rules. But when doing so, the predicates `Dish` and `dislikes` also belong to the description logic part of the knowledge base, and thus rules (2) and (4) are no longer DL-safe.

Summing up, we can treat the rules of Fig. 6.6 in at least two ways: either we use rules (2), (4), (5), and (6) as DL-safe rules, or we use rule (1) and rules (3) to (6) as DL Rules. Neither approach is quite satisfying, since we have to neglect one or the other rule in each of the cases. But the definition of DL-safety shows us a way to get closer to our original rule set. Namely, whenever a rule is not DL-safe for a particular knowledge base, it can be modified to become DL-safe. All we have to do is to add further non-DL-atoms to the rule premise for all variables that did not appear in such an atom yet. Which further non-DL-atoms should this be? In fact, we can simply introduce a new unary non-DL-predicate O and use atoms of the form $O(x)$ to ensure the DL-safety conditions for a variable x. When viewing rules (1) and rules (3) to (6) as DL Rules, e.g., we can modify rule (2) to become DL-safe as follows:

$$(2') \quad \texttt{orderedDish}(x, y) \wedge \texttt{dislikes}(x, y) \wedge O(x) \wedge O(y) \rightarrow \texttt{Unhappy}(x)$$

This new rule is indeed DL-safe since both x and y occur in non-DL-atoms, and hence it can be used together with the other (DL) rules. But, unfortunately, this rule does not allow for any additional conclusions! The reason is that there is no information about O, and therefore we can always find an interpretation where O is interpreted as the empty set, so that rule (2') is never applicable. Adding $O(x)$ and $O(y)$ imposes additional conditions for applying the rule. Therefore we would like to ensure that O must encompass

[7] The name "DL-safe" actually originates from a related notion of "safety" that has been considered for datalog in the field of deductive databases.