Contributions to

Inductive Logic Programming

Afstudeerscriptie Bestuurlijke Informatica
R. M. de Wolf
114903

Erasmus Universiteit Rotterdam
Dr. S.-H. Nienhuys-Cheng

Mei 1996

Contents

Preface

This graduation thesis is the result of nearly two years of research into various aspects of Inductive Logic Programming (ILP), performed by my supervisor Shan-Hwei Nienhuys-Cheng and myself. ILP is a very young field of research, which can be seen as the intersection of Machine Learning and Logic Programming. Accordingly, it is concerned with learning a general theory from given examples, within the framework provided by (clausal) logic. Like so many other young research communities, ILP is characterized by a mild form of chaos. Most definitions and results are only available in widely scattered research papers. As a consequence, concepts are not always uniformly defined, often vague, and proofs are not always correct. Evidently, what ILP needs is a book collecting these concepts and results in a self-contained, unified and rigorous manner.

Near the end of the spring of 1994, just at the moment when I started worrying about a topic on which to graduate, Shan-Hwei came along and asked me to join her in writing precisely such a book. The research presented in this thesis is actually a spin-off of the research we have done for our book. This book is by now nearly finished and is to be published early next year.

The research in this thesis can be divided into three parts: deduction, specialization of theories, and least generalizations and greatest specializations of sets of clauses. These directions of research correspond to Chapters 2, 3 and 4, respectively, which form the main part of the thesis. Chapter 2 is based on [NCW95a, NCW95d, NCW95e, NCW95f, NCW96d][1], a number of articles jointly written by Shan-Hwei and myself. Chapter 3 is based on our papers [NCW95b, NCW95c, NCW96a, NCW96c], while Chapter 4 is a slightly revised version of our article [NCW96b]. These three chapters are preceded by an introductory chapter which defines and discusses the basic problem setting of ILP, gives a brief history of the field and outlines the thesis. Furthermore, in order to make the thesis a self-contained text an appendix is added, which gives the main definitions from logic that we need.

Let me finish this preface by expressing my gratitude to Shan-Hwei Nienhuys-Cheng. During the past two years, as our book slowly took shape and the above-mentioned articles were written, I have spent many hours discussing, chatting and (sometimes) arguing with her. I would like to thank her very much for introducing me to the world of scientific research, for giving me a feel of what a good proof should look like, and for our pleasant, fruitful, and very stimulating cooperation.

[1] Our paper [NCW95f] was awarded the Best Paper Award at NAIC'95.

Chapter 1

What is Inductive Logic Programming?

1.1 The importance of learning

> Him she found sweating with toil as he moved to and fro about his bellows in eager haste; for he was fashioning tripods, twenty in all, to stand around the wall of his well-builded hall, and golden wheels had he set beneath the base of each that of themselves they might enter the gathering of the gods at his wish and again return to his house, a wonder to behold.
> *Iliad*, XVIII, 372–377 (pp. 315–317 of [Hom24], second volume).

This quotation from Homer's Iliad is perhaps the first ever reference in Western literature to something like *Artificial Intelligence*: man-made (or in this case, god-made) artifacts displaying intelligent behaviour. As Thetis, Achilles' mother, enters Hephaestus' house in order to fetch her son a new armour, she finds Hephaestus constructing something we today would call *robots*. His twenty tripods are *of themselves* to serve the gathering of the gods (bring them food, etc.), whenever Hephaestus so desires.

Let us consider for a moment the kind of behaviour such a tripod should display. Obviously, it should be able to recognise Hephaestus' voice, and to extract his wishes from his words. But furthermore, when serving the gods, the tripod should "know" and act upon many requirements, such as the following:

1. If there is roasted owl for dinner, don't give any to Pallas Athena.
2. Don't come too close to Hera if Zeus has committed adultery again.
3. Stop fetching wine for Dionysus when he is too drunk.

   ...

It is clear that this list can be continued without end. Again and again, one can think of new situations that the tripod should be able to adapt to properly. It seems impossible to take all these requirements into account explicitly in the construction of the intelligent tripod. The task of "coding" each of the infinite number of requirements into the tripods may be considered too much, even for Hephaestus, certainly one of the most industrious among the Greek gods.

One solution to this problem would be to initially endow the tripod with a modest amount of general knowledge about what he should do, and to give it the ability to *learn* from the way the environment reacts to its behaviour. That is, if the tripod does something wrong, it can adjust its knowledge and its behaviour accordingly, thus avoiding making the same mistake in the future.[1] In that way, the tripod need not know everything beforehand. Instead, it can build up most of the required knowledge along the way. Thus the tripod's ability to learn would save Hephaestus a lot of trouble.

The importance of learning is not restricted to artifacts built to serve divine wishes. Also for many more earthly purposes, the need for learning can easily be seen. For instance, constructing a knowledge base for some expert system by interviewing experts and writing down the rules they give, is a very expensive and time-consuming business. It would be much easier if the expert system were able to learn its rules itself, from a number of given examples. Such *learning from examples* will be the topic of this thesis.

1.2 Inductive learning

Learning a general theory from specific examples, commonly called *induction*, has been a topic of inquiry for centuries. It is often seen as a main source of scientific knowledge. Suppose we are given a large number of patient's records from a hospital, consisting of properties of each patient, including symptoms and diseases. We want to find some general rules, concerning which symptoms indicate which diseases. The hospital's records provide *examples* from which we can find clues as to what those rules are. Consider measles, a virus disease. If every patient in the hospital who has a fever and has red spots suffers from measles, we could infer the general rule

1. "If someone has a fever and red spots, he has measles."

Moreover, if each patient with measles also has red spots, we can infer

2. "If someone has measles, he will get red spots."

These inferences are cases of induction. Note that these rules not only tell us something about the people in the hospital's records, but are in fact about *everyone*. Accordingly, they have predictive power: they can be used to make predictions about future patients with the same symptoms or diseases.

Usually when we want to learn something, we do not start from scratch: most often we already have some *background knowledge* relevant to the learning task. For instance, in the hospital records we might find a case of a patient a in an early fase of measles: he has a fever, but not yet red spots. Then the previous rule no. 1 cannot be used. Now suppose in the records we see that this person has the same address as another patient b suffering from measles. Since we know that measles is a rather contagious disease, we can infer that a

[1] Of course, for this scheme to work, we have to assume that the tripod "survives" its initial failures. If Zeus immediately smashes the tripod into pieces for bringing him white instead of red wine, the tripod won't be able to learn from its experience.

also has measles. The fact that measles is contagious, is not something that is expressed in the hospital records. Instead, it is a piece of knowledge that we already had. Nevertheless, this piece of background knowledge combined with the examples allows us to induce the general rule

> 3. "If x has a fever, y has measles, and x and y live in the same house, then x has measles."

This rule can then be combined with rule no. 2 to predict that x will get red spots.

Induction is often viewed as the dual of deduction: in the latter case we derive the special case from the general theory, while in the former, we construct a general theory from a number of given particular cases, namely the examples. One important difference between deduction and induction, is the fact that deduction is *truth-preserving*: if the general theory is true, then the derived particular cases are also true. Induction, on the other hand, is not truth-preserving: the examples may be true, while the induced theory is false. For instance, even if our rules on measles are true for all records of all the hospitals in the world, they may still be false for people not in the records.

The study of induction can be approached from many angles. It used to be mainly an issue for philosophers of science (see Section 1.5), but is nowadays also often studied in relation to computer algorithms, within the field of Artificial Intelligence (AI). As Marvin Minsky, one of the founders of AI, wrote: "Artificial Intelligence is the science of making machines do things that would require intelligence if done by man" [Min68, p. v]. Given this view, the study of induction is indeed part of AI, since learning from examples certainly requires intelligence if done by man.

The branch of AI which studies learning is called *Machine Learning*. Some of the main approaches in Machine Learning are learning in *neural networks*, *decision trees*, *genetic algorithms* and finally *logic*. The latter approach is nowadays called Inductive Logic Programming (ILP). Stephen Muggleton, when introducing the name Inductive Logic Programming, defined this field as the intersection of Machine Learning and Logic Programming. Thus ILP studies learning from examples, within the framework provided by clausal logic. Here the examples and background knowledge are given as clauses, and the theory that is to be induced from these, is also to consist of clauses. Using logic has some important advantages over other approaches used in Machine Learning:

- Logic provides a uniform and very expressive means of representation: the background knowledge and the examples, as well as the induced theory, can all be represented as formulas in a clausal language.
- Knowledge represented as rules and facts over certain predicates comes much closer to natural language than any of the other approaches. Hence the set of clauses that an ILP-system induces is much easier to interpret for us humans than, for instance, a neural network.
- The use of background knowledge fits very naturally within a logical approach towards Machine Learning.

The remainder of this chapter is organized as follows. In the next section, we will define the problem setting of induction in the precise terms of clausal logic and introduce some terminology. In Section 1.4 we discuss some alternatives to this setting. Section 1.5 gives a brief survey of the history of induction in general and ILP in particular. We end the chapter with an outline of the rest of the thesis.

1.3 The problem setting for ILP

Inductive Logic Programming concerns learning a general theory from given examples on the predicates that we want to learn, possibly taking background knowledge into account. We can distinguish between two kinds of examples: positive examples, which are true, and negative examples, which are false. Usually, the positive and negative examples are given as sets E^+ and E^-, respectively, of *ground atoms*. However, ground *clauses* are also sometimes used as examples, for instance in a *least generalization*-approach. In fact, one might even use non-ground clauses as examples, though this would be very unusual.

In ILP, both background knowledge and the induced theory are represented as finite sets of clauses. After the learning is done, the theory together with the background knowledge should imply all given positive examples (this is called *completeness*) and should not contradict the given negative examples (*consistency*). Completeness and consistency together form *correctness*.

Definition 1.1 Let Σ be a finite set of clauses and E^+ and $E^- = \{e_1, e_2, \ldots\}$ be sets of clauses. Σ is *complete* w.r.t. E^+ if $\Sigma \models E^+$. Σ is *consistent* w.r.t. E^- if $\Sigma \cup \{\neg e_1, \neg e_2, \ldots\}$ is satisfiable. Σ is *correct* w.r.t. E^+ and E^-, if Σ is complete w.r.t. E^+ and consistent w.r.t. E^-. $\diamond$

It follows from Proposition A.1 that if Σ implies one of the clauses in E^-, then it is not consistent w.r.t. E^-. The converse need not hold. For instance, let $\Sigma = \{P(a) \vee P(b)\}$ and $E^- = \{P(a), P(b)\}$. Then Σ does not imply any of the negative examples, but is still not consistent w.r.t. E^-. However, if we restrict the possible theories to definite programs and the negative examples to ground atoms, then the converse *does* hold:

Proposition 1.1 *Let Π be a definite program and E^- be a set of ground atoms. Then Π is consistent w.r.t. E^- iff $\Pi \not\models e$, for every $e \in E^-$.*

Proof Π is consistent w.r.t. $E^- = \{e_1, e_2, \ldots\}$ iff
$\Pi \cup \{\neg e_1, \neg e_2, \ldots\}$ is satisfiable iff (by Proposition A.4)
$\Pi \cup \{\neg e_1, \neg e_2, \ldots\}$ has an Herbrand model iff
M_Π does not contain any $e \in E^-$ iff (by Theorem A.6)
$\Pi \not\models e$, for every $e \in E^-$. $\square$

Several deviations from correctness are the following:

Definition 1.2 Let Σ be a finite set of clauses and E^+ and E^- be sets of clauses. Σ is *too strong* w.r.t. E^-, if Σ is not consistent w.r.t. E^-. Σ is *too weak* w.r.t. E^+, if Σ is not complete w.r.t. E^+.

Σ is *overly general* w.r.t. E^+ and E^-, if Σ is complete w.r.t. E^+ but not consistent w.r.t. E^-. Σ is *overly specific* w.r.t. E^+ and E^-, if Σ is consistent w.r.t. E^- but not complete w.r.t. E^+. $\diamond$

Note that Σ is correct iff it is neither too strong nor too weak.

Example 1.1 Suppose we have $E^+ = \{P(s(0)), P(s^3(0)), P(s^5(0)), P(s^7(0))\}$, and $E^- = \{P(0), P(s^2(0)), P(s^4(0))\}$. Then $\Sigma = \{(P(s^2(x)) \leftarrow P(x)), P(s(0))\}$ is correct w.r.t. E^+ and E^-. Note that Σ can be viewed as characterizing the odd numbers.

$\Sigma' = \{P(s^2(x))\}$ is both too strong w.r.t. E^- *and* too weak w.r.t. E^+. It is too strong because it implies some negative examples and it is too weak because it does not imply the positive example $P(s(0))$.

$\Sigma'' = \{P(s(x))\}$ is overly general w.r.t. E^+ and E^-. $\triangleleft$

Now the learning problem for ILP can be formally defined. This problem setting sometimes goes under the names of *normal* setting, or *explanatory* setting (since the theory should, in a sense, be an *explanation* of the examples).

Inductive Logic Programming: Problem Setting.

> **Given:** A finite set of clauses B (*background knowledge*), and sets of clauses E^+ and E^- (positive and negative *examples*).
> **Find:** A finite set of clauses Σ (*theory*), such that $\Sigma \cup B$ is correct w.r.t. E^+ and E^-.

As we have emphasized above, E^+ and E^- are most often restricted to ground *atoms*. We may sometimes be learning from scratch. In this case, no background knowledge is present, and B (the empty set) can be dropped from the problem setting.

Note that a solution Σ does not always exist. The first reason for this is rather trivial: $B \cup E^+$ may be inconsistent w.r.t. the negative examples, for instance if $P(a)$ is both a positive and a negative example at the same time. To solve this, we have to require that $B \cup E^+$ is consistent w.r.t. E^-.

The second reason for the non-existence of a solution is more profound. Note that our problem setting allows *infinite* sets of examples. One instance of this is Shapiro's setting for model inference [Sha81b]. Here the examples are given in an *enumeration*, which may be infinite. Allowing an infinite number of examples implies, roughly, that there are "more" possible sets of examples than there are theories. Hence a correct theory does not always exist, even when the examples can only be ground atoms and background knowledge is not used, as proved in the next theorem.

The proof of this theorem employs two different "kinds of infinity". The first kind concerns sets containing the same number of elements as the set of natural numbers. Such sets are called *enumerably infinite*. The second kind of infinite set is called *uncountable*. An example of an uncountable set is the set of

real numbers. It is well-known that the *power set* of an enumerably infinite set S (the set of all subsets of S) is uncountable and that the latter is "larger" than the former. A more extensive introduction into these matters can be found in many mathematics books, for instance [BJ89].

Theorem 1.1 *There exist sets E^+ and E^- of ground atoms, such that there is no finite set of clauses which is correct w.r.t. E^+ and E^-.*

Proof Consider a clausal language $\mathcal{C}$ containing (possibly among others) a function symbol of arity ≥ 1 and a constant a. Let $\mathcal{A}$ be the set of ground atoms in $\mathcal{C}$. If Σ is some finite set of clauses, let $\mathcal{A}_\Sigma = \{A \in \mathcal{A} \mid \Sigma \models A\}$.

The number of clauses in $\mathcal{C}$ is enumerably infinite. Then because a theory is a finite set of clauses, the number of theories is also enumerably infinite. Thus the number of different $\mathcal{A}_\Sigma$'s induced by all possible theories, is also only enumerably infinite.

The power set of $\mathcal{A}$ is uncountable. Since an uncountable set is much larger than an enumerably infinite one, there must be a set $E^+ \subseteq \mathcal{A}$, such that there is no finite Σ for which $\mathcal{A}_\Sigma = E^+$. Define $E^- = \mathcal{A} \backslash E^+$. Now for a theory Σ to be correct w.r.t. E^+ and E^-, we must have $\mathcal{A}_\Sigma = E^+$. Hence there is no such Σ. □

If E^+ is finite, then $\Sigma = E^+$ will be a correct theory, but a rather uninteresting one. In this case, we would not have learned anything beyond the given examples: the induced theory has no predictive power. To avoid this, we can put some constraints on the theory. For instance, we might demand that Σ contains less clauses than the number of given positive examples. In that case, $\Sigma = E^+$ is ruled out. Since constraints like these mainly depend on the particular application at hand, we will not devote much attention to them.

Anyhow, if one or more correct theories do exist, then they are "hidden" somewhere in the set of clauses in the language we use. Accordingly, finding a satisfactory theory means that we have to *search* among the available clauses: learning is searching for a correct theory [Mit82]. Hence the set of available clauses is called the *search space*.

The two basic steps in the search for a correct theory are *specialization* and *generalization*. If the current theory (together with the background knowledge) is too strong, it needs to be weakened. That is, we need to find a more specific theory, such that the new theory and the background knowledge are consistent w.r.t. the negative examples. This is called specialization. On the other hand, if the current theory does not imply all positive examples, it needs to be strengthened: we need to find a more general theory which (together with the background knowledge) can imply all positive examples. This is generalization. Note that a theory may be both too strong and too weak at the same time, witness Σ' in Example 1.1. In this case, both specialization *and* generalization are called for. In general, finding a correct theory means repeatedly adjusting the theory to the examples with these two technique (specialization and generalization). Whether a particular theory is too weak or too strong, can be tested using one of the proof procedures we will introduce in the next chapter.

In general, most ILP-systems conform roughly to the following scheme:

Input: B, E^+ and E^-.
Output: A theory Σ, such that $\Sigma \cup B$ is correct w.r.t. E^+ and E^-.

Start with some initial (possibly empty[2]) theory Σ.
Repeat

1. If $\Sigma \cup B$ is too strong, specialize Σ.
2. If $\Sigma \cup B$ is too weak, generalize Σ.

until $\Sigma \cup B$ is correct w.r.t. E^+ and E^-.
Output Σ.

Thus the main operations an ILP-system should perform, are specialization and generalization. The following chapters can be considered as an investigation into a number of different approaches towards specialization and generalization, which can be used when searching for a correct theory. We will now introduce some terminology often used in ILP:

Top-down and bottom-up
One useful distinction among ILP-systems concerns the direction in which a system searches. First, there is the *top-down* approach, which starts with a Σ such that $\Sigma \cup B$ is overly general, and specializes this. Secondly, there is the *bottom-up* approach which starts with a Σ such that $\Sigma \cup B$ is overly specific, and generalizes this. Admittedly, a top-down system may sometimes locally adapt itself to the examples by a generalization step. Such a generalization step may be needed to correct a (large) earlier specialization step, which made the theory too weak. After the correction, the system continues its general top-down search.

Analogously, a bottom-up system may sometimes make a specialization step. Nevertheless, a system can usually be classified in a natural way as top-down or bottom-up, depending on the general direction of its search.

Example 1.2 Consider the sets E^+ and E^- of Example 1.1. Assume the background knowledge is empty. A top-down approach may take the following steps to reach a correct theory.

1. Start with $\Sigma = \{P(x)\}$.
2. This is clearly overly general, since it implies all negative examples. Specialize it to $\Sigma = \{P(s(x)), P(0)\}$.
3. Σ is still too general, for instance, it implies $P(0) \in E^-$. Specialize it to $\Sigma = \{P(s^2(x)), P(s(0))\}$.
4. Now Σ no longer implies $P(0)$, but it is still overly general. When we specialize further to $\Sigma = \{(P(s^2(x)) \leftarrow P(x)), P(s(0))\}$, we end up with a theory that is correct w.r.t. E^+ and E^-.

◁

[2]If we start with a non-empty theory Σ, the learning task is sometimes called *theory revision*.

Single- and multiple-predicate learning
We can also distinguish between *single-predicate learning* and *multiple-predicate learning*. In the former case, all given examples are instances of only one predicate P and the aim of the learning task is to find a set of clauses which implies $P(x_1, \ldots, x_n)$ just for those tuples $\langle x_1, \ldots, x_n \rangle$ whose denotation "belongs" to the concept denoted by P. In other words, the set of clauses should "recognize" the instances of P.

In multiple-predicate learning, the examples are instances of more than one predicate. Note that multiple-predicate learning cannot always be split into several single-predicate problems, because the different predicates in a multiple-predicate learning task may be related.

Batch learning and incremental learning
The distinction between *batch learning* and *incremental learning* concerns the way the examples are given. In batch learning, we are given all examples E^+ and E^- right at the outset. This has the advantage that *noise* (errors in the given examples) can be measured and dealt with by applying statistical techniques to the set of all examples [LD94]. Since the treatment of noise is usually application-dependent, we will not give much attention to noisy examples in this thesis.

On the other hand, in incremental learning the examples are given one by one, and the system each time adjusts its theory to the examples given so far, before obtaining the next example.

Interactive and non-interactive
Interactive systems can interact with their user in order to obtain some extra information. For instance, they can ask the user whether some particular ground atom is true or not. In this way, an interactive system generates some of its own examples during the search. A non-interactive system does not have the possibility to interact with the user.

Bias
Bias concerns anything which constrains the search for theories [UM82]. We can distinguish two kinds of bias: *language* bias and *search* bias.

Language bias has to do with constraints on the clauses in the search space. These may for instance be a restriction to Horn clauses, to clauses without function symbols and constants, to clauses with at most n literals, etc. The more restrictions we put on clauses, the smaller the search space, and hence the faster a system will finish its search. On the other hand, restrictions on the available clauses may cause many good theories to be overlooked. For example, we may restrict the search space to clauses of at most 5 literals, but if the only correct theory contains clauses of 6 or more literals, no solution will be found. Thus there is in general a *trade-off* between the efficiency of an ILP-system and the quality of the theory it comes up with.

One important matter concerning language bias, is the capability of a system to introduce new predicates when needed. A restriction of the language to the predicates already in use in the background theory and the examples

may sometimes be too strict. In that case *predicate invention* (the automatic introduction of new useful predicates) is called for. For example, if we are learning about family relations and neither the examples nor the background knowledge contain a predicate for *parenthood*, it would be nice if the system could introduce such a useful predicate itself.

Search bias has to do with the way a system searches its space of available clauses. One extreme is exhaustive search, which searches the search space completely. However, usually exhaustive search would take far too much time, so the search has to be guided by certain *heuristics*. These indicate which parts of the space are searched and which are ignored. Again, this may cause the system to overlook some good theories. So here we see another trade-off between efficiency and the quality of the final theory.

If a system has found that a correct theory is not available using its present language and search bias, it can try again using a more general language and/or a more thorough search procedure. This is called *shifting the bias*.

1.4 Other problem settings

The normal problem setting that we introduced above, is used in some form or other by the majority of ILP-researchers. However, in recent years a family of other problem settings has appeared. These settings have in common that the induced theory should no longer *imply* the positive examples, but should be a general relation that is *true* for the examples. Examples are Helft's *non-monotonic* setting for induction [Hel89, DRD94], Flach's *weak* induction [Fla92] and *confirmatory* induction [Fla94, Fla95]. These settings are well-suited for the problem of *data-mining* or *knowledge discovery*: given a large amount of data (usually only positive examples), find "interesting" regularities among the data. However, since as yet there is not much consensus in ILP on one particular setting for this problem, we restrict ourselves to the "normal" problem setting defined in the previous section. Nevertheless, specialization and generalization of clausal theories are the main operations not only for our setting, but also within the alternative settings. Hence the techniques of the next chapters are applicable within those alternative settings as well.

Apart from comparing our setting with alternative settings used *within* ILP, we can also compare it with problem settings *outside* of ILP. One of these is the problem of *abduction*, first introduced by the philosopher Charles Sanders Peirce [Pei58]. The logical form of abduction is roughly the same as for induction [DK96, KKT93]. Both proceed from given examples and some background knowledge. However, the theory that abduction produces should be a *particular fact* (often representable as one or more ground atoms) that together with the background knowledge explains the examples, whereas induction should produce a *general theory*.

For example, suppose you are Robinson Crusoe on his island and you see a strange human footprint in the sand. Since you know that human footprints are produced by human beings and the footprint is not your own, you can conclude on the basis of your background knowledge that someone else has visited your

island. Inferring this particular explanation of the example (the presence of the footprint) is a case of abduction.

1.5 A brief history of the field

Like most other scientific disciplines, the study of induction started out as a part of philosophy. Philosophers particularly focused on the role induction plays in the empirical sciences. For instance, the ancient Greek philosopher Aristotle characterized science roughly as deduction from first principles, which were to be obtained by induction from experience [Ari60].

After the Middle Ages, the philosopher Francis Bacon [Bac20] again stressed the importance of induction from experience as the main scientific activity. In later centuries, induction was taken up by many philosophers. David Hume [Hum56, Hum61] formulated what is nowadays called the 'problem of induction', or 'Hume's problem': how can induction from a finite number of cases result in knowledge about the infinity of cases to which an induced general rule applies? What justifies inferring a general rule (or "law of nature") from a finite number of cases? Surprisingly, Hume's answer was that there is *no* such justification. In his view, it is simply a psychological fact about humans beings that when we observe some particular pattern recur in different cases (without observing counterexamples to the pattern), we tend to expect this pattern to appear in all similar cases. In Hume's view, this inductive expectation is a *habit*, analogous to the habit of a dog who runs to the door after hearing his master call, expecting to be let out.

Later philosphers such as John Stuart Mill [Mil58] tried to answer Hume's problem by stating conditions under which an inductive inference is justified. Other philosophers who made important comments on induction were Stanley Jevons [Jev74] and Charles Sanders Peirce [Pei58].

In our century, induction was mainly discussed by philosophers and mathematicians who were also involved in the development and application of formal logic. Their treatment of induction was often in terms of the probability or the "degree of confirmation" that a particular theory or hypothesis receives from available empirical data. Some of the main contributors are Bertrand Russell [Rus80, Rus48], Rudolf Carnap [Car52, Car50], Carl Hempel [Hem45a, Hem45b, Hem66], Hans Reichenbach [Rei49], and Nelson Goodman [Goo83]. Particularly in Goodman's work, an increasing number of unexpected conceptual problems appeared for induction.

In the 1950s and 1960s, induction was sworn off by philosophers of science such as Karl Popper [Pop59].[3] However, in roughly those same years, it was recognised in the rapidly expanding field of Artificial Intelligence that the knowledge an AI-system needs to perform its tasks, should not all be hand-coded into the system beforehand. Instead, it is much more efficient to provide the system with a relatively small amount of knowledge and with the ability to

[3] Interestingly enough, Thomas Kuhn, Poppers antipode in the philosophy of science, later became involved in computer models of (inductive) concept learning from examples. See pp. 474–482 of [Kuh77].

adapt itself to the situations it encounters—to *learn* from its experience. Thus the study of induction switched from philosophy to Artificial Intelligence.

Clause Sammut [Sam93] starts his article on the history of ILP with the work of Bruner, Goodnow and Austin [BGA56] in cognitive psychology. They analyzed the way human beings learn concepts from positive and negative instances (examples) of that concept. In the early 1960s, Banerji [Ban64] used first-order logic as a representational tool for such concept learning.

Around 1970, Gordon Plotkin [Plo70, Plo71b, Plo71a] was probably the first to formalize induction in terms of *clausal* logic. His idea was to generalize given ground clauses (positive examples) by computing their *least generalization*. This generalization could be relative to background knowledge consisting of ground literals. Plotkin's work, which is related to that of John Reynolds [Rey70], is still quite prominent within ILP. Clauses are still used by virtually everyone for expressing theory, examples and background knowledge, and Plotkin's use of *subsumption* as a notion of generality is also widespread. During the 1970s, Plotkin's work was continued by Steven Vere [Ver75, Ver77], while Brian Cohen's incremental system CONFUCIUS was inspired by Banerji.

In the early 1980s, Sammut's MARVIN [Sam81, SB86] was a direct descendant of CONFUCIUS. MARVIN is an interactive concept learner, which employs both generalization and specialization. At around the same time, Ehud Shapiro [Sha81b, Sha81a] defined his setting for *model inference* and contructed his model inference algorithm. This is a top-down algorithm aimed at finding *complete axiomatizations* of given examples. Shapiro's work contains many seminal ideas, in particular the use of the *Backtracing Algorithm* for finding false clauses in the theory, and the concept of a *refinement operator*, used for specializing a theory. Shapiro implemented his algorithm, though only for Horn clauses, in his model inference system MIS. He later incorporated this work in his PhD thesis [Sha83], as part of a system for debugging definite programs.

Then in the second half of the 1980s—no doubt partly as a consequence of the increasing popularity of Logic Programming and PROLOG—research concerning machine learning within a clausal framework increased rapidly. Wray Buntine [Bun86, Bun88] generalized subsumption, in order to overcome some of its limitations. Stephen Muggleton built his system DUCE [Mug87], aimed at generalizing given *propositional* clauses. It became clear that DUCE's generalization operators could be seen as inversions of resolution steps. Thus in [MB88] Muggleton, together with Buntine, introduced *inverse resolution*. They implemented inverse resolution, both as an operator for making generalization steps and as a tool for predicate invention in CIGOL ('logic' backwards). In the next years, inverse resolution drew a lot of attention and sparked off much new research.

Some early alternatives to inverse resolution were implemented in FOIL, LINUS and GOLEM. FOIL [Qui90, QCJ93] is based on a downward refinement operator guided by information-based search heuristics, in which J. R. Quinlan generalized his earlier work on decision trees [Qui86] to Horn clauses. LINUS was developed by Nada Lavrač and Sašo Džeroski [LDG91, LD94]. It solves ILP-problems by transforming them to a simpler *attribute-value* form, represented as a set of objects with certain properties, and then applying one of several

possible attribute-value learners to learn a general theory from this simpler form. See [LD92a, LD92b, LD94] for a detailed comparison of FOIL and LINUS. Muggleton and Feng's GOLEM [MF92] is in a way a return to Plotkin: it is based on Plotkin's relative least generalization, though with additional restrictions for the sake of efficiency.

In 1990, Stephen Muggleton first introduced the name Inductive Logic Programming, and defined this field as the intersection of Machine Learning and Logic Programming [Mug90, Mug91]. In the next year he organized, together with Pavel Brazdil, the first International Workshop on Inductive Logic Programming, bringing together for the first time a number of researchers involved in learning from examples in a clausal framework. Since 1991 these International Workshops have been repeated every year, establishing ILP as a flourishing field of inquiry.

1.6 An outline of the thesis

In this section, we will give an outline of the remainder of the thesis. Three topics are of particular concern in ILP: deduction, specialization, and generalization. Each of these will be addressed in later chapters.

Deduction allows us to find out whether the current theory is correct (complete and consistent) w.r.t. the examples. This is clearly important, since it determines the direction in which the theory has to be adapted: if the theory is not complete, it has to be strengthened; if the theory is not consistent, it has to be weakened. In the next chapter we will investigate four different deductive procedures, each based on the *resolution* principle. For "unconstrained" resolution, linear resolution and SLD-resolution, we will prove a major completeness result, called the *Subsumption Theorem*. Moreover, we will show that this theorem is equivalent to the *refutation-completeness*, for each of these kinds of resolution. On the other hand, we will also show that both of these completeness results do not hold for *input* resolution.

Specialization can be used to weaken a theory. In Chapter 3 we investigate the use of *unfolding* as a specialization tool. We define three increasingly strong specialization methods, UD_1-specialization, UD_2-specialization and UDS-specialization, based on unfolding, clause deletion and subsumption. The latter is a complete specialization technique for definite programs, while the first two are not.

Finally, in Chapter 4 we discuss *least generalizations* and *greatest specializations* of sets of clauses. These can respectively be used to strengthen and weaken a theory. Usually, least generalizations are only considered under the *subsumption* order. We extend it here to the *logical implication* order, which is more powerful than subsumption.

It is interesting to note that the proofs of both the completeness result given in Chapter 3, as well as the main new result of Chapter 4 (the existence of a least generalization under implication in the presence of a function-free clause), depend on the Subsumption Theorem(s) of Chapter 2. This gives a kind of unity and coherence to the thesis.

Chapter 2

The Subsumption Theorem for Several Forms of Resolution

2.1 Introduction

The *Subsumption Theorem* is the following statement:

> If Σ is a set of clauses and C is a clause, then Σ logically implies C ($\Sigma \models C$) iff C is a tautology, or there exists a clause D which subsumes C and which can be derived from Σ by some form of resolution.

Different versions of the theorem exist, depending on the instantiation of "some form of resolution." We could allow arbitrary binary trees of resolution steps ("unconstrained resolution") as derivation, or we could allow only linear derivations, etc. This is similar to the *refutation-completeness* for proof by contradiction: here we have the refutation-completeness of unconstrained resolution, the refutation-completeness of linear resolution, etc.

The refutation-completeness is a form of completeness that is much better known than the Subsumption Theorem. It states that a set of clauses is unsatisfiable iff the set has a refutation (a derivation of the empty clause $\Box$, which represents a contradiction). It can be used to prove any case of logical implication between clauses. For if we have a set Σ, a clause C, θ is a Skolem substitution for C w.r.t. Σ and $C\theta = L_1 \vee \ldots \vee L_n$, then $\Sigma \models C$ iff $\Sigma \cup \{\neg L_1, \ldots, \neg L_n\}$ is unsatisfiable iff $\Sigma \cup \{\neg L_1, \ldots, \neg L_n\}$ has a refutation.

However, the Subsumption Theorem is a more "direct" form of completeness than the refutation-completeness. By the Subsumption Theorem, we can straightforwardly prove C from Σ by taking a number of resolution steps starting from clauses in Σ, and then taking a subsumption step leading to C. There is no need to use the detour which first negates C and then applies proof by refutation. In a derivation of a clause which subsumes C, the relation between on one hand the premises in Σ and on the other hand the conclusion C, is much

easier to see than in a proof by refutation. Accordingly, the Subsumption Theorem gives us a more clear view of the structure of logical implication than the refutation-completeness. For this reason, the Subsumption Theorem is sometimes a more convenient result than the refutation-completeness, perhaps not for efficient deduction, but certainly for theoretical analysis.

Examples of such theoretical analysis are the various ways the theorem is applied in Inductive Logic Programming. The use of subsumption is very popular in ILP, since it is decidable and machine-implementable. However, subsumption is weaker than implication: if C subsumes D then $C \models D$, but not always the other way around, take for instance $C = P(f(x)) \leftarrow P(x)$ and $D = P(f^2(x)) \leftarrow P(x)$. So it is desirable to make the step from subsumption to implication, and the Subsumption Theorem provides an excellent "bridge" for those who want to make this step, since it states that implication = resolution + subsumption. It is used for instance in [Mug92c, IA93, IA95, LNC94b, LNC94a]. The theorem is also an essential ingredient in the proofs of the main results in later chapters of the present thesis. It is rather doubtful whether we would have found those same results if we only had the refutation-completeness at our disposal, but not the Subsumption Theorem.

As our survey later on in this section will show, in automated theorem proving the Subsumption Theorem received most attention around 1970. In ILP, the Subsumption Theorem was first rediscovered by Bain and Muggleton [BM92].[1] A proof of the Subsumption Theorem for unconstrained resolution, based on the refutation-completeness, is given in the appendix of [BM92]. However, this proof seems not fully correct. For example, it does not take *factors* into account, whereas factors are necessary for completeness. Without factors one cannot derive the empty clause $\Box$ from the unsatisfiable set $\{(P(x) \vee P(y)), (\neg P(u) \vee \neg P(v))\}$ (see [GN87]). Furthermore, it is not always clear how the concepts that are used in the proof are defined, and how the Skolemization works.

Even though the proof in [BM92] is not quite correct, it is often quoted—sometimes even incorrectly. The two main formulations of the Subsumption Theorem that we have found in ILP-literature, are the following:

S Let Σ be a set of clauses and C a clause which is not a tautology. Define $\mathcal{R}^0(\Sigma) = \Sigma$ and $\mathcal{R}^n(\Sigma) = \mathcal{R}^{n-1}(\Sigma) \cup \{C \mid C \text{ is a resolvent of } C_1, C_2 \in \mathcal{R}^{n-1}(\Sigma)\}$. Also define $\mathcal{R}^*(\Sigma) = \mathcal{R}^0(\Sigma) \cup \mathcal{R}^1(\Sigma) \cup \ldots$. Then the Subsumption Theorem is stated as follows:
$\Sigma \models C$ iff there exists a clause $D \in \mathcal{R}^*(\Sigma)$ such that D subsumes C.

S′ Let Σ be a set of clauses and C a clause which is not a tautology. Define $\mathcal{L}^1(\Sigma) = \Sigma$ and $\mathcal{L}^n(\Sigma) = \{C \mid C \text{ is a resolvent of } C_1 \in \mathcal{L}^{n-1}(\Sigma) \text{ and } C_2 \in \Sigma\}$. Also define $\mathcal{L}^*(\Sigma) = \mathcal{L}^1(\Sigma) \cup \mathcal{L}^2(\Sigma) \cup \ldots$. Then the Subsumption Theorem is stated as follows:
$\Sigma \models C$ iff there exists a clause $D \in \mathcal{L}^*(\Sigma)$ such that D subsumes C.

[1] From personal communication with Stephen Muggleton, we know Bain and Muggleton discovered the theorem themselves, independently of [Lee67]. Only afterwards did they found out from references in other literature that their theorem was roughly the same as the theorem in Lee's thesis.

S is given in [BM92], **S′** is given in [Mug92c]. [Mug92c] does not include a proof of **S′**, but refers instead to [BM92]. In other work such as [IA93, NCLT93, LNC94b, MDR94, BG96], the theorem is also given in the form of **S′**. These texts do not give a proof of **S′**, but refer instead to [BM92] or [Mug92c]. That is, they refer to a proof of **S** assuming that this is also a proof of **S′**. But clearly that is not the case, because **S′** demands that at least one of the parent clauses of a clause in $\mathcal{L}^*(\Sigma)$ is a member of Σ, so **S′** is stronger than **S**. In fact, whereas **S** is true, **S′** *is actually false!* If **S′** were true, then input resolution would be refutation-complete which it is not, as we will see in Section 2.6.

The confusion about **S′** is perhaps a consequence of the subtle distinction between linear resolution and input resolution. **S′** employs a form of *input* resolution, which is a special case of linear resolution. Linear resolution is complete, as is well-known, but input resolution is not.

However, the articles we mentioned do not always use **S′** itself. [LNC94b, NCLT93] are restricted to Horn clauses. In Section 2.7 we show that SLD-resolution for Horn clauses has its own Subsumption Theorem, so for Horn clauses there is no problem. If we examine [Mug92c, IA93, MDR94, BG96], carefully, then we see that the results of these articles (which are also about non-Horn clauses) only depend on a special case of **S′**, namely the case where Σ consists of a single clause. Unfortunately, **S′** does not even hold in this special case. We give a counterexample in Section 2.6. This means that results which are consequences of this special case of **S′** need to be reconsidered.[2] These particularly include results on *nth powers* and *nth roots*. If $D \in \mathcal{L}^n(\{C\})$, then D is called an nth power of C, and C is called an nth root of D. Clearly, the falsity of **S′** renders false the completeness of powers and roots reported in [Mug92c, MDR94, BG96].

This confusion in the ILP-community about various forms of the Subsumption Theorem provided the motivation for the research we present in this chapter. Our aim was to find out for which versions of resolution the Subsumption Theorem holds, and for which it does not.

Let us first see what results have already been proved in the literature. Surprisingly, the Subsumption Theorem is mentioned nowhere in the standard reference books on resolution, such as [CL73, Lov78, Llo87]. Hence we have to rely on journals, conference proceedings and theses. A weak form of the Subsumption Theorem was first proved by Lee in 1967 in his PhD thesis [Lee67], only 2 years after Robinson's introduction of the resolution principle in [Rob65]. His result is the following: $\Sigma \models C$ iff C is a tautology, or there exists a clause D which *implies* C (and thus not necessarily *subsumes* C) and which can be derived from Σ by unconstrained resolution.

The "real" Subsumption Theorem—i.e., where D subsumes C rather than only implying it—appears to have been first proved in [SCL69]. Here the result is proved for several forms of *semantic* resolution. Since semantic resolution is a constrained form of resolution, their results immediately imply the Sub-

[2] Idestam-Almquist has adjusted his results from [IA93] in [IA95], incorporating our findings as published in [NCW95d].

sumption Theorem for unconstrained resolution.[3] Kowalski [Kow70] explicitly proved the result for unconstrained resolution, but his proof is rather sketchy and presupposes knowledge of semantic trees. Minicozzi and Reiter [MR72] proved the Subsumption Theorem for linear resolution. After that, interest in the Subsumption Theorem seems to have faded somewhat. However, recently Inoue [Ino92] has developed SOL-resolution (Skip Ordered Linear resolution) and proved a version of the Subsumption Theorem for it. He also gave an overview of the results of [Lee67, SCL69, MR72].

In this chapter, we consider four kinds of resolution: "unconstrained" resolution, linear resolution, input resolution, and SLD-resolution, the latter only for Horn clauses. We collect some of the results mentioned in the last paragraph and contribute some results of our own.

The chapter is organized as follows. In the next section we start out with our main definitions. In Section 2.3 we give a new, direct proof of the Subsumption Theorem for unconstrained resolution. In our opinion, this proof is easier to understand than earlier proofs of the same result [Kow70, BM92], which presuppose the refutation-completeness of resolution. In Section 2.4, we then show that the refutation-completeness of unconstrained resolution is an immediate corollary of the Subsumption Theorem. Conversely, in the same section we also give a second proof of the Subsumption Theorem, this time starting from the refutation-completeness. Thus these two completeness results are actually equivalent: the one can be proved from the other.

The Subsumption Theorem holds for linear resolution as well. In Section 2.5 we give a proof of this result which is similar to the proof given in [MR72]. Moreover, we also show that the Subsumption Theorem for linear resolution is equivalent to the refutation-completeness of linear resolution. In Section 2.6, we show that the Subsumption Theorem does not hold for input resolution, not even in case Σ contains only one clause, which is a new result.

Finally, in Section 2.7 we discuss SLD-resolution for Horn clauses. We first give a proof of the well-known refutation-completeness of SLD-resolution. This proof is easier to understand than the one given in [Llo87], since our proof does not require fixed-point theory. We then proceed to prove the Subsumption Theorem for SLD-resolution. This new result generalizes Theorem 18 of [MP94], which gives the result for the case where Σ contains only one clause. Moreover, as in the cases of unconstrained and linear resolution, we show that the Subsumption Theorem is equivalent to the refutation-completeness also in case of SLD-resolution.[4]

2.2 Preliminaries

In this section we define the main concepts concerning resolution.

[3] The name "completeness theorem for consequence finding" is also sometimes used. As far as we know, the name "Subsumption Theorem" was introduced in [Kow70].

[4] Though this equivalence holds for each of the forms of resolution that we discuss here, it does not holds for every conceivable kind of resolution. [MR72] discusses m.c.l.-resolution. This is refutation-complete, but the Subsumption Theorem does not hold for it.

Definition 2.1 Let C_1 and C_2 be clauses. If C_1 and C_2 have no variables in common, then they are said to be *standardized apart.* ◇

Definition 2.2 Let $C_1 = L_1 \vee \ldots \vee L_i \vee \ldots \vee L_m$ and $C_2 = M_1 \vee \ldots \vee M_j \vee \ldots \vee M_n$ be two clauses which are standardized apart. If the substitution θ is an mgu (most general unifier) of the set $\{L_i, \neg M_j\}$, then the clause

$$(L_1 \vee \ldots \vee L_{i-1} \vee L_{i+1} \vee \ldots \vee L_m \vee M_1 \vee \ldots \vee M_{j-1} \vee M_{j+1} \vee \ldots \vee M_n)\theta$$

is called a *binary resolvent* of C_1 and C_2. The literals L_i and M_j are said to be the literals *resolved upon.* ◇

Definition 2.3 Let C be a clause, $L_1, \ldots, L_n$ $(n \geq 1)$ some unifiable literals from C and θ an mgu for the set $\{L_1, \ldots, L_n\}$. Then the clause obtained by deleting $L_2\theta, \ldots, L_n\theta$ from $C\theta$ is called a *factor* of C. ◇

Note that every non-empty clause C is a factor of C itself, using the identity substitution ε as mgu for one literal in C. Factors are sometimes built into the resolution step itself—for instance in Robinson's original paper [Rob65], where *sets* of literals from both parent clauses are unified—but we have chosen to separate the definitions of a factor and a binary resolvent. The reason for this is that binary resolution without factors is sufficient in case of SLD-resolution for Horn clauses.

Definition 2.4 Let C_1 and C_2 be two clauses. A *resolvent* R of C_1 and C_2 is a binary resolvent of a factor of C_1 and a factor of C_2, where the literals resolved upon are the literals unified by the respective factors. C_1 and C_2 are called the *parent clauses* of R. ◇

It is easy to show that resolution is sound: if R is a resolvent of C_1 and C_2, then $\{C_1, C_2\} \models R$.

Below we define a *derivation.* In later sections, we will put some constraints on this concept, yielding, respectively, linear, input and SLD-derivations. The kind of derivation defined in this section, will sometimes be referred to as "unconstrained" resolution.

Definition 2.5 Let Σ be a set of clauses and C a clause. A *derivation* of C from Σ is a finite sequence of clauses $R_1, \ldots, R_k = C$, such that each R_i is either in Σ or a resolvent of two clauses in $\{R_1, \ldots, R_{i-1}\}$. If such a derivation exists, we write $\Sigma \vdash_r C$.

A derivation of the empty clause $\square$ from Σ is called a *refutation* of Σ. ◇

A derivation of a clause C from a set Σ can be represented as a binary tree of resolution steps, with clauses from Σ as leaves and C as root.

If we add a subsumption step to a derivation, we get a *deduction.*

Definition 2.6 Let C and D be clauses. We say C *subsumes* D if there exists a substitution θ such that $C\theta \subseteq D$. ◇

If C subsumes D, then $C \models D$. Subsumption is also sometimes called θ-subsumption.

Example 2.1 $C = P(x) \vee Q(x, y)$ subsumes $D = P(a) \vee Q(a, y) \vee R(x)$. ◁

Definition 2.7 Let Σ be a set of clauses and C a clause. We say there exists a *deduction* of C from Σ, written as $\Sigma \vdash_d C$, if C is a tautology, or if there exists a clause D such that $\Sigma \vdash_r D$ and D subsumes C. If $\Sigma \vdash_d C$, we say C can be *deduced* from Σ. ◇

Example 2.2 To illustrate these definitions, we will give an example of a deduction of the clause $C = R(a) \vee S(a)$ from the set $\Sigma = \{(P(x) \vee Q(x) \vee R(x)), (\neg P(x) \vee Q(a)), (\neg P(x) \vee \neg Q(x)), (P(x) \vee \neg Q(x))\}$. Figure 2.1 shows a derivation of the clause $D = R(a) \vee R(a)$ from Σ. Note that we use the factor $Q(a) \vee R(a)$ of the parent clause $C_6 = Q(x) \vee R(x) \vee Q(a)$ in the last step of the derivation, and also the factor $P(y) \vee R(y)$ of $C_5 = P(y) \vee P(y) \vee R(y)$ in the step leading to C_7. Since D subsumes C, we have $\Sigma \vdash_d C$.

It is not very difficult to see the equivalence between our definition of a derivation and the definition of $\mathcal{R}^n(\Sigma)$ we gave in Section 2.1. For instance, in figure 2.1, C_1, C_2, C_3, C_4, C_1' are variants of clauses in $\mathcal{R}^0(\Sigma)$ (C_1 and C_1' are variants of the same clause). C_5, C_6 are in $\mathcal{R}^1(\Sigma)$, C_7 is in $\mathcal{R}^2(\Sigma)$ and D is in $\mathcal{R}^3(\Sigma)$.

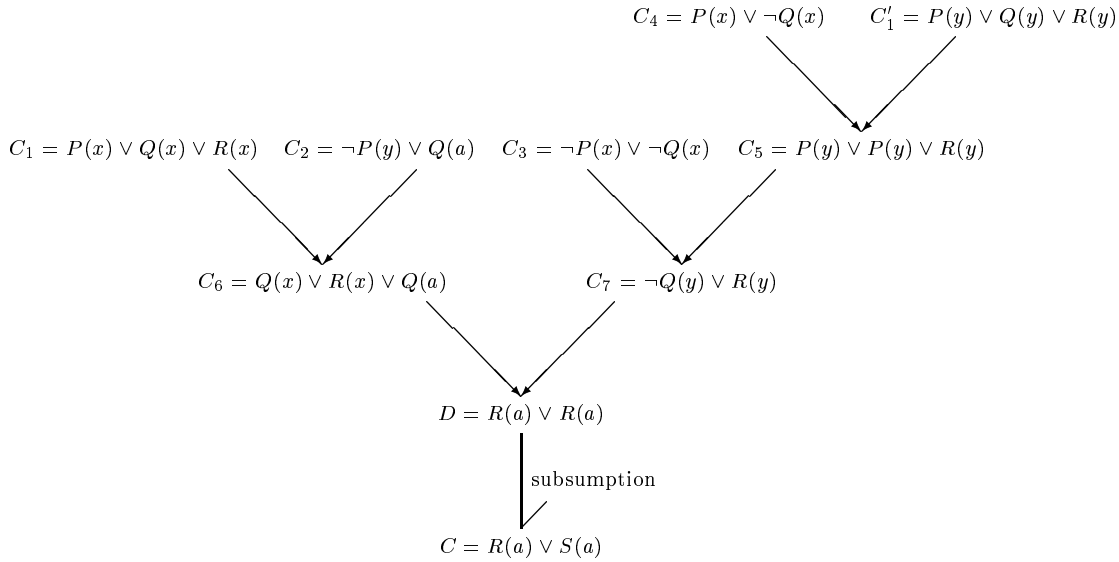


Figure 2.1: A deduction of C from Σ

◁

2.3 The Subsumption Theorem

In this section, we prove the Subsumption Theorem for (unconstrained) resolution: $\Sigma \models C$ iff $\Sigma \vdash_d C$. Thus any clause which is a logical consequence of

Σ, can be deduced from Σ. We prove this in a number of successive steps in the following subsections. First we prove the result in case both Σ and C are ground, then we prove it in case Σ consists of arbitrary clauses but C is ground, and finally we prove the theorem when neither Σ nor C need be ground.

2.3.1 The Subsumption Theorem for ground Σ and C

Lemma 2.1 *Let Σ be a set of ground clauses and C be a ground clause. If $\Sigma \models C$, then $\Sigma \vdash_d C$.*

Proof By Theorem A.3, we can assume Σ is finite. Assume C is not a tautology. Then we need to find a clause D such that $\Sigma \vdash_r D$ and $D \subseteq C$ (for ground clauses D and C, D subsumes C iff $D \subseteq C$). The proof is by induction on the number of clauses in Σ.

1. Suppose $\Sigma = \{C_1\}$. We will show that $C_1 \subseteq C$. Suppose $C_1 \not\subseteq C$. Then there exists a literal L such that $L \in C_1$ but $L \notin C$. Let I be an interpretation which makes L true and all literals in C false (such an I exists, since C is not a tautology). Then I is a model of C_1, but not of C. But that contradicts $\Sigma \models C$. So $C_1 \subseteq C$, and $\Sigma \vdash_d C$.
2. (See figure 2.2 for illustration of this case). Suppose the theorem holds if $|\Sigma| \leq m$. We will prove that this implies that the theorem also holds if $|\Sigma| = m + 1$. Let $\Sigma = \{C_1, \ldots, C_{m+1}\}$ and $\Sigma' = \{C_1, \ldots, C_m\}$. If C_{m+1} subsumes C or $\Sigma' \models C$, then the theorem holds. So assume C_{m+1} does not subsume C and $\Sigma' \not\models C$.
   The idea is to derive, using the induction hypothesis, a number of clauses from which a derivation of a subset of C can be constructed. First note that since $\Sigma' \cup \{C_{m+1}\} \models C$, it follows from Theorem A.1 that $\Sigma' \models (C_{m+1} \rightarrow C)$, hence $\Sigma' \models C \vee \neg C_{m+1}$.
   Let $L_1, \ldots, L_k$ be all the literals in C_{m+1} which are not in C ($k \geq 1$ since C_{m+1} does not subsume C). Then we can write $C_{m+1} = L_1 \vee \ldots \vee L_k \vee C'$, where $C' \subseteq C$. Since C does not contain L_i ($1 \leq i \leq k$), the clause $C \vee \neg L_i$ is not a tautology. Also, since $\Sigma' \models C \vee \neg C_{m+1}$ and C_{m+1} is ground, we have that $\Sigma' \models C \vee \neg L_i$, for each i. Then by the induction hypothesis there exists for each i a ground clause D_i such that $\Sigma' \vdash_r D_i$ and $D_i \subseteq (C \vee \neg L_i)$.
   We will use C_{m+1} and the derivations from Σ' of these D_i to construct a derivation of a subset of C from Σ. For each i, $\neg L_i \in D_i$, otherwise $D_i \subseteq C$ and $\Sigma' \models C$. So we can write each D_i as $\neg L_i \vee D_i'$, and $D_i' \subseteq C$ (the case where some D_i contains $\neg L_i$ more than once can be solved by taking a factor of D_i).
   Now we can construct a derivation of the ground clause defined as $D = C' \vee D_1' \vee \ldots \vee D_k'$ from Σ, using C_{m+1} and the derivations of $D_1, \ldots, D_k$ from Σ'. See figure 2.2 for a schematic representation of this derivation. In this tree, the derivations of $D_1, \ldots, D_k$ are indicated by the vertical dots. So we have that $\Sigma \vdash_r D$. Since $C' \subseteq C$, and $D_i' \subseteq C$ for each i, we have that $D \subseteq C$. Hence $\Sigma \vdash_d C$.

□

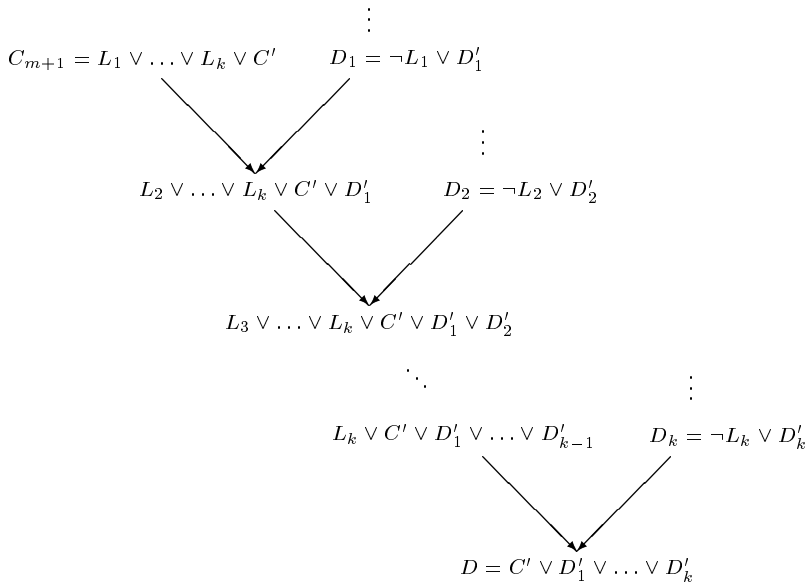


Figure 2.2: The tree for the derivation of D from Σ

2.3.2 The Subsumption Theorem when C is ground

In this section, we will prove the Subsumption Theorem in case C is ground and Σ is a set of arbitrary clauses. The idea is to "translate" $\Sigma \models C$ to $\Sigma_g \models C$, where Σ_g is a set of ground instances of clauses of Σ. Then by Lemma 2.1 there is a clause D such that $\Sigma_g \vdash_r D$ and D subsumes C. Finally, we "lift" this derivation to a derivation from Σ. The next two results show that logical implication between clauses can be translated to logical implication between ground clauses. The first of these is Herbrand's Theorem.

Theorem 2.1 (Herbrand) *A set of clauses Σ is unsatisfiable iff there exists a finite unsatisfiable set Σ_g of ground instances of clauses in Σ.*

Proof

$\Leftarrow$: Σ_g is a finite set of ground instances of clauses in Σ, so $\Sigma \models \Sigma_g$. Hence if Σ_g is unsatisfiable, then Σ is unsatisfiable.

$\Rightarrow$: Let Σ' be the (possibly infinite) set of all ground instances of clauses in Σ. Let I be an Herbrand interpretation. It is not very difficult to see that I is an Herbrand model of a clause C iff I is an Herbrand model of the set of all ground instances of C. Therefore I is a model of Σ iff I is a model of Σ'. Now we have the following:

Σ is unsatisfiable iff (by Proposition A.4)
Σ has no Herbrand models iff
Σ' has no Herbrand models iff (by Proposition A.4)
Σ' is unsatisfiable.

Finally, by the Compactness Theorem (Theorem A.2) there is a *finite* unsatisfiable subset Σ_g of Σ'. □

Theorem 2.2 *Let Σ be a set of clauses and C be a ground clause. If $\Sigma \models C$, then there exists a finite set Σ_g of ground instances of clauses in Σ, such that $\Sigma_g \models C$.*

Proof Let $C = L_1 \vee \ldots \vee L_k$ $(k \geq 0)$. If Σ is unsatisfiable then the lemma follows immediately from Theorem 2.1, so suppose Σ is satisfiable. Note that since C is ground, $\neg C$ is equivalent to $\neg L_1 \wedge \ldots \wedge \neg L_k$. Then:

> $\Sigma \models C$ iff (by Proposition A.1)
> $\Sigma \cup \{\neg C\}$ is unsatisfiable iff
> $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ is unsatisfiable iff (by Theorem 2.1)
> there exists a finite unsatisfiable set Σ', consisting of ground instances of clauses from $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$.

Since Σ is satisfiable, the unsatisfiable set Σ' must contain one or more members of the set $\{\neg L_1, \ldots, \neg L_k\}$, i.e. $\Sigma' = \Sigma_g \cup \{\neg L_{i_1}, \ldots, \neg L_{i_j}\}$, where Σ_g is a finite non-empty set of ground instances of clauses in Σ. So:

> Σ' is unsatisfiable iff
> $\Sigma_g \cup \{\neg L_{i_1}, \ldots, \neg L_{i_j}\}$ is unsatisfiable iff
> $\Sigma_g \cup \{\neg(L_{i_1} \vee \ldots \vee L_{i_j})\}$ is unsatisfiable iff (by Proposition A.1)
> $\Sigma_g \models (L_{i_1} \vee \ldots \vee L_{i_j})$.

Since $\{L_{i_1}, \ldots, L_{i_j}\} \subseteq C$, it follows that $\Sigma_g \models C$. □

Example 2.3 Let $\Sigma = \{(P(f(x)) \vee \neg P(x)), P(x)\}$ and $C = P(f(f(a)))$. Then $\Sigma \models C$. $\Sigma_g = \{(P(f(f(a))) \vee \neg P(f(a))), (P(f(a)) \vee \neg P(a)), P(a)\}$ is a set of ground instances of clauses of Σ, and $\Sigma_g \models C$. ◁

The following two lemmas are sufficient to "lift" a derivation, that is, to turn a derivation from instances of certain clauses into a derivation from those clauses themselves.

Lemma 2.2 *Let C_1 and C_2 be two clauses and C_1' and C_2' be instances of C_1 and C_2, respectively. If R' is a resolvent of C_1' and C_2', then there exists a resolvent R of C_1 and C_2, such that R' is an instance of R.*

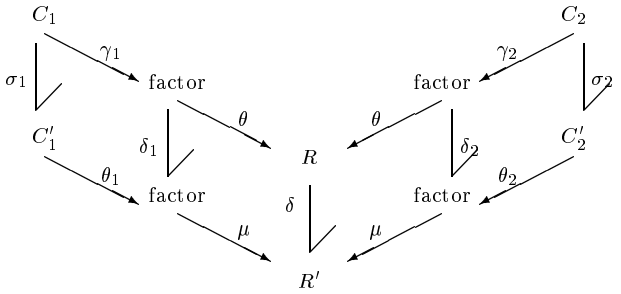
Proof We assume without loss of generality that C_1 and C_2, and C_1' and C_2' are standardized apart. Let $C_1 = L_1 \vee \ldots \vee L_m$, $C_2 = M_1 \vee \ldots \vee M_n$, $C_1' = C_1\sigma_1$ and $C_2' = C_2\sigma_2$. Suppose R' is a resolvent of C_1' and C_2'. Then R' is a binary resolvent of a factor of C_1' and a factor of C_2'. See the figure for illustration.

For notational convenience, we assume without loss of generality that the factor of C_1' is $(L_1 \vee \ldots \vee L_a)\sigma_1\theta_1$, where θ_1 is an mgu for $L_a\sigma_1, \ldots, L_m\sigma_1$. Similarly, the factor of C_2' that is used, is $(M_1 \vee \ldots \vee M_b)\sigma_2\theta_2$, where θ_2 is an mgu for $M_b\sigma_2, \ldots, M_n\sigma_2$. Let $L_i\sigma_1\theta_1$ and $M_j\sigma_2\theta_2$ be the literals resolved upon, with mgu μ. Abbreviate $L_1 \vee \ldots \vee L_{i-1} \vee L_{i+1} \vee \ldots \vee L_a$ to D_1 and $M_1 \vee \ldots \vee M_{j-1} \vee M_{j+1} \vee \ldots \vee M_b$ to D_2. Then $R' = (D_1\sigma_1\theta_1 \vee D_2\sigma_2\theta_2)\mu$.

By our assumption of standardizing apart, this can be written as $R' = (D_1 \vee D_2)\sigma_1\theta_1\sigma_2\theta_2\mu$.

Let γ_1 be an mgu for $L_a \vee \ldots \vee L_m$. Then $(L_1 \vee \ldots \vee L_a)\gamma_1$ is a factor of C_1. Note that $\sigma_1\theta_1$ is a unifier for $L_a, \ldots, L_m$. Since γ_1 is an mgu for $L_a, \ldots, L_m$, there exists a substitution δ_1 such that $\sigma_1\theta_1 = \gamma_1\delta_1$. Similarly, $(M_1 \vee \ldots \vee M_b)\gamma_2$ is a factor of C_2, with γ_2 as mgu for $M_b \vee \ldots \vee M_n$, and there is a δ_2 such that $\sigma_2\theta_2 = \gamma_2\delta_2$.

Since $L_i\sigma_1\theta_1$ and $\neg M_j\sigma_2\theta_2$ can be unified (they have μ as mgu) and γ_i is more general than $\sigma_i\theta_i$ $(i = 1, 2)$, $L_i\gamma_1$ and $\neg M_j\gamma_2$ can be unified. Let θ be an mgu for $L_i\gamma_1$ and $\neg M_j\gamma_2$. Define $R = (D_1\gamma_1 \vee D_2\gamma_2)\theta$, which can be written as $R = (D_1 \vee D_2)\gamma_1\gamma_2\theta$. Since R is a binary resolvent of the above-mentioned factors of C_1 and C_2, it is a resolvent of C_1 and C_2.



It remains to show that R' is an instance of R. Since $L_i\gamma_1\delta_1\delta_2\mu = L_i\sigma_1\theta_1\delta_2\mu = L_i\sigma_1\theta_1\mu = \neg M_j\sigma_2\theta_2\mu = \neg M_j\gamma_2\delta_2\mu = \neg M_j\gamma_2\delta_1\delta_2\mu$, the substitution $\delta_1\delta_2\mu$ is a unifier for $L_i\gamma_1$ and $\neg M_j\gamma_2$. θ is an mgu for $L_i\gamma_1$ and $\neg M_j\gamma_2$, so there exists a substitution δ such that $\delta_1\delta_2\mu = \theta\delta$. Therefore $R' = (D_1 \vee D_2)\sigma_1\theta_1\sigma_2\theta_2\mu = (D_1 \vee D_2)\gamma_1\delta_1\gamma_2\delta_2\mu = (D_1 \vee D_2)\gamma_1\gamma_2\delta_1\delta_2\mu = (D_1 \vee D_2)\gamma_1\gamma_2\theta\delta = R\delta$. Hence R' is an instance of R. □

Lemma 2.3 (Derivation lifting) *Let Σ be a set of clauses and Σ' a set of instances of clauses in Σ. Suppose $R_1', \ldots, R_k'$ is a derivation of the clause R_k' from Σ'. Then there exists a derivation $R_1, \ldots, R_k$ of the clause R_k from Σ, such that R_i' is an instance of R_i, for each i.*

Proof The proof is by induction on k.

1. Suppose $k = 1$. $R_1' \in \Sigma'$, so there exists a clause $R_1 \in \Sigma$ such that R_1' is an instance of R_1.
2. Suppose the lemma holds if $k \leq m$. Let $R_1', \ldots, R_m', R_{m+1}'$ be a derivation of R_{m+1}' from Σ'. By the induction hypothesis, there exists a derivation $R_1, \ldots, R_m$ of R_m from Σ, such that R_i' is an instance of R_i for all i, $1 \leq i \leq m$. If $R_{m+1}' \in \Sigma'$, the lemma is obvious. Otherwise, R_{m+1}' is a resolvent of two clauses C_1' and C_2' in $\{R_1', \ldots, R_m'\}$. Then there exist two clauses C_1 and C_2 in $\{R_1, \ldots, R_m\}$ such that C_1' is an instance of C_1 and C_2' is an instance of C_2. It follows from Lemma 2.2 that there is a resolvent R_{m+1} of C_1 and C_2, such that R_{m+1}' is an instance of R_{m+1}. So the lemma holds for $k = m + 1$.

□

The previous lemmas are sufficient to prove the Subsumption Theorem for the case where C is ground.

Lemma 2.4 *Let Σ be a set of clauses and C be a ground clause. If $\Sigma \models C$, then $\Sigma \vdash_d C$.*

Proof Assume C is not a tautology. We want to find a clause D such that $\Sigma \vdash_r D$ and D subsumes C. From $\Sigma \models C$ and Theorem 2.2, there exists a finite set Σ_g such that each clause in Σ_g is a ground instance of a clause in Σ, and $\Sigma_g \models C$. Then from Lemma 2.1 there exists a clause D' such that $\Sigma_g \vdash_r D'$, and D' subsumes C. Let $R'_1, \ldots, R'_k = D'$ be a derivation of D' from Σ_g. It follows from Lemma 2.3 that we can lift this to a derivation $R_1, \ldots, R_k$ of R_k from Σ, where D' is an instance of R_k. Let $D = R_k$. Then $\Sigma \vdash_r D$ and D subsumes C (since D' subsumes C). □

2.3.3 The Subsumption Theorem (general case)

Finally we prove the Subsumption Theorem for arbitrary Σ and C. The following lemma shows that if we have derived some clause D from Σ which subsumes $C\theta$—where θ is a Skolem substitution for C w.r.t. Σ—then D also subsumes C. For instance, suppose $D = P(x)$, $C = P(y) \vee Q(z)$ and $\theta = \{y/a, z/b\}$. D subsumes $C\theta$, but since θ replaces each variable by a constant that does not appear in Σ, C or D, D also subsumes C itself.

Lemma 2.5 *Let C and D be clauses. Let $\theta = \{x_1/a_1, \ldots, x_n/a_n\}$ be a Skolem substitution for C w.r.t. D. If D subsumes $C\theta$, then D also subsumes C.*

Proof Since D subsumes $C\theta$, there exists a substitution σ such that $D\sigma \subseteq C\theta$. Let σ be the substitution $\{y_1/t_1, \ldots, y_m/t_m\}$. Let σ' be the substitution obtained from σ by replacing each a_i by x_i in every t_j. Note that $\sigma = \sigma'\theta$. Since θ only replaces each x_i by a_i $(1 \leq i \leq n)$, it follows that $D\sigma' \subseteq C$, so D subsumes C. □

Finally we can prove the general case of the Subsumption Theorem:

Theorem 2.3 (Subsumption Theorem) *Let Σ be a set of clauses and C be a clause. Then $\Sigma \models C$ iff $\Sigma \vdash_d C$.*

Proof

$\Leftarrow$: By the soundness of resolution and subsumption.

$\Rightarrow$: Assume C is not a tautology. Let θ be a Skolem substitution for C w.r.t. Σ. Then $C\theta$ is a ground clause which is not a tautology, and $\Sigma \models C\theta$. So by Lemma 2.4 there is a clause D such that $\Sigma \vdash_r D$ and D subsumes $C\theta$. Since D is derived from Σ, D does not contain any of the constants in θ. Therefore θ is also a Skolem substitution for C w.r.t. D. Then by Lemma 2.5, D subsumes C. Hence $\Sigma \vdash_d C$. □

2.4 The refutation-completeness

2.4.1 From Subsumption Theorem to refutation-completeness

The Subsumption Theorem actually tells us that resolution and subsumption form a complete set of derivation-rules for clauses. Though the resolution rule by itself is not complete for clauses in general, for instance, $P(x) \models P(a)$ but $P(x) \not\vdash_r P(a)$, resolution is complete w.r.t. unsatisfiable sets of clauses. This refutation-completeness is an easy consequence of the Subsumption Theorem:

Theorem 2.4 (Refutation-completeness) *Let Σ be a set of clauses. Then Σ is unsatisfiable iff $\Sigma \vdash_r \Box$.*

Proof

$\Leftarrow$: By the soundness of resolution.

$\Rightarrow$: Suppose Σ is unsatisfiable. Then $\Sigma \models \Box$. So by Theorem 2.3 there exists a clause D, such that $\Sigma \vdash_r D$ and D subsumes the empty clause $\Box$. But $\Box$ is the only clause which subsumes $\Box$, so $D = \Box$. $\Box$

2.4.2 From refutation-completeness to Subsumption Theorem

In the previous subsection, we showed that the refutation-completeness is a direct consequence of the Subsumption Theorem. Here we will show the converse: that we can obtain the Subsumption Theorem from the refutation-completeness. This establishes the equivalence of the Subsumption Theorem and the refutation-completeness: the one can be proved from the other.

To prove the Subsumption Theorem from the refutation-completeness, we will first show how to turn a refutation of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ into a deduction of $L_1 \vee \ldots \vee L_k$ from Σ. Thus our proof is constructive, and somewhat similar to the approach in [BM92]. We start with an example. Suppose $\Sigma = \{(P(x) \vee \neg R(f(f(b)))), (R(f(x)) \vee \neg R(x))\}$ and $C = P(x) \vee Q(x) \vee \neg R(b)$. First we note that $\theta = \{x/a\}$ is a Skolem substitution for C w.r.t. Σ. Now $\neg C\theta \Leftrightarrow \{\neg P(a), \neg Q(a), R(b)\}$. Figure 2.3 shows a refutation of $\Sigma \cup \{\neg P(a), \neg Q(a), R(b)\}$.

Now by omitting the leaves of the refutation-tree which come from $\neg C\theta$ (the framed literals) and by making appropriate changes in the tree, we get a derivation of the clause $D = P(x) \vee \neg R(b)$ (figure 2.4). D subsumes C, so we have turned the refutation of figure 2.3 into a deduction of C from Σ.

This approach also works in the general case. The following lemma does most of the work.

Lemma 2.6 *Let Σ be a set of clauses and $C = L_1 \vee \ldots \vee L_k$ be a non-tautologous ground clause. If $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_r \Box$, then $\Sigma \vdash_d C$.*

Proof Suppose $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_r \Box$. Then there exists a refutation $R_1, \ldots, R_n = \Box$ of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$. Let r be the number of resolvents in this sequence ($r = n-$ the number of members of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ in $R_1, \ldots, R_n$). We prove the lemma by induction on r.

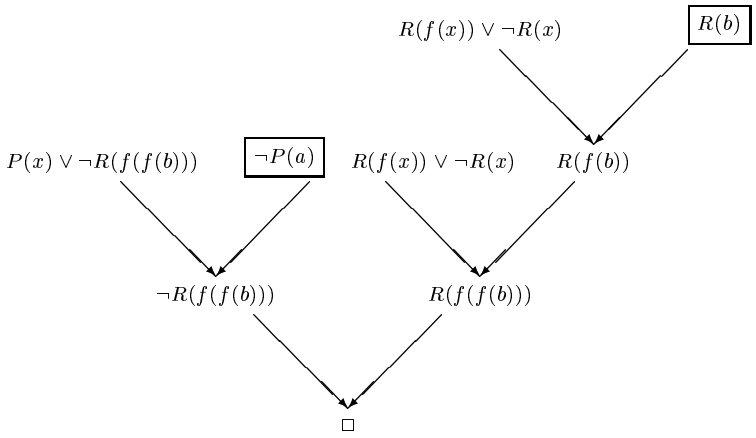


Figure 2.3: A refutation of $\Sigma \cup \{\neg P(a), \neg Q(a), R(b)\}$

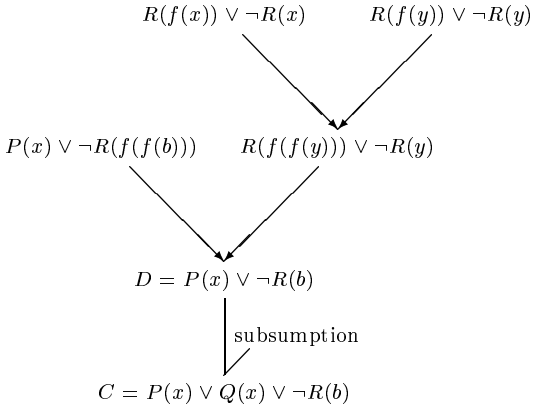


Figure 2.4: A deduction of C from Σ, obtained from the previous figure

1. Suppose $r = 0$. Then $R_n = \square \in \Sigma$, so the lemma holds.
2. Suppose the lemma holds for $r \leq m$. We will prove that this implies that the lemma also holds for $r = m + 1$. Let $R_1, \ldots, R_n = \square$ be a refutation of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ containing $m + 1$ resolvents. Let R_i be the first resolvent. Then $R_1, \ldots, R_n = \square$ is a refutation of $\Sigma \cup \{R_i\} \cup \{\neg L_1, \ldots, \neg L_k\}$ containing only m resolvents, since R_i is now one of the original premises. Hence by the induction hypothesis, there is a clause D, such that $\Sigma \cup \{R_i\} \vdash_r D$ and D subsumes C.
   Suppose R_i is itself a resolvent of two members of Σ. Then we also have $\Sigma \vdash_r D$, so the lemma holds in this case. Note that R_i cannot be a resolvent of two members of $\{\neg L_1, \ldots, \neg L_k\}$ because this set does not contain a complementary pair, since C is not a tautology.
   The only remaining case we have to check, is where R_i is a resolvent of $C' \in \Sigma$ and some $\neg L_s$ $(1 \leq s \leq k)$. Let $C' = M_1 \vee \ldots \vee M_j \vee \ldots \vee M_h$. Suppose R_i is a binary resolvent of $(M_1 \vee \ldots \vee M_j)\sigma$ (a factor of C', using σ as an mgu for $\{M_j, \ldots, M_h\}$) and $\neg L_s$, with θ as mgu for $M_j\sigma$ and L_s. Then $R_i = (M_1 \vee \ldots \vee M_{j-1})\sigma\theta$ and $C'\sigma\theta = R_i \vee L_s \vee \ldots \vee L_s$ ($h - j + 1$ copies of L_s), since $M_j, \ldots, M_h$ are all unified to L_s by $\sigma\theta$.
   Now replace each time R_i appears as leaf in the derivation-tree of D by

$C'\sigma\theta = R_i \vee L_s \vee \ldots \vee L_s$, and add $L_s \vee \ldots \vee L_s$ to all decendants of such an R_i-leaf. Then we obtain a derivation of $D \vee L_s \vee \ldots \vee L_s$ from $\Sigma \cup \{C'\sigma\theta\}$. Since $C'\sigma\theta$ is an instance of a clause from Σ, we can lift (by Lemma 2.3) this derivation to a derivation from Σ of a clause D', which has $D \vee L_s \vee \ldots \vee L_s$ as an instance. Since D subsumes C, D' also subsumes C. Hence $\Sigma \vdash_d C$.

□

Now we can prove the Subsumption Theorem (Theorem 2.3) once more, this time starting from Theorem 2.4.

Theorem 2.3 (Subsumption Theorem) *Let Σ be a set of clauses and C be a clause. Then $\Sigma \models C$ iff $\Sigma \vdash_d C$.*

Proof

$\Leftarrow$: By the soundness of resolution and subsumption.

$\Rightarrow$: If C is a tautology, the theorem is obvious. Assume C is not a tautology. Let θ be a Skolem substitution for C w.r.t. Σ. Suppose $C\theta = L_1 \vee \ldots \vee L_k$. Since C is not a tautology, $C\theta$ is not a tautology. $C\theta$ is ground and $\Sigma \models C\theta$, so by Proposition A.1 the set of clauses $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ is unsatisfiable. Then it follows from Theorem 2.4 that $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_r \square$. Therefore by Lemma 2.6, there exists a clause D such that $\Sigma \vdash_r D$ and D subsumes $C\theta$. Finally, from Lemma 2.5, D also subsumes C itself. Hence $\Sigma \vdash_d C$. □

Now that we have shown that the Subsumption Theorem can be proved from the refutation-completeness, and vice versa, we also have the following:

Theorem 2.5 *For unconstrained resolution, the Subsumption Theorem and the refutation-completeness are equivalent.*

2.5 Linear resolution

Linear resolution is characterized by the linear shape of its derivations. It is more efficient than unconstrained resolution, because the number of possible derivations is significantly decreased by the linear constraint on the shape of a derivation. It was independently introduced by Loveland [Lov70] and Luckham [Luc70]. An important further restriction called *SL-resolution* (Linear resolution with a Selection function) was introduced and shown to be refutation-complete by Kowalski and Kuehner [KK71]. Minicozzi and Reiter proved the Subsumption Theorem for linear resolution in [MR72]. More recently, Inoue [Ino92] developed SOL-resolution (Skip Ordered Linear resolution) and proved a version of the Subsumption Theorem for it.

2.5.1 Definitions

For the sake of transparency, we will define a very simple form of linear resolution here. Many features and restrictions could be added on to improve efficiency (see the references given above). We will prove the Subsumption Theorem and the refutation-completeness for this form of linear resolution. After that, we will define a further restriction of linear resolution called *input* resolution and show that this is *not* complete for general clauses, not even when the set of premises contains only one clause.

Definition 2.8 Let Σ be a set of clauses and C be a clause. A *linear derivation* of C from Σ is a finite sequence of clauses $R_0, \ldots, R_k = C$, such that $R_0 \in \Sigma$ and each R_i with $1 \leq i \leq k$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma \cup \{R_0, \ldots, R_{i-2}\}$.

R_0 is called the *top clause*, $R_0, \ldots, R_k$ the *center clauses*, and $C_1, \ldots, C_k$ are called the *side clauses* of this linear derivation. If a linear derivation of C from Σ exists, we write $\Sigma \vdash_{lr} C$.

A linear derivation of $\Box$ from Σ is called a *linear refutation* of Σ. $\diamond$

Linear derivations are characterized by the "linear" shape of their corresponding derivation-trees. See figure 2.5. Such a tree can be turned into a derivation-tree for unconstrained resolution by adding the derivations of each side clause C_i which is not in Σ.

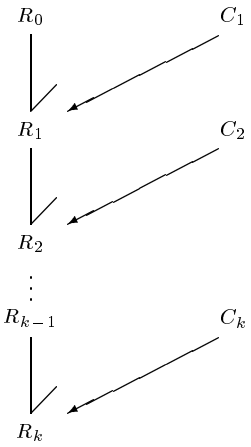


Figure 2.5: The characteristic shape of a linear derivation

Linear deductions are defined as follows:

Definition 2.9 Let Σ be a set of clauses and C a clause. There exists a *linear deduction* of C from Σ, written as $\Sigma \vdash_{ld} C$, if C is a tautology, or if there exists a clause D such that $\Sigma \vdash_{lr} D$ and D subsumes C. $\diamond$

2.5.2 The refutation-completeness

A proof of the refutation-completeness of a form of linear resolution called *OL-resolution* (Ordered Linear resolution), is given in Theorem 7.2 of [CL73].

However, this proof contains an error. In fact, OL-resolution is not refutation-complete, as described on pp. 324–325 of [Ino92]. Nevertheless, we can adapt the proof of [CL73] to yield a correct proof for our own definition of linear resolution. First we prove the case for ground clauses, which is then lifted. The proof of the following lifting lemma is similar to Lemma 2.3.

Lemma 2.7 (Linear derivation lifting) *Let Σ be a set of clauses and Σ' be a set of instances of clauses in Σ. Suppose $R'_0, \ldots, R'_k$ is a linear derivation of the clause R'_k from Σ'. Then there exists a linear derivation $R_0, \ldots, R_k$ of the clause R_k from Σ, such that R'_i is an instance of R_i, for each i.*

The next lemma is the refutation-completeness of linear resolution for ground clauses.

Lemma 2.8 *If Σ is an unsatisfiable set of ground clauses and $C \in \Sigma$ such that $\Sigma \backslash \{C\}$ is satisfiable, then there is a linear refutation of Σ with C as top clause.*

Proof By the Compactness Theorem (Theorem A.2), we can assume Σ is finite. Let n be the number of distinct ground atoms appearing in literals in clauses in Σ. We prove the lemma by induction on n.

1. If $n = 0$, then $\Sigma = \{\Box\}$. Since $\Sigma \backslash \{C\}$ is satisfiable, $C = \Box$
2. Suppose the lemma holds for $n \leq m$ and suppose $m + 1$ distinct atoms appear in Σ. We distinguish two cases.
   Case 1: Suppose $C = L$, where L is a literal. We first delete all clauses from Σ which contain the literal L (so we also delete C itself from Σ). Then we replace clauses which contain the literal $\neg L$ by clauses constructed by deleting these $\neg L$ (so for example, $L_1 \vee \neg L \vee L_2$ will be replaced by $L_1 \vee L_2$). Call the finite set obtained in this way ?. Note that the literal L, nor its negation, appears in clauses in ?. If M were a Herbrand model of ?, then $M \cup \{L\}$ would be a Herbrand model of Σ. Thus since Σ is unsatisfiable, ? must be unsatisfiable.
   Now let Σ' be an unsatisfiable subset of ?, such that every proper subset of Σ' is satisfiable. Σ' must contain a clause D' obtained from a member of Σ which contained $\neg L$, for otherwise the unsatisfiable set Σ' would be a subset of $\Sigma \backslash \{C\}$, contradicting the assumption that $\Sigma \backslash \{C\}$ is satisfiable. By construction of Σ', we have that $\Sigma' \backslash \{D'\}$ is satisfiable. Furthermore, Σ' contains at most m distinct atoms, so by the induction hypothesis there exists a linear refutation of Σ' with top clause D'. See the left of figure 2.6 for illustration.
   The side clauses in this refutation that are not previous center clauses, are either members of Σ or obtained from members of Σ by the deletion of $\neg L$. In the latter kind of side clauses, put back the deleted $\neg L$ literals, and add these $\neg L$ to all later center clauses. Note that afterwards, these center clauses may contain multiple copies of $\neg L$. In particular, the last center clause changes from $\Box$ to $\neg L \vee \ldots \vee \neg L$. Since D' is a resolvent of C and $D = \neg L \vee D' \in \Sigma$, we can add C and D as parent clauses on top of the previous top clause D'. That way, we get a linear derivation of $\neg L \vee \ldots \vee \neg L$ from Σ, with top clause C. Finally, the literals in

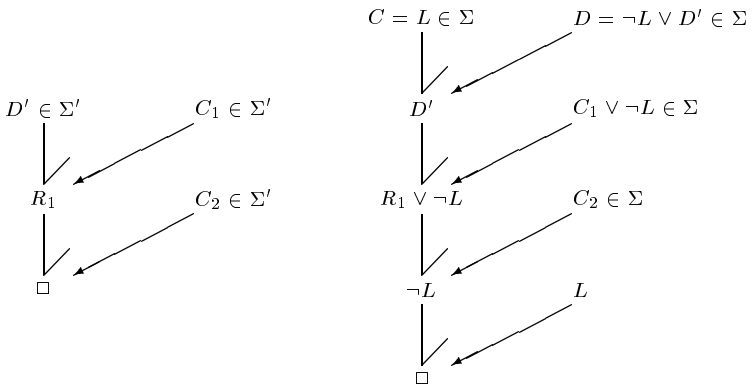


Figure 2.6: Case 1 of the proof

$\neg L \vee \ldots \vee \neg L$ can be resolved away using the top clause $C = L$ as side clause. This yields a linear refutation of Σ with top clause C (see the right of figure 2.6).

Case 2: Suppose $C = L \vee C'$, where C' is a non-empty clause. C' cannot contain $\neg L$, for otherwise C would be a tautology, contradicting the assumption that Σ is unsatisfiable while $\Sigma \backslash \{C\}$ is satisfiable.

Obtain Σ' from Σ by deleting clauses containing $\neg L$, and by removing the literal L from the remaining clauses. Note that $C' \in \Sigma'$. If M were an Herbrand model of Σ', then $M \cup \{\neg L\}$ would be an Herbrand model of Σ. Thus since Σ is unsatisfiable, Σ' is unsatisfiable.

Furthermore, because $\Sigma \backslash \{C\}$ is satisfiable, by Proposition A.4 there is an Herbrand model M' of $\Sigma \backslash \{C\}$. Since Σ is unsatisfiable, M' is not a model of C. L is a literal in C, hence L must be false under M'. Every clause in $\Sigma' \backslash \{C'\}$ is obtained from a clause in $\Sigma \backslash \{C\}$ by deleting L from it. Since M' is a model of every clause in $\Sigma \backslash \{C\}$ and L is false under M', every clause in $\Sigma' \backslash \{C'\}$ is true under M'. Therefore M' is a model of $\Sigma' \backslash \{C'\}$, which shows that $\Sigma' \backslash \{C'\}$ is satisfiable.

Then by the induction hypothesis, there exists a linear refutation of Σ' with top clause C'. Now similar to case 1, put back previously deleted L literals to the top and side clauses, and to the appropriate center clauses. This gives a linear derivation of $L \vee \ldots \vee L$ from Σ with top clause C.

Note that $\{L\} \cup (\Sigma \backslash \{C\})$ is unsatisfiable, because L is false in any Herbrand model of $\Sigma \backslash \{C\}$, as shown above. On the other hand, $\Sigma \backslash \{C\}$ is satisfiable. Thus by case 1 of this proof, there exists a linear refutation of $\{L\} \cup (\Sigma \backslash \{C\})$ with top clause L. Since L is a factor of $L \vee \ldots \vee L$, we can put our linear derivation of $L \vee \ldots \vee L$ "on top" of this linear refutation of $\{L\} \cup (\Sigma \backslash \{C\})$ with top clause L, thus obtaining a linear refutation of Σ with top clause C.

$\square$

Theorem 2.6 (Refutation-completeness of linear resolution) *Let Σ be a set of clauses. Then Σ is unsatisfiable iff $\Sigma \vdash_{lr} \square$.*

Proof

$\Leftarrow$: By the soundness of resolution.

$\Rightarrow$: Suppose Σ is unsatisfiable. Then by Theorem 2.1, there is a finite unsatisfiable set Σ_g of ground instances of clauses in Σ'. Let Σ'_g be an unsatisfiable subset of Σ_g and $C \in \Sigma'_g$ such that $\Sigma'_g \backslash \{C\}$ is satisfiable. From Lemma 2.8, we have $\Sigma'_g \vdash_{lr} \Box$. Hence $\Sigma \vdash_{lr} \Box$ by Lemma 2.7. $\Box$

2.5.3 The Subsumption Theorem

Starting from the refutation-completeness, it is now possible to prove also the Subsumption Theorem for linear resolution. Our proof is similar to the one given in [MR72]. We use the refutation-completeness and then turn a linear refutation into a linear deduction, using the following lemma:

Lemma 2.9 *Let Σ be a set of clauses and $C = L_1 \vee \ldots \vee L_k$ be a non-tautologous ground clause. If $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{lr} \Box$, then $\Sigma \vdash_{ld} C$.*

Proof Suppose $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{lr} \Box$. Then there exists a linear refutation $R_0, \ldots, R_n = \Box$ of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$. Notice that the top clause and the first side clause in this linear refutation cannot both be members of $\{\neg L_1, \ldots, \neg L_k\}$, because C is not a tautology. Thus we can assume $R_0 \in \Sigma$. It is then possible to prove by induction on n that this linear refutation can be transformed into a linear deduction of C from Σ with top clause R_0:

1. If $n = 0$, then $R_0 = \Box \in \Sigma$.
2. Suppose the lemma holds for $n \leq m$. Let $R_0, \ldots, R_{m+1} = \Box$ be a linear refutation of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$. Then $R_1, \ldots, R_{m+1}$ is a linear refutation of $\Sigma \cup \{R_1\} \cup \{\neg L_1, \ldots, \neg L_k\}$. By the induction hypothesis, there is a linear derivation of a clause D from $\Sigma \cup \{R_1\}$, with top clause R_1, such that D subsumes C.
   Suppose R_1 is itself a resolvent of two members of Σ. Then we also have $\Sigma \vdash_{lr} D$, so the lemma holds in this case.
   The only remaining case we have to check, is where R_1 is a resolvent of $R_0 \in \Sigma$ and some $\neg L_s$ $(1 \leq s \leq k)$. Let $R_0 = M_1 \vee \ldots \vee M_j \vee \ldots \vee M_h$. Suppose R_1 is a binary resolvent of $(M_1 \vee \ldots \vee M_j)\sigma$ (a factor of R_0, using σ as an mgu for $\{M_j, \ldots, M_h\}$) and $\neg L_s$, with θ as mgu for $M_j\sigma$ and L_s. Then $R_1 = (M_1 \vee \ldots \vee M_{j-1})\sigma\theta$ and $R_0\sigma\theta = R_1 \vee L_s \vee \ldots \vee L_s$ ($h - j + 1$ copies of L_s), since $M_j, \ldots, M_h$ are all unified to L_s by $\sigma\theta$.
   Now replace each time R_1 appears as leaf (i.e., top or side clause) in the derivation-tree of D by $R_0\sigma\theta = R_1 \vee L_s \vee \ldots \vee L_s$, and add $L_s \vee \ldots \vee L_s$ to all decendants of such an R_1-leaf. This gives a new derivation, in which each resolvent is the corresponding resolvent in the old derivation of D plus some extra copies of L_s. Thus we obtain a linear derivation of $D \vee L_s \vee \ldots \vee L_s$ from $\Sigma \cup \{R_0\sigma\theta\}$. Since $R_0\sigma\theta$ is an instance of a clause from Σ, we can lift (by Lemma 2.7) this derivation to a derivation from Σ of a clause D', which has $D \vee L_s \vee \ldots \vee L_s$ as an instance. Since D subsumes C, D' also subsumes C. Hence $\Sigma \vdash_{ld} C$. $\Box$

Theorem 2.7 (Subsumption Theorem for linear resolution) *Let Σ be a set of clauses and C be a clause. Then $\Sigma \models C$ iff $\Sigma \vdash_{ld} C$.*

Proof

$\Leftarrow$: By the soundness of resolution and subsumption.

$\Rightarrow$: If C is a tautology, the theorem is obvious. Assume C is not a tautology. Let θ be a Skolem substitution for C w.r.t. Σ. Let $C\theta$ be the clause $L_1 \vee \ldots \vee L_k$. Since C is not a tautology, $C\theta$ is not a tautology. $C\theta$ is ground and $\Sigma \models C\theta$, so the set of clauses $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ is unsatisfiable by Proposition A.1. Then it follows from Theorem 2.6 that $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{lr} \square$. Therefore by Lemma 2.9, there exists a clause D such that $\Sigma \vdash_{lr} D$ and D subsumes $C\theta$. From Lemma 2.5, D also subsumes C itself. Hence $\Sigma \vdash_{ld} C$. $\square$

We have now proved the Subsumption Theorem of linear resolution starting from the refutation-completeness of linear resolution. Conversely, the latter also follows immediately from the former, in the same way as Theorem 2.4 followed from Theorem 2.3 in the previous section. Hence also for linear resolution we have the equivalence between these two completeness results.

Theorem 2.8 *For linear resolution, the Subsumption Theorem and the refutation-completeness are equivalent.*

2.6 Input resolution

Linear resolution is a restriction of unconstrained resolution. Linear resolution can itself be further restricted to *input* resolution, by stipulating that each side clause should be a member of Σ. Contrary to linear resolution, input resolution is not complete, not even when the set of premises Σ contains only one clause. Before we give our counterexample, we will first formally define input resolution:

Definition 2.10 Let Σ be a set of clauses and C be a clause. An *input derivation* of C from Σ is a linear derivation in which each side clause C_i is a member of Σ. The side clauses $C_1, \ldots, C_k$ in an input derivation are also called *input clauses*. If an input derivation of C from Σ exists, we write $\Sigma \vdash_{ir} C$.

An input derivation of $\square$ from Σ is called an *input refutation* of Σ. $\diamond$

Definition 2.11 Let Σ be a set of clauses and C a clause. There exists an *input deduction* of C from Σ, written as $\Sigma \vdash_{id} C$, if C is a tautology, or if there exists a clause D such that $\Sigma \vdash_{ir} D$ and D subsumes C. $\diamond$

It is well-known that input resolution is not refutation-complete. A simple propositional example suffices to show this. Let $\Sigma = \{(P \vee Q), (P \vee \neg Q), (\neg P \vee Q), (\neg P \vee \neg Q)\}$. Figure 2.7 shows a refutation by unconstrained resolution of Σ. This proves that Σ is unsatisfiable.

Unfortunately, there does not exist an *input* refutation of Σ. It is easy to see the reason for this. To reach the empty clause $\square$, the last input clause in an input refutation of Σ should contain only one literal, or have a factor containing

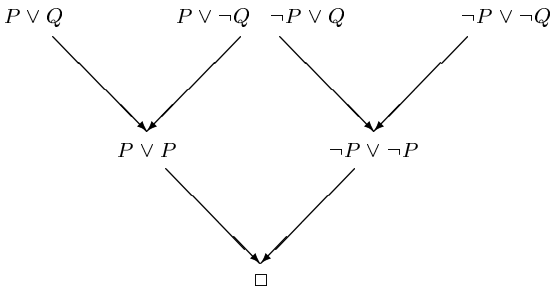


Figure 2.7: An unconstrained refutation of Σ

only one literal. However, each clause in Σ contains two distinct literals. Hence there is no input refutation of Σ.

So input resolution is not refutation-complete. This implies also that the Subsumption Theorem does not hold either for input resolution, since the refutation-completeness would be a direct consequence of it. We can in fact prove a stronger negative result, namely that the Subsumption Theorem for input resolution is not even true in the simple case where Σ contains only a single clause. In our counterexample we let $\Sigma = \{C\}$, where C is the following clause:

$$C = P(x_1, x_2) \vee Q(x_2, x_3) \vee \neg Q(x_3, x_4) \vee \neg P(x_4, x_1).$$

Figure 2.8 shows that clause D (see below) can be derived from C by unconstrained resolution. This also shows that $C \models D$.

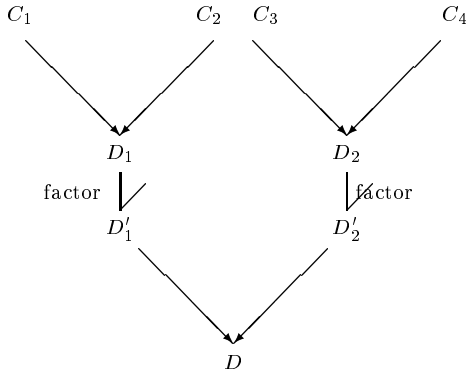


Figure 2.8: The derivation of D from C by unconstrained resolution

Figure 2.8 makes use of the clauses listed below. C_1, C_2, C_3, C_4 are variants of C. D_1 is a binary resolvent of C_1 and C_2, D_2 is a binary resolvent of C_3 and C_4 (the underlined literals are the literals resolved upon). D_1' is a factor of D_1, using the substitution $\{x_5/x_1, x_6/x_2\}$. D_2' is a factor of D_2, using $\{x_{11}/x_{12}, x_{13}/x_9\}$. Finally, D is a binary resolvent of D_1' and D_2'.

$$C_1 = P(x_1, x_2) \vee \underline{Q(x_2, x_3)} \vee \neg Q(x_3, x_4) \vee \neg P(x_4, x_1).$$

$C_2 = P(x_5, x_6) \vee Q(x_6, x_7) \vee \underline{\neg Q(x_7, x_8)} \vee \neg P(x_8, x_5)$.
$C_3 = P(x_9, x_{10}) \vee \underline{Q(x_{10}, x_{11})} \vee \neg Q(x_{11}, x_{12}) \vee \neg P(x_{12}, x_9)$.
$C_4 = P(x_{13}, x_{14}) \vee Q(x_{14}, x_{15}) \vee \underline{\neg Q(x_{15}, x_{16})} \vee \neg P(x_{16}, x_{13})$.
$D_1 = P(x_1, x_2) \vee \neg Q(x_3, x_4) \vee \neg \underline{P(x_4, x_1)} \vee P(x_5, x_6) \vee Q(x_6, x_2) \vee \neg P(x_3, x_5)$.
$D_2 = P(x_9, x_{10}) \vee \neg Q(x_{11}, x_{12}) \vee \neg P(x_{12}, x_9) \vee P(x_{13}, x_{14}) \vee Q(x_{14}, x_{10}) \vee$
$\neg P(x_{11}, x_{13})$.
$D_1' = \underline{P(x_1, x_2)} \vee \neg Q(x_3, x_4) \vee \neg P(x_4, x_1) \vee Q(x_2, x_2) \vee \neg P(x_3, x_1)$.
$D_2' = P(x_9, x_{10}) \vee \neg Q(x_{12}, x_{12}) \vee \underline{\neg P(x_{12}, x_9)} \vee P(x_9, x_{14}) \vee Q(x_{14}, x_{10})$.
$D = \neg Q(x_3, x_4) \vee \neg P(x_4, x_1) \vee Q(x_2, x_2) \vee \neg P(x_3, x_1) \vee P(x_2, x_{10}) \vee$
$\neg Q(x_1, x_1) \vee P(x_2, x_{14}) \vee Q(x_{14}, x_{10})$.

Thus D can be derived from C using unconstrained resolution. However, neither D nor a clause which subsumes D can be derived from C using only *input* resolution. We prove this in Proposition 2.1. This shows that input resolution is not complete, not even if Σ contains only one clause.

The following lemma shows that each clause which can be derived from C by input resolution contains an instance of $P(x_1, x_2) \vee \neg P(x_4, x_1)$ or an instance of $Q(x_2, x_3) \vee \neg Q(x_3, x_4)$.

Lemma 2.10 *Let C be as defined above. If $C \vdash_{ir} E$, then E contains an instance of $P(x_1, x_2) \vee \neg P(x_4, x_1)$ or an instance of $Q(x_2, x_3) \vee \neg Q(x_3, x_4)$.*

Proof Let $R_0, \ldots, R_k = E$ be an input derivation of E from C. We prove the lemma by induction on k:

1. $R_0 = C$, so the lemma is obvious if $k = 0$.
2. Suppose the lemma holds for $k \leq n$. Let $R_0, \ldots, R_{n+1} = E$ be an input derivation of E from C. Note that the only factor of C is C itself. Therefore E is a binary resolvent of C and a factor of R_m. Let θ be the mgu used in obtaining this binary resolvent. If $P(x_1, x_2)$ or $\neg P(x_4, x_1)$ is the literal resolved upon in C, then E must contain $(Q(x_2, x_3) \vee \neg Q(x_3, x_4))\theta$. Otherwise $Q(x_2, x_3)$ or $\neg Q(x_3, x_4)$ is the literal resolved upon in C, so then E contains $(P(x_1, x_2) \vee \neg P(x_4, x_1))\theta$. Hence the lemma also holds for $k = n + 1$.

□

Proposition 2.1 *Let C and D be as defined above. Then $C \nvdash_{id} D$.*

Proof Suppose $C \vdash_{id} D$. Then since D is not a tautology, there exists a clause E such that $C \vdash_{ir} E$ and E subsumes D. From Lemma 2.10 we know that E contains an instance of $P(x_1, x_2) \vee \neg P(x_4, x_1)$ or an instance of $Q(x_2, x_3) \vee \neg Q(x_3, x_4)$. It is easy to see that neither $P(x_1, x_2) \vee \neg P(x_4, x_1)$ nor $Q(x_2, x_3) \vee \neg Q(x_3, x_4)$ subsumes D. But then E does not subsume D, so we found a contradiction. Hence $C \nvdash_{id} D$. □

So we see that input resolution is not complete: $C \models D$, but $C \not\vdash_{id} D$. This is unfortunate, since input resolution is more efficient than unconstrained resolution or linear resolution. However, if we restrict ourselves to *Horn clauses*, a special case of input resolution called *SLD-resolution* can be shown to be complete. This will be the topic of the next section.

2.7 SLD-resolution

SLD-resolution for Horn clauses was introduced by Kowalski [Kow74]. It is simpler than the unconstrained or linear resolution that we need for general clauses.

Definition 2.12 Let Σ be a set of Horn clauses and C be a Horn-clause. An *SLD-derivation* of C from Σ is a finite sequence of Horn clauses $R_0, \ldots, R_k = C$, such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a binary resolvent of R_{i-1} and a definite program clause $C_i \in \Sigma$, using the head of C_i and a *selected atom* in the body of R_{i-1} as the literals resolved upon.

R_0 is called the *top clause* and the C_i are the *input clauses* of this SLD-derivation. If an SLD-derivation of C from Σ exists, we write $\Sigma \vdash_{sr} C$. An SLD-derivation of $\Box$ from Σ is called an *SLD-refutation* of Σ. $\diamond$

Note that either each R_i in an SLD-derivation is a goal, or each R_i is a definite program clause. Also note that each resolvent in an SLD-derivation is a *binary* resolvent, so no factors are used here. The selected atom can be selected by a so-called *computation rule*, and it can be shown that the refutation-completeness of SLD-resolution is independent of the computation rule that is used. We will not go into that here (see [Llo87]).

Definition 2.13 Let Σ be a set of Horn clauses and C a Horn clause. There exists an *SLD-deduction* of C from Σ, written as $\Sigma \vdash_{sd} C$, if C is a tautology, or if there is a Horn clause D, such that $\Sigma \vdash_{sr} D$ and D subsumes C. $\diamond$

Example 2.4 Consider $\Sigma = \{P(0, x, x), (P(s(x), y, s(z)) \leftarrow P(x, y, z))\}$, a set of clauses which formalizes addition. Let us see how we can prove $C = P(s^2(0), s(0), s^3(0))$ (that is, $2 + 1 = 3$) from this set by SLD-resolution. Figure 2.9 shows an SLD-derivation of $R_2 = P(s^2(0), y, s^2(y))$ from Σ. Here the selected atoms are underlined. Since R_2 subsumes C, we have $\Sigma \vdash_{sd} C$.

$\lhd$

2.7.1 The refutation-completeness

In this subsection, we will prove the well-know result that SLD-resolution is refutation-complete: a set of Horn clauses is unsatisfiable iff it has an SLD-refutation. Our proof is similar to the proof for the refutation-completeness of linear resolution that we gave in Section 2.5. It is different from the proof given in [Llo87], since our proof does not require fixed-point theory. Instead, it only uses the basic definitions of resolution. First we establish the refutation-completeness for ground Horn clauses:

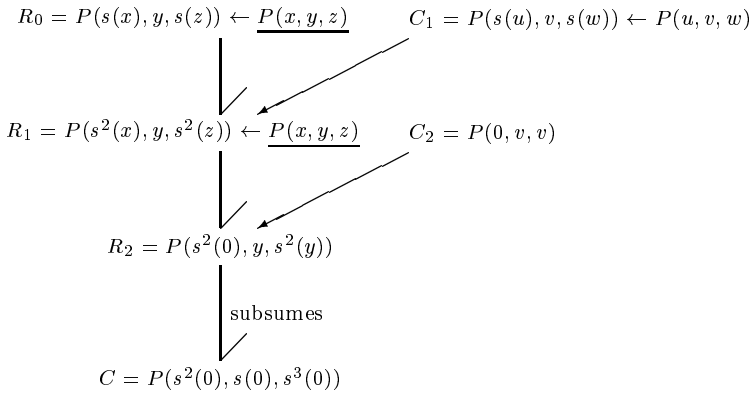


Figure 2.9: An SLD-deduction of C from Σ

Lemma 2.11 *If Σ is a finite unsatisfiable set of ground Horn clauses, then $\Sigma \vdash_{sr} \square$.*

Proof Let n be the number of atomic clauses (clauses which consist of a single positive literal) in Σ. The proof is by induction on n.

1. If $n = 0$, then $\square \in \Sigma$, for otherwise the empty set would be an Herbrand model of Σ.
2. Suppose the lemma holds for $0 \leq n \leq m$. Suppose Σ contains $m + 1$ atomic clauses. If $\square \in \Sigma$ the lemma is obvious, so suppose $\square \notin \Sigma$.
   Let A be an atomic clause in Σ. We first delete all clauses from Σ which have A as head (so we also delete the atomic clause A from Σ). Then we replace clauses which have A in their body by clauses constructed by deleting these atoms A from the body (so for example, $B \leftarrow A, B_1, \ldots, B_k$ will be replaced by $B \leftarrow B_1, \ldots, B_k$). Call the set obtained in this way Σ'.
   If M were a Herbrand model of Σ', then $M \cup \{A\}$ would be a Herbrand model of Σ. Thus since Σ is unsatisfiable, Σ' must be unsatisfiable. Σ' only contains m atomic clauses, so by the induction hypothesis, there is an SLD-refutation of Σ'. If this refutation only uses clauses from Σ' which were also in Σ, then this is also an SLD-refutation of Σ, so then we are done.
   Otherwise, if C is the top clause or an input clause in this refutation and $C \notin \Sigma$, then C was obtained from some $C' \in \Sigma$ by deleting all atoms A from the body of C'. For all such C, do the following: restore the previously deleted copies of A to the body of C (which turns C into C' again), and add these atoms A to all later resolvents. This way, we can turn the SLD-refutation of Σ' into an SLD-derivation of $\leftarrow A, \ldots, A$ from Σ. (See figure 2.10 for illustration, where we add previously deleted atoms A to the bodies of R_0 and C_2.) Since also $A \in \Sigma$, we can construct an SLD-refutation of Σ, using A a number of times as input clause to resolve away all members of the goal $\leftarrow A, \ldots, A$.

□

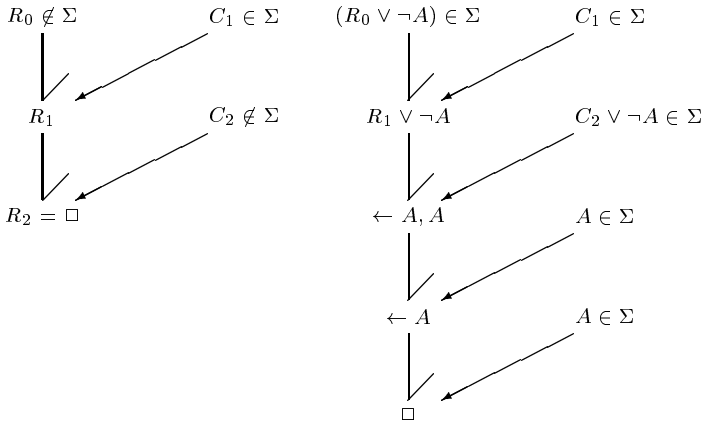


Figure 2.10: The SLD-refutations of Σ' (left) and Σ (right)

The proof of the lifting lemma for SLD-resolution is similar to Lemma 2.3.

Lemma 2.12 (SLD-derivation lifting) *Let Σ be a set of Horn clauses and Σ' be a set of instances of clauses in Σ. Suppose $R'_0, \ldots, R'_k$ is an SLD-derivation of the clause R'_k from Σ'. Then there exists an SLD-derivation $R_0, \ldots, R_k$ of the clause R_k from Σ, such that R'_i is an instance of R_i, for each i.*

The previous lemmas allow us to prove the refutation-completeness of SLD-resolution:

Theorem 2.9 (Refutation-completeness of SLD-resolution) *Let Σ be a set of Horn clauses. Then Σ is unsatisfiable iff $\Sigma \vdash_{sr} \square$.*

Proof

$\Leftarrow$: By the soundness of resolution.

$\Rightarrow$: Suppose Σ is unsatisfiable. By Theorem 2.1, there is a finite unsatisfiable set Σ' of ground instances of clauses in Σ. From Lemma 2.11, we have $\Sigma' \vdash_{sr} \square$. Using Lemma 2.12, we can lift this to $\Sigma \vdash_{sr} \square$. $\square$

2.7.2 The Subsumption Theorem

Here we will prove the Subsumption Theorem for SLD-resolution. As in the case of linear resolution, we establish this result by translating a refutation to a deduction, using the following lemma:

Lemma 2.13 *Let Σ be a set of Horn clauses and $C = L_1 \vee \ldots \vee L_k$ be a non-tautologous ground Horn clause. If $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{sr} \square$, then $\Sigma \vdash_{sd} C$.*

Proof Suppose $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{sr} \square$, that is, there exists an SLD-refutation $R_0, \ldots, R_n = \square$ of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$. By induction on n:

1. If $n = 0$, then $R_0 = \square \in \Sigma$, so then the lemma is obvious.

2. Suppose the lemma holds for $n \leq m$. Let $R_0, \ldots, R_{m+1} = \square$ be an SLD-refutation of $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$. Then $R_1, \ldots, R_{m+1}$ is an SLD-refutation of $\Sigma \cup \{R_1\} \cup \{\neg L_1, \ldots, \neg L_k\}$. By the induction hypothesis, there is an SLD-derivation $R'_1, R'_2, \ldots, R'_l$ from $\Sigma \cup \{R_1\}$, where R'_l subsumes C. Note that R_1 must be a definite goal, so R_1 can only be used as top clause in this derivation.
   If $R'_1 \neq R_1$, then $R'_1 \in \Sigma$. Moreover, in that case R_1 is used nowhere in the SLD-derivation of R'_l, so then this is an SLD-derivation of R'_l from Σ, and hence $\Sigma \vdash_{sd} C$. In case $R'_1 = R_1$, we distinguish three possibilities:
   1. R_1 is a binary resolvent of a goal $G \in \Sigma$ and a definite clause $C_1 \in \Sigma$. Then $G, R'_1, R'_2, \ldots, R'_l$, with C_1 as first input clause, is an SLD-derivation from Σ. R'_l subsumes C, so then $\Sigma \vdash_{sd} C$.
   2. R_1 is a binary resolvent of a negative literal $\neg L \in \{\neg L_1, \ldots, \neg L_k\}$ and a definite clause $C_1 \in \Sigma$ (note that this means that C is a definite program clause, with L as head). Let θ be the mgu used in this resolution-step, so $C_1\theta = L \vee R_1$. Then $C_1\theta, L \vee R'_2, \ldots, L \vee R'_l$ is an SLD-derivation of $L \vee R'_l$ from $\Sigma \cup \{C_1\theta\}$. (See Figure 2.11 for illustration.) $C_1\theta$ is an instance of a clause in Σ, so by Lemma 2.12, we can find an SLD-derivation from Σ of a clause D, of which $L \vee R'_l$ is an instance. Since R'_l subsumes C and $L \in C$, $L \vee R'_l$ subsumes C, and hence D also subsumes C. Therefore $\Sigma \vdash_{sd} C$.

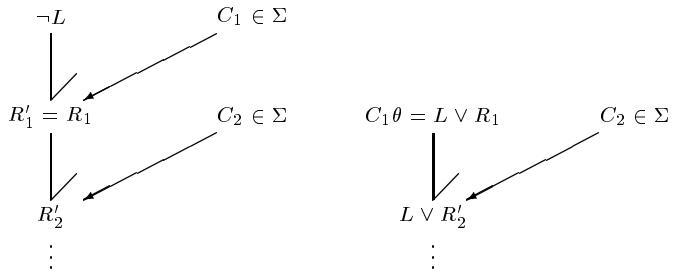


Figure 2.11: Illustration of case 2 of the proof

   3. R_1 is a binary resolvent of a goal $G \in \Sigma$ and a positive literal $L \in \{\neg L_1, \ldots, \neg L_k\}$. Let θ be the mgu used in this resolution step, so $G\theta = \neg L \vee R_1$. Then $G\theta = \neg L \vee R'_1, \neg L \vee R'_2, \ldots, \neg L \vee R'_l$ is an SLD-derivation of $\neg L \vee R'_l$ from $\Sigma \cup \{G\theta\}$. $G\theta$ is an instance of a clause in Σ, so by Lemma 2.12, we can find an SLD-derivation from Σ of a clause D, of which $\neg L \vee R'_l$ is an instance. Since R'_l subsumes C and $\neg L \in C$, $\neg L \vee R'_l$ subsumes C, and hence D also subsumes C. Therefore $\Sigma \vdash_{sd} C$.

□

Now we can prove the Subsumption Theorem for SLD-resolution. This result generalizes Theorem 18 of [MP94], which gives the theorem for the case where Σ contains only one clause (though ignoring that C may be a tautology).

Theorem 2.10 (Subsumption Theorem for SLD-resolution) *Let Σ be a set of Horn clauses and C be a Horn clause. Then $\Sigma \models C$ iff $\Sigma \vdash_{sd} C$.*

Proof

$\Leftarrow$: By the soundness of resolution and subsumption.

$\Rightarrow$: If C is a tautology, the theorem is obvious. Assume C is not a tautology. Let θ be a Skolem substitution for C w.r.t. Σ. Let $C\theta$ be the clause $L_1 \vee \ldots \vee L_k$. Since C is not a tautology, $C\theta$ is not a tautology. $C\theta$ is ground and $\Sigma \models C\theta$, so by Proposition A.1 the set of clauses $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\}$ is unsatisfiable. Then it follows from Theorem 2.9 that $\Sigma \cup \{\neg L_1, \ldots, \neg L_k\} \vdash_{sr} \Box$. Therefore by Lemma 2.13, there exists a clause D such that $\Sigma \vdash_{sr} D$ and D subsumes $C\theta$. From Lemma 2.5, D also subsumes C itself. Hence $\Sigma \vdash_{sd} C$. $\Box$

Note the following special case of this result: if Π is a definite program and A is an atom such that $\Pi \models A$, then there exists an atom B such that $\Pi \vdash_{sr} B$ and A is an instance of B.

Furthermore, analogous to the case of linear resolution, we also have the following equivalence:

Theorem 2.11 *For SLD-resolution, the refutation-completeness and the Subsumption Theorem are equivalent.*

2.8 Summary

The Subsumption Theorem is the following statement:

> If Σ is a set of clauses and C is a clause, then $\Sigma \models C$ iff C is a tautology, or there exists a clause D which subsumes C and which can be derived from Σ by some form of resolution.

This theorem is a more direct form of completeness than the better-known refutation-completeness of resolution and hence sometimes more useful, particularly for theoretical analysis.

Different versions of the theorem exist, depending on the instantiation of "some form of resolution." We have proved here that the Subsumption Theorem holds for unconstrained resolution and linear resolution for general clauses. Moreover, for each of these two forms of resolution, the Subsumption Theorem is equivalent to the refutation-completeness of that form of resolution: the one can be proved from the other. On the other hand, the Subsumption Theorem does not hold for input resolution, not even in the simple case where Σ contains only one clause. For SLD-resolution for Horn clauses, the Subsumption Theorem does hold, and is again equivalent to the refutation-completeness of SLD-resolution.

Chapter 3

Unfolding

3.1 Introduction

In an ILP-problem, it is sometimes the case that we initially start with a theory that is overly general: it is complete, but not consistent. The problem of finding a correct theory then becomes the problem of *specializing* the initial theory to a correct one. In this chapter we will investigate how such specialization can be done using *unfolding*. This is a specialization-operator which constructs resolvents from given parent clauses. We will restrict attention to definite program clauses, so the theories should be definite programs. Furthermore, we will also assume that the given examples E^+ and E^- consist of ground atoms (ground instances of one or more predicates).

Let us first formally define the *specialization problem*:

> **Given**: A definite program Π and two disjoint sets of ground atoms E^+ and E^-, such that Π is overly general w.r.t. E^+ and E^-, and suppose there exists a definite program Π' such that $\Pi \models \Pi'$ and Π' is correct w.r.t. E^+ and E^-.
> **Find**: One such a Π'.

Clearly, this is a special case of the general problem setting of Chapter 1. We need to presuppose the existence of a correct specialization Π' of Π, because a correct program does not always exist, as proved in Theorem 1.1. Hence trying to solve a specialization problem only makes sense when a correct specialization exists. Note that background knowledge can be included in Π, so we will not mention background knowledge separately in this chapter.

A natural way to specialize Π is, first, to replace a clause in Π by all its resolvents upon some body-atom in this clause. Constructing these resolvents is called unfolding. The new program obtained in this way after unfolding a clause in Π, is clearly implied by Π. The function of the replaced clause is taken over by the set of resolvents produced by unfolding. We can then, secondly, delete some new clauses from the program that have to do with the negative examples, thus specializing the program. Hopefully, after repeating these two steps a number of times, we can get rid of all negative examples. This method was introduced in [BIA94].

For simplicity, let all examples be ground instances of $P(x_1, \ldots, x_n)$, for some predicate P. The motivation for the method described above, is the fact that it can be used to prune negative examples from the SLD-tree for $\Pi \cup \{\leftarrow P(x_1, \ldots, x_n)\}$.[1] We will illustrate this by an example. Consider the program Π, consisting of the following clauses:

$C_1 = P(x, y) \leftarrow Q(x, y)$
$C_2 = Q(b, b) \leftarrow Q(a, a)$
$C_3 = Q(a, a)$

and $E^+ = \{P(b, b)\}$, $E^- = \{P(a, a)\}$. The SLD-tree for $\Pi \cup \{\leftarrow P(x, y)\}$ is shown on the left of figure 3.1. The success branches corresponding to refutations of positive examples are marked with a '+', for negative examples with a '−'.

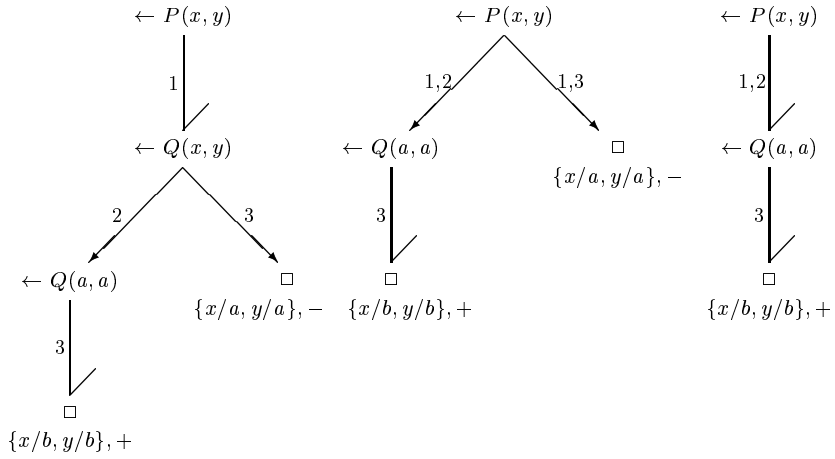


Figure 3.1: The SLD-trees for Π, Π' and Π''

$P(a, a)$ is a negative example, so we would like to remove this by weakening the program. This could be done by deleting C_1 or C_3 from Π. However, this would also make the positive example $P(b, b)$ no longer derivable, thus rendering the program too weak. Another way to specialize is, first, to unfold C_1 upon $Q(x, y)$. The following $C_{1,2}$ and $C_{1,3}$ are the two clauses produced by unfolding C_1.

$C_{1,2} = P(b, b) \leftarrow Q(a, a)$ (resolvent of C_1 and C_2)
$C_{1,3} = P(a, a)$ (resolvent of C_1 and C_3)

Now we replace the unfolded clause C_1 by its resolvents $C_{1,2}$ and $C_{1,3}$. This results in $\Pi' = \{C_2, C_3, C_{1,2}, C_{1,3}\}$. The SLD-tree for $\Pi' \cup \{\leftarrow P(x, y)\}$ is shown in the middle of figure 3.1. In this tree, the negative example is directly connected to the root, via the branch that uses $C_{1,3}$. Now the negative example can be pruned from the tree by deleting $C_{1,3}$ from Π', which does not affect the

[1] An SLD-tree for $\Pi \cup \{G\}$ is a tree containing all SLD-derivations from $\Pi \cup \{G\}$ with the goal G as top clause, in which the selected atoms are selected by some computation rule. See [Llo87] for more information on SLD-trees.

positive example. Then we obtain $\Pi'' = \{C_2, C_3, C_{1,2}\}$, which is correct w.r.t. E^+ and E^-. The SLD-tree for $\Pi'' \cup \{\leftarrow P(x, y)\}$ is simply the tree for Π', after the rightmost branch has been pruned (right of figure 3.1).

The idea behind this method is the following:

1. Unfolding removes some internal nodes from the SLD-tree, for instance, the internal node $\leftarrow Q(x, y)$ in the tree on the left of figure 3.1. This tends to separate the positive from the negative examples and also brings them closer to the root of the tree.
2. If a negative example hangs directly from the root and its input clause C is not used elsewhere in the tree for a positive example, then the program can be specialized by deleting C.

In other words: unfolding can transform the SLD-tree in such a way that negative examples can be pruned by deleting clauses from the program, without also pruning positive examples.[2] Thus the use of unfolding as a specialization tool can be motivated by looking at SLD-trees and the SLD-refutations those trees contain.

In this chapter we first define UD_1-specialization and UD_2-specialization, which employ unfolding (each in their own way) and clause deletion. It will be seen from some examples that we give later on, that both of these specialization methods are incomplete: some specialization problems cannot be solved in this way. However, if we look at program specialization through the perspective of SLD-derivations rather than refutations, then we can see from the Subsumption Theorem for SLD-resolution that *subsumption* is what we need to make our specialization technique complete. Thus in Section 3.5, we define UDS-specialization, a specialization technique based on **U**nfolding, clause **D**eletion and **S**ubsumption. We prove that UDS-specialization is complete: every specialization problem has a UDS-specialization as a solution. Finally, in Section 3.6 we go into the relation between program specialization by unfolding and program generalization by inverse resolution.

3.2 Unfolding

In this section, we define unfolding, which will be used in the next sections to solve specialization problems.

Definition 3.1 Let Π be a definite program, $C = A \leftarrow B_1, \ldots, B_n$ a definite program clause in Π and B_i the i-th atom in the body of C. Let $\{C_1, \ldots, C_m\}$ be the set of clauses in Π whose head can be unified with B_i. Then *unfolding* C *upon* B_i *in* Π means constructing the set $U_{C,i} = \{D_1, \ldots, D_m\}$, where each D_j is the resolvent of C_j and C, using B_i and the head of C_j as the literals resolved upon. $\diamond$

[2] In [BIA94, Bos95a], Boström and Idestam-Almquist present the algorithm SPECTRE, which implements this specialization technique for single-predicate learning. In [Bos95b], SPECTRE II is presented, which overcomes some difficulties of SPECTRE concerning recursive clauses and which can be applied to multiple-predicate learning. Unfolding was also implemented in [AGB95], combined with a version of Shapiro's Backtracing Algorithm [Sha81b].

Example 3.1 Let Π consist of the following clauses:

$C_1 = P(f(x)) \leftarrow P(x), Q(x)$
$C_2 = Q(x) \leftarrow R(x, a)$
$C_3 = P(f(a))$
$C_4 = Q(b)$

Suppose we want to unfold C_1 upon $Q(x)$ in the program Π. $\{C_2, C_4\}$ is the set of clauses in Π whose head can be unified with $Q(x)$, so $U_{C_1,2} = \{(P(f(x)) \leftarrow P(x), R(x, a)), (P(f(b)) \leftarrow P(b))\}$. $\lhd$

Note that $U_{C,i}$ may be the empty set. This is the case if there is no program clause whose head unifies with the i-th atom in the body of C. Note also that an atom cannot be unfolded, since it has no body-atoms.

Using the set $U_{C,i}$, we can construct a new program from Π in two ways. The first way, used in [BIA94], *replaces* C by $U_{C,i}$, thus obtaining the program $(\Pi \backslash \{C\}) \cup U_{C,i}$. The second way *adds* $U_{C,i}$ to Π, without deleting the unfolded clause C from the program.

Definition 3.2 Let Π be a definite program and $U_{C,i}$ the set of clauses constructed by unfolding C upon B_i in Π. Then $\Pi_{u1,C,i} = (\Pi \backslash \{C\}) \cup U_{C,i}$ is called the *type 1 program* resulting from unfolding C upon B_i in Π. $\Pi_{u2,C,i} = \Pi \cup U_{C,i}$ is called the *type 2 program* resulting from unfolding C upon B_i in Π. $\diamond$

In the next sections, we will see how these two types of unfolding can be used for program specialization. Here we will first show that constructing the type 1 program preserves the least Herbrand model of the program, while constructing the type 2 program preserves logical equivalence, which is stronger.

Proposition 3.1 *Let Π be a definite program, G a definite goal and $\Pi_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in Π. Then $\Pi \cup \{G\} \vdash_{sr} \Box$ iff $\Pi_{u1,C,i} \cup \{G\} \vdash_{sr} \Box$.*

Proof

$\Leftarrow$: Suppose $\Pi_{u1,C,i} \cup \{G\} \vdash_{sr} \Box$. Then by the soundness of resolution, $\Pi_{u1,C,i} \cup \{G\}$ is unsatisfiable. It is easy to see that $\Pi \models \Pi_{u1,C,i}$. Hence $\Pi \cup \{G\}$ is unsatisfiable, and by Theorem 2.9, we have $\Pi \cup \{G\} \vdash_{sr} \Box$.

$\Rightarrow$: Suppose $\Pi \cup \{G\} \vdash_{sr} \Box$ and C (the unfolded clause) is $A \leftarrow B_1, \ldots, B_i, \ldots, B_n$, which we abbreviate to $A \leftarrow \overline{B_1}, B_i, \overline{B_2}$ (where $\overline{B_1} = B_1, \ldots, B_{i-1}$ and $\overline{B_2} = B_{i+1}, \ldots, B_n$). B_i is the atom unfolded upon. If there is an SLD-refutation of $\Pi \cup \{G\}$ in which C isn't used as an input clause, then this is also an SLD-refutation of $\Pi_{u1,C,i} \cup \{G\}$. But suppose C is used as input clause in all SLD-refutations of $\Pi \cup \{G\}$. We will prove that from such a refutation, a refutation of $\Pi_{u1,C,i} \cup \{G\}$ can be constructed.

Suppose we have a refutation of $\Pi \cup \{G\}$ with goals $G_0, \ldots, G_n$ and input clauses $C_1, \ldots, C_n$, which uses C at least once as input clause. By the independence of the computation rule (Theorem 9.2 of [Llo87]), we can assume that for any k, if C is the input clause in the step leading from G_{k-1} to G_k, then the instance of B_i that is inserted in G_k by C, is the selected atom in G_k.

Suppose the j-th input clause is C. We picture this part of the refutation on the left of figure 3.2. Here we make the following notational conventions:

- G_{j-1}, the $(j-1)$-th goal, is the goal $\leftarrow A_1, \ldots, A_k, \ldots, A_m$, which we abbreviate to $\leftarrow \overline{A_1}, A_k, \overline{A_2}$.
- The input clause used in the $(j+1)$-th step is $C_{j+1} = A' \leftarrow \overline{B'}$, where $\overline{B'}$ is an abbreviation of $B'_1, \ldots, B'_r$.
- θ_j is an mgu for A_k and A (used in the j-th resolution step).
- θ_{j+1} is an mgu for $B_i\theta_j$ and A' (used in the $(j+1)$-th resolution step).

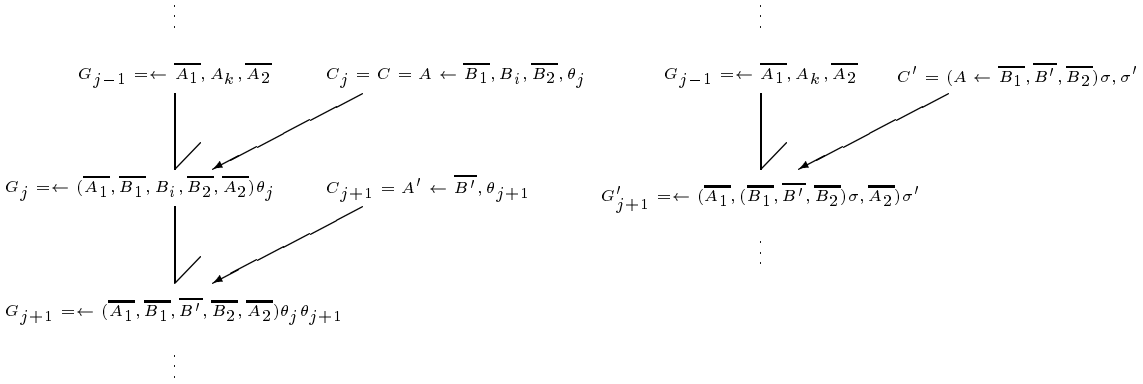


Figure 3.2: From the tree on the left, we can construct the tree on the right, using C' instead of C.

Since the $(j+1)$-th step of the tree on the left of figure 3.2 shows that B_i and A' can be unified (say, with mgu σ), the clause $C' = (A \leftarrow \overline{B_1}, \overline{B'}, \overline{B_2})\sigma$ (the result of resolving C with $C_{j+1} = A' \leftarrow \overline{B'}$) must be in $U_{C,i}$. We assume without loss of generality that G_{j-1}, $C_j = C$, C_{j+1}, and C' are standardized apart.

What we want is to construct a tree which, instead of using C in the j-th step, uses C'. For this, we will show that G_{j+1} is a variant of the goal G'_{j+1}, which can be derived from G_{j-1} and C'. Then we can replace the j-th step (which uses C) and the $(j+1)$-th step by one single step which doesn't need C anymore, but instead uses C'.

θ_{j+1} is an mgu for A' and $B_i\theta_j$ and $A'\theta_j = A'$ (because of the standardizing apart), so $\theta_j\theta_{j+1}$ is a unifier for A' and B_i. σ is an mgu for A' and B_i, so there exists a substitution γ such that $\sigma\gamma = \theta_j\theta_{j+1}$. $A\sigma\gamma = A\theta_j\theta_{j+1} = A_k\theta_j\theta_{j+1} = A_k\sigma\gamma = A_k\gamma$, so γ is a unifier for $A\sigma$ and A_k. This shows that $A\sigma$ and A_k can be unified. Let σ' be an mgu for $A\sigma$ and A_k. Let $G'_{j+1} = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma'$ be the goal derived from G_{j-1} and C'. We will show that G_{j+1} and G'_{j+1} are variants.

1. We have already shown that γ is a unifier for $A\sigma$ and A_k. Furthermore, σ' is an mgu for $A\sigma$ and A_k, so there exists a substitution δ such that $\sigma'\delta = \gamma$. Now $G_{j+1} = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\theta_j\theta_{j+1} = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\gamma = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\sigma'\delta = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma'\delta = G'_{j+1}\delta$

2. σ' is an mgu for A_k and $A\sigma$ and $A_k\sigma = A_k$ (because of the standardizing apart), so $\sigma\sigma'$ is a unifier for A_k and A. Furthermore, θ_j is an mgu for A_k and A, so there exists a substitution γ' such that $\theta_j\gamma' = \sigma\sigma'$.

$A'\gamma' = A'\theta_j\gamma' = A'\sigma\sigma' = B_i\sigma\sigma' = B_i\theta_j\gamma'$, so γ' is a unifier for A' and $B_i\theta_j$. Furthermore, θ_{j+1} is an mgu for A' and $B_i\theta_j$, so there exists a substitution δ' such that $\theta_{j+1}\delta' = \gamma'$. Now we have $G'_{j+1} = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma' = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\sigma' = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\theta_j\gamma' = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\theta_j\theta_{j+1}\delta' = G_{j+1}\delta'$

We have shown that $G_{j+1} = G'_{j+1}\delta$ and $G'_{j+1} = G_{j+1}\delta'$, so G_{j+1} and G'_{j+1} are variants.

Since G_{j+1} and G'_{j+1} are variants, we have shown that the two resolution steps leading from G_{j-1} to G_{j+1} can be replaced by a single resolution step, which uses C' as input clause. In the same way, we can eliminate all other uses of C as input clause in the rest of the tree, by constructing a refutation which uses some clause in $U_{C,i}$ to replace a usage of C, each time replacing two resolution steps by one single resolution step. Finally we get an SLD-refutation of $\Pi \cup U_{C,i} \cup \{G\}$ which doesn't use C at all. This means that we have in fact found an SLD-refutation of $\Pi_{u1,C,i} \cup \{G\}$. □

A direct consequence of the proof given above, is the following:

Corollary 3.1 *Let Π be a definite program, G a definite goal and $\Pi_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in Π. Suppose there exists an SLD-refutation of length n of $\Pi \cup \{G\}$, which uses C r times as input clause. Then there exists an SLD-refutation of length $n - r$ of $\Pi_{u1,C,i} \cup \{G\}$.*

Intuitively, this corollary shows that unfolding makes refutations shorter. So unfolding has the potential of improving the efficiency of an SLD-based theorem prover. Especially unfolding often-used clauses is worthwhile, since then the value r mentioned in the corollary is highest. On the other hand, unfolding usually increases the number of clauses. So what we see here is an interesting trade-off between the number of clauses and the average length of a refutation: unfolding usually decreases the average length of a refutation, but also usually increases the number of clauses in the program.

We now proceed to prove that constructing the type 1 program preserves the least Herbrand model M_Π of the program. This is also proved in [TS84], though differently from our proof.

Theorem 3.1 *Let Π be a definite program, $C \in \Pi$ and $\Pi_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in Π. Then $M_\Pi = M_{\Pi_{u1,C,i}}$.*

Proof Let A be some ground atom. Then:
$A \in M_\Pi$ iff (by Theorem A.6)
$\Pi \models A$ iff (by Proposition A.1)
$\Pi \cup \{\leftarrow A\}$ is unsatisfiable iff (by Theorem 2.9)
$\Pi \cup \{\leftarrow A\} \vdash_{sr} \square$ iff (by Proposition 3.1)
$\Pi_{u1,C,i} \cup \{\leftarrow A\} \vdash_{sr} \square$ iff (by Theorem 2.9)
$\Pi_{u1,C,i} \cup \{\leftarrow A\}$ is unsatisfiable iff (by Proposition A.1)
$\Pi_{u1,C,i} \models A$ iff (by Theorem A.6)
$A \in M_{\Pi_{u1,C,i}}$.

Hence $M_\Pi = M_{\Pi_{u1,C,i}}$. □

Thus constructing the type 1 program preserves the least Herbrand model. However, it does not preserve logical equivalence. Take for instance $\Pi = \{C = P(f(x)) \leftarrow P(x)\}$. Then $\Pi_{u1,C,1} = \{P(f^2(x)) \leftarrow P(x)\}$. Now $M_\Pi = M_{\Pi_{u1,C,1}} = \emptyset$, but $\Pi \not\Leftrightarrow \Pi_{u1,C,1}$ since $\Pi_{u1,C,1} \not\models \Pi$. Note that this means that a specialization of Π need not be a specialization of $\Pi_{u1,C,i}$. This is actually one of the reasons for the fact that type 1 unfolding and clause deletion cannot solve all specialization problems (see Section 3.3).

On the other hand, constructing the type 2 program *does* preserve logical equivalence. Since $\Pi \subseteq \Pi_{u2,C,i}$ we have $\Pi_{u2,C,i} \models \Pi$; and because $\Pi_{u2,C,i} \backslash \Pi$ is a set of resolvents of clauses in Π, we also have $\Pi \models \Pi_{u2,C,i}$.

Proposition 3.2 *Let Π be a definite program, $C \in \Pi$ and $\Pi_{u2,C,i}$ the type 2 program resulting from unfolding C upon B_i in Π. Then $\Pi \Leftrightarrow \Pi_{u2,C,i}$.*

3.3 UD_1-specialization

As we have seen in the introduction, unfolding together with clause deletion can be used to solve some specialization problems. In this section we formalize this in a method called UD_1-specialization. The name is an acronym for **U**nfolding and clause **D**eletion, the '1' indicates that we use the type 1 program resulting from unfolding here. UD_1-specialization corresponds to the approach taken in [BIA94].

Definition 3.3 Let Π and Π' be definite programs. We say Π' is a *UD_1-specialization* of Π, if there exists a sequence $\Pi_1 = \Pi, \Pi_2, \ldots, \Pi_n = \Pi'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n-1$, either

1. $\Pi_{j+1} = \Pi_{j_{u1,C,i}}$.
2. $\Pi_{j+1} = \Pi_j \backslash \{C\}$ for some $C \in \Pi_j$.

◇

If $\Pi_{j+1} = \Pi_{j_{u1,C,i}}$, then each clause in Π_{j+1} is either in Π_j, or a resolvent of two clauses in Π_j. Hence $\Pi_j \models \Pi_{j+1}$ in this case. If $\Pi_{j+1} = \Pi_j \backslash \{C\}$, then clearly $\Pi_j \models \Pi_{j+1}$. Thus we have the following:

Proposition 3.3 *Let Π be a definite program and Π' a UD_1-specialization of Π. Then $\Pi \models \Pi'$.*

For a solution Π' to a specialization problem, we have two conditions: $\Pi \models \Pi'$ and Π' should be correct w.r.t. E^+ and E^-. The previous proposition shows that a UD_1-specialization of Π always satisfies the first condition.

However, the second condition cannot always be satisfied by UD_1-specialization. Two kinds of steps can be taken here: Π_{j+1} can be the result of unfolding a clause in Π_j, or by deleting a clause from Π_j. The first kind of step preserves the least Herbrand model, the second kind possibly reduces it. In fact, not only deleting a clause, but also the unfolding step may weaken

the program. For instance, suppose $\Pi = \{P(a), (P(x) \leftarrow P(f(x)))\}$. Then $\Pi' = \{P(a), (P(x) \leftarrow P(f^2(x)))\}$ is the result of unfolding $P(x) \leftarrow P(f(x))$ in Π. Whereas this unfolding step has not affected the least Herbrand model—$M_\Pi = M_{\Pi'} = \{P(a)\}$—it has indeed made the program weaker: $\Pi \models \Pi'$, but $\Pi' \not\models \Pi$.

Actually, even if a correct program Π' is implied by the original program Π, this Π' need no longer be implied by a program Π'' obtained from Π by UD_1-specialization. Since further UD_1-specializations of Π'' can only yield programs which are implied by Π'' (and hence do not imply the solution Π'), UD_1-specialization will not reach a solution of the specialization problem in this case. Consider $\Pi = \{(P(f(x)) \leftarrow P(x)), P(a)\}$. Let $M_\Pi = \{P(a), P(f(a)), P(f^2(a)), P(f^3(a))\ldots\}$, and let $E^+ = M_\Pi \backslash \{P(f^2(a))\}$ and $E^- = \{P(f^2(a))\}$. See figure 3.3.

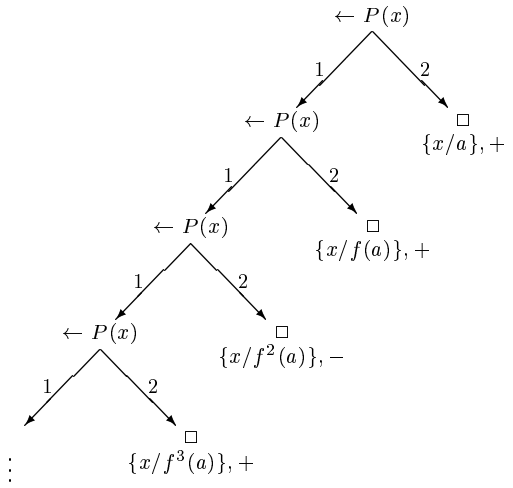


Figure 3.3: The SLD-tree for $\Pi \cup \{\leftarrow P(x)\}$

Let $\Pi_1 = \Pi$. The only clause that can be unfolded is $P(f(x)) \leftarrow P(x)$. Unfolding this clause results in

$$\Pi_2 = \{(P(f^2(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

Then unfolding $P(f^2(x)) \leftarrow P(x)$ gives

$$\Pi_3 = \{(P(f^4(x)) \leftarrow P(x)), P(f^3(a)), P(f^2(a)), P(f(a)), P(a)\}.$$

Notice that $M_{\Pi_1} = M_{\Pi_2} = M_{\Pi_3}$, but unfolding has nevertheless weakened the program: $\Pi_1 \models \Pi_2 \models \Pi_3$, but $\Pi_2 \not\models \Pi_1$ and $\Pi_3 \not\models \Pi_2$. In Π_3, $P(f^4(x)) \leftarrow P(x)$ can be unfolded, etc. It is not difficult to see that in general, any UD_1-specialization of Π is a subset of

$$\{P(f^{2^n}(x)) \leftarrow P(x)), P(f^{2^n-1}(a)), P(f^{2^n-2}(a)), \ldots, P(f^2(a)), P(f(a)), P(a)\},$$

for some n. In order to specialize this program such that $P(f^2(a))$ is no longer derivable, we must in any case remove $P(f^2(a))$. However, this would also prune some of the positive examples (such as $P(f^{2^n+2}(a))$) from the program

via the clause $P(f^{2^n}(x)) \leftarrow P(x)$. Hence there is no UD_1-specialization that solves this particular specialization problem. Note that

$$\Pi'' = \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), P(f(a)), P(a)\}$$

is a solution for this particular specialization problem. $\Pi \models \Pi''$, but the specializations $\Pi_2, \Pi_3, \ldots$ no longer imply this correct program Π''. So in this case, UD_1-specialization has "skipped" over the right solution. In the next section, we will show how this can be solved by UD_2-specialization.

3.4 UD_2-specialization

The previous example showed the incompleteness of UD_1-specialization. But suppose we change our strategy, such that the unfolded clause is not removed immediately from the program. That is, suppose we use type 2 instead of type 1 unfolding. This increases the number of clauses that can later on be used in unfolding. In this case, we *can* find a correct specialization w.r.t. the examples given in Section 3.3, as follows. We start with $\Pi'_1 = \Pi$, and unfold $P(f(x)) \leftarrow P(x)$ without removing the unfolded clause. This gives Π'_2:

$$\Pi'_2 = \{(P(f^2(x)) \leftarrow P(x)), (P(f(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

Now we unfold $P(f^2(x)) \leftarrow P(x)$, again without removing the unfolded clause. This gives Π'_3:

$$\begin{aligned}\Pi'_3 = \quad & \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), (P(f^2(x)) \leftarrow P(x)),\\ & (P(f(x)) \leftarrow P(x)), P(f^3(a)), P(f^2(a)), P(f(a)), P(a)\}.\end{aligned}$$

If we remove $(P(f^2(x)) \leftarrow P(x))$, $(P(f(x)) \leftarrow P(x))$, $P(f^3(a))$ and $P(f^2(a))$ from Π'_3, we obtain Π'':

$$\Pi'' = \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

This is a correct specialization of Π w.r.t. E^+ and E^-: $\Pi'' \models E^+$ and $\Pi'' \not\models P(f^2(a))$.

This example induces a second kind of specialization, UD_2-specialization, which differs from UD_1-specialization in the use of type 2 unfolding instead of type 1 unfolding.[3]

Definition 3.4 Let Π and Π' be definite programs. We say Π' is a *UD_2-specialization* of Π, if there exists a sequence $\Pi_1 = \Pi, \Pi_2, \ldots, \Pi_n = \Pi'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n-1$, either

1. $\Pi_{j+1} = \Pi_{ju2,C,i}$.

[3]Henrik Boström (personal communication) made us aware of the fact that the covering algorithm of [Bos95a], with which his unfolding-algorithm SPECTRE is compared, is in fact equivalent to our UD_2-specialization. He also gave an example of a solution of a specialization problem which could be found by the covering algorithm, though not by SPECTRE, because the hypothesis-space of SPECTRE is a proper subset of the hypothesis-space of the covering algorithm.

2. $\Pi_{j+1} = \Pi_j \backslash \{C\}$ for some $C \in \Pi_j$.

$\diamond$

Note that any UD_1-specialization is also a UD_2-specialization, since obtaining the type 2 program and then removing the unfolded clause in the next step is equivalent to obtaining the type 1 program. The following proposition is obvious:

Proposition 3.4 *Let* Π *be a definite program and* Π' *a* UD_2*-specialization of* Π*. Then* $\Pi \models \Pi'$*.*

Since any UD_1-specialization is a UD_2-specialization, while some UD_2-specializations cannot be found with UD_1-specialization (see the example above), UD_2-specialization is "more complete" than UD_1-specialization. Unfortunately, UD_2-specialization is still not sufficiently strong to provide a solution for all specialization problems. Consider the following: $\Pi = \{P(x)\}$, $E^+ = \{P(f(a)), P(f^2(a))\}$ and $E^- = \{P(a)\}$. $\Pi' = \{P(f(x))\}$ is a solution for this specialization problem. However, no solution can be found by UD_2-specialization. Since Π contains only a single atom, no unfolding can take place here. Hence the only UD_2-specializations of Π are Π itself and the empty set, neither of which is correct. So some specialization problems do not have a UD_2-specialization as a solution.

3.5 UDS-specialization

In order to extend UD_2-specialization to a method which can solve all specialization problems, we have to allow the possibility of taking a *subsumption step.*[4] In general, we can define UDS-specialization (**U**nfolding, clause **D**eletion, **S**ubsumption) as follows:

Definition 3.5 Let Π and Π' be definite programs. We say Π' is a *UDS-specialization* of Π, if there exists a sequence $\Pi_1 = \Pi, \Pi_2, \ldots, \Pi_n = \Pi'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n-1$, one of the following holds:

1. $\Pi_{j+1} = \Pi_{j_{u2,C,i}}$.
2. $\Pi_{j+1} = \Pi_j \backslash \{C\}$ for some $C \in \Pi_j$.
3. $\Pi_{j+1} = \Pi_j \cup \{C\}$ for a C that is subsumed by a clause in Π_j.

$\diamond$

UDS-specialization is indeed complete: any specialization problem has a UDS-specialization as solution. For the proof of completeness, we use the Subsumption Theorem for SLD-resolution (Theorem 2.10).

Theorem 3.2 *Let* Π *and* Π' *be definite programs, such that* Π' *contains no tautologies. Then* $\Pi \models \Pi'$ *iff* Π' *is a UDS-specialization of* Π*.*

[4] Subsumption can be seen as a solution to the problem of *ambivalent* leaves in an SLD-tree [BIA94, Bos95b].

Proof

$\Leftarrow$: By the soundness of resolution and subsumption.

$\Rightarrow$: Suppose $\Pi \models \Pi'$. Then for every $C \in \Pi'$, we have $\Pi \models C$. Let C be some particular clause in Π' that is not in Π. Then by the Subsumption Theorem for SLD-resolution, there exists an SLD-derivation from Π of a clause D which subsumes C, as shown in figure 3.4.

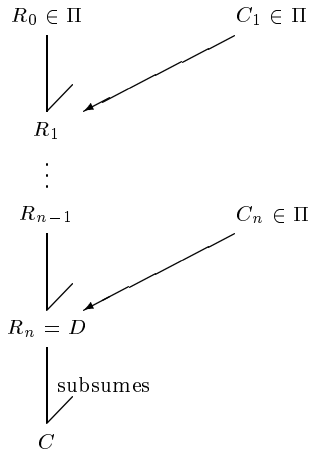


Figure 3.4: An SLD-deduction of C from Π

Since R_1 is a resolvent of R_0 and C_1 (upon the selected atom B_i in R_0), if we unfold R_0 in Π upon B_i we get the program $\Pi_{u2,R_0,i}$ which contains R_1. Now when we unfold R_1 in $\Pi_{u2,R_0,i}$, we get a program which contains R_2, etc. Thus after n applications of (type 2) unfolding, we can produce a UDS-specialization (a superset of Π) containing the clause $R_n = D$. Since D subsumes C, we can add C to the program, by the third item in the definition of UDS-specialization.

If we do this for every $C \in \Pi'$ that is not in Π, we get a program Π'' which contains every clause in Π'. Since Π'' is obtained from Π by a finite number of applications of unfolding and subsumption, Π'' is a UDS-specialization of Π. Now delete from Π'' all those clauses that are not in Π'. Then we obtain Π' as a UDS-specialization of Π. Thus if $\Pi \models \Pi'$, then Π' is a UDS-specialization of Π. □

Now suppose we have Π, Π', E^+ and E^-, such that $\Pi \models \Pi'$ and Π' is correct w.r.t. E^+ and E^-. We can assume Π' contains no tautologies. Then it follows from the previous theorem that Π' is a UDS-specialization of Π. This shows that UDS-specialization is complete:[5]

Corollary 3.2 (Completeness of UDS-specialization) *Every specialization problem with Π as initial program has a UDS-specialization of Π as solution.*

[5]UDS-specialization need not specialize *minimally* in the sense advocated in [Wro93]: a UDS-specialization of an initial Π may be considerably different from Π. On the other hand, the approach of [Wro93] has the disadvantage that each clause in a specialized theory should be equal to or subsumed by a clause in the initial Π (p. 71, postulate 1), which is quite restrictive.

Efficiency:
Note that if we want to unfold some particular clause C, we actually only need to consider the resolvents of C and clauses from the original Π. This is clear from figure 3.4, since in order to produce R_{i+1}, we only need to resolve R_i with C_{i+1}, which is a member of the original Π. In other words, we only need to add a subset of $U_{C,i}$ to the program. We might define $U'_{C,i}$ as the set of resolvents upon B_i of C and clauses from the original Π and then use $\Pi_{j+1} = \Pi_j \cup U'_{C,i}$ instead of $\Pi_{j+1} = \Pi_{j_{u2,C,i}} = \Pi_j \cup U_{C,i}$. This reduces the number of clauses that unfolding produces, and hence improves efficiency.

3.6 Relation with inverse resolution

As we have already seen in Chapter 1, there are basically two possible approaches in ILP. We have the *top-down* approach (of which UDS-specialization is an example) which starts with an overly general program and specializes this, and the *bottom-up* approach, which starts with an overly specific program and generalizes this. There is an interesting relation between our previous analysis of program specialization on the one hand, and program generalization by *inverse resolution* (see for instance [MB88, Mug92a, Rou92, SA93]) on the other hand. The inversion of resolution is a well-known approach towards generalization in ILP. Here the inversion of a resolution step can be viewed as the dual of unfolding.

However, in the same way as specialization by unfolding is not complete without subsumption, its dual also needs (the inversion of) subsumption. Most research in inverse resolution has focused on inverting resolution steps, mostly ignoring the inversion of the final subsumption step. By the previous analysis, inverting a subsumption step will be necessary for completeness. For example, we cannot generalize $\Pi = \{P(f(x))\}$ to $\Pi' = \{P(x)\}$ just by inverting resolution steps.

3.7 Summary

The *specialization problem*, a special case of the general problem setting for ILP, can be stated as follows:

> **Given**: A definite program Π and two disjoint sets of ground atoms E^+ and E^-, such that Π is overly general w.r.t. E^+ and E^-, and suppose there exists a definite program Π' such that $\Pi \models \Pi'$ and Π' is correct w.r.t. E^+ and E^-.
> **Find**: One such a Π'.

Unfolding, constructing the set $U_{C,i}$ of resolvents of a clause $C \in \Pi$ with clauses in Π, can be used as a tool for solving such problems. The type 1 program is obtained by replacing C in Π by $U_{C,i}$, while the type 2 program is $\Pi \cup U_{C,i}$. Constructing the type 1 program preserves the least Herband model, while the type 2 program preserves logical equivalence with the original program.

We defined three increasingly strong specialization techniques here. UD_1- and UD_2-specialization employ clause deletion and, respectively, the type 1 and type 2 programs resulting from unfolding. Both are incomplete. If we add to UD_2-specialization the possibility of taking a subsumption step, we obtain UDS-specialization. This is a complete specialization method: every specialization problem with Π as initial program has a UDS-specialization of Π as solution.

Chapter 4

Least Generalizations and Greatest Specializations

4.1 Introduction

Inductive Logic Programming is concerned with learning from examples. Learning from examples means adjusting a theory to the examples. As we have seen in Chapter 1, the two main operations in ILP for adjustment of a theory, are *generalization* and *specialization*. Generalization strengthens a theory that is too weak, while specialization weakens a theory that is too strong. These operations only make sense within a *generality order*, which is a relation stating when some clause is more general than some other clause.

The three most important generality orders used in ILP are subsumption (also called θ-subsumption), logical implication and implication relative to background knowledge.[1] In the subsumption order, we say that clause C is more general than D—or, equivalently, D is more specific than C—in case C subsumes D. In the implication order, we say that C is more general than D if C logically implies D. Finally, C is more general than D relative to background knowledge Σ (Σ is a set of clauses), if $\{C\} \cup \Sigma$ logically implies D.

Of these three orders, subsumption is the most tractable. In particular, subsumption is decidable, whereas logical implication is not decidable, not even for Horn clauses, as established in [MP92]. In turn, relative implication is harder than implication: both are undecidable, but proof procedures for implication need to take only derivations from C into account, whereas a proof procedure for *relative* implication should check all derivations from $\{C\} \cup \Sigma$.

Within a generality order, there are two approaches to generalization or specialization. The first approach generalizes or specializes *individual* clauses. We will not discuss this in any detail in this chapter, and only mention it for completeness' sake. This approach can be traced back to Reynolds' concept of a *cover* [Rey70]. It was implemented for example by Shapiro in the subsumption order, in the form of *refinement operators* [Sha81b]. However, a clause C which implies another clause D need not subsume this D. For instance, take $C = P(f(x)) \leftarrow P(x)$ and $D = P(f^2(x)) \leftarrow P(x)$. Then C does not subsume D,

[1] There is also relative subsumption [Plo71b], which will be briefly touched in Section 4.4.

but $C \models D$. Thus subsumption is weaker than implication. A further sign of this weakness is the fact that tautologies need not be subsume-equivalent, even though they are logically equivalent.

The second approach generalizes or specializes *sets* of clauses. This is the approach we will be concerned with in this chapter. Here the concept of a *least generalization*[2] is important. The use of such *least* generalizations allows us to generalize cautiously, avoiding over-generalization. Least generalizations of sets of clauses were first discussed by Plotkin [Plo70, Plo71a, Plo71b]. He proved that any finite set S of clauses has a least generalization under subsumption (LGS). This is a clause which subsumes all clauses in S and which is subsumed by all other clauses that also subsume all clauses in S. Positive examples can be generalized by taking their LGS.[3] Of course, we need not take an LGS of *all* positive examples, which would yield a theory consisting of only one clause. Instead, we might divide the positive examples into subsets, and take a separate LGS of each subset. That way we obtain a theory containing more than one clause.

For this second approach, subsumption is again not fully satisfactory. For example, if S consists of the clauses $D_1 = P(f^2(a)) \leftarrow P(a)$ and $D_2 = P(f(b)) \leftarrow P(b)$, then the LGS of S is $P(f(y)) \leftarrow P(x)$. The clause $P(f(x)) \leftarrow P(x)$, which seems more appropriate as a least generalization of S, cannot be found by Plotkin's approach, because it does not subsume D_1. As this example also shows, the subsumption order is particularly unsatisfactory when we consider *recursive* clauses: clauses which can be resolved with themselves.

Because of the weakness of subsumption, it is desirable to make the step from the subsumption order to the more powerful implication order. Accordingly, it is important to find out whether Plotkin's positive result on the existence of LGS's also holds for implication. Thus the question whether any finite set of clauses has a least generalization under implication (LGI), has been devoted quite a lot of attention recently. Sofar, this question has only been partly answered. For instance, Idestam-Almquist [IA93, IA95] studies least generalizations under *T-implication* as an approximation to LGI's. Muggleton and Page [MP94] investigate *self-saturated* clauses. A clause is self-saturated if it is subsumed by any clause which implies it. A clause D is a self-saturation of C, if C and D are logically equivalent and D is self-saturated. As stated in [MP94], if two clauses C_1 and C_2 have self-saturations D_1 and D_2, then an LGS of D_1 and D_2 is also an LGI of C_1 and C_2. This answers our question concerning the existence of LGI's for clauses which have a self-saturation. However, Muggleton and Page also show that there exist clauses which have no self-saturation. So the concept of self-saturation cannot solve our question in general.

Use of the third generality order, relative implication, is even more desirable than the use of "plain" implication. Relative implication allows us to take background knowledge into account, which can be used to formalize many useful properties and relations of the domain of application. For this reason, least

[2] Least generalizations are often called least *general* generalizations, for instance in [Plo71b, MP94, IA93, IA95, Nib88], though not in [Plo70], but we feel this 'general' is redundant.

[3] There is also a relation between least generalization under subsumption and *inverse resolution* [Mug92a].

generalizations under implication relative to background knowledge also deserve attention.

Apart from the least generalization, there is also its dual: the *greatest specialization*. Greatest specializations have been accorded much less attention in ILP than least generalizations, but the concept of a greatest specialization may nevertheless be useful (see the beginning of Section 4.6).

In this chapter, we give a systematic treatment of the existence and non-existence of least generalizations and greatest specializations, applied to each of these three generality orders. Apart from distinguishing between these three orders, we also distinguish between languages of general clauses and more restricted languages of Horn clauses. Though most researchers in ILP restrict attention to Horn clauses, general clauses are also sometimes used [Plo70, Plo71b, Sha81b, DRB93, IA93, IA95]. Moreover, many researchers who do not use general clauses actually allow negative literals to appear in the body of a clause. That is, they use clauses of the form $A \leftarrow L_1, \ldots, L_n$, where A is an atom and each L_i is a literal. These are called *program* clauses [Llo87]. Program clauses are in fact logically equivalent to general clauses. For instance, the program clause $P(x) \leftarrow Q(x), \neg R(x)$ is equivalent to the non-Horn clause $P(x) \vee \neg Q(x) \vee R(x)$. For these two reasons, we consider not only languages of Horn clauses, but also pay attention to languages of general clauses.

The combination of three generality orders and two different possible languages of clauses gives a total of six different ordered languages. For each of these, we can ask whether least generalizations (LG's) and greatest specializations (GS's) always exist. We survey results already obtained by others and also contribute some answers of our own. For the sake of clarity, we will summarize the results of our survey right at the outset. In the following table '+' signifies a positive answer, '−' means a negative answer.

	Horn clauses		General clauses	
Quasi-order	LG	GS	LG	GS
Subsumption ($\succeq$)	+	+	+	+
Implication ($\models$)	−	−	+ for function-free	+
Relative implication ($\models_\Sigma$)	−	−	−	+

Table 4.1: Existence of LG's and GS's

Our own contributions to this table are threefold. First and foremost, we prove that if S is a finite set of clauses containing at least one non-tautologous function-free clause[4] (apart from this non-tautologous function-free clause, S may contain an arbitrary finite number of other clauses, including clauses which contain functions), then there exists a computable LGI of S. This result is on the one hand based on the Subsumption Theorem, which allows us to restrict attention to finite sets of ground instances of clauses and on the other hand on a modification of some proofs concerning T-implication which can be found in [IA93, IA95]. An immediate corollary of this result is the existence and computability of an LGI of any finite set of function-free clauses. As far as we

[4] A clause which only contains constants and variables as terms.

know, both our general LGI-result and this particular corollary are new results.

Niblett [Nib88, p. 135] claims that "it is simple to show that there are lggs if the language is restricted to a fixed set of constant symbols since all Herbrand interpretations are finite." Yet even for this special case of our general result, it appears that no proof has been published. Initially, we found a direct proof of this case, but this was not really any simpler than the proof of the more general result that we give in this chapter. Niblett's idea that the proof is simple may be due to some confusion about the relation between Herbrand models and logical implication (which is defined in terms of *all* models, not just Herbrand models). We will describe this at the end of Subsection 4.5.1. Or perhaps one might think that the decidability of implication for function-free clauses immediately implies the existence of an LGI. But in fact, decidability is not a sufficient condition for the existence of a least generalization. For example, it is decidable whether one function-free clause C implies another function-free clause D relative to function-free background knowledge. Yet least generalizations relative to function-free background knowledge do not always exist, as we will show in Section 4.7.

Our LGI-result does not solve the general question of the existence of LGI's, but it does provide a positive answer for a large class of cases: the presence of one non-tautologous function-free clause in a finite S already guarantees the existence and computability of an LGI of S, no matter what other clauses S contains.[5] Because of the prominence of function-free clauses in ILP, this case may be of great practical signifcance. Often, in particular in implementations of ILP-systems, the language is required to be function-free, or function symbols are removed from clauses and put in the background knowledge by techniques such as *flattening* [Rou92]. Well-known ILP-systems such as FOIL [QCJ93], LINUS [LD94] and MOBAL [MWKE93] all use only function-free clauses. More than one half of the ILP-systems surveyed by David Aha [Aha92] is restricted to function-free clauses. Function-free clauses are also sufficient for most applications concerning databases.

Our second contribution shows that a set S need not have a least generalization relative to some background knowledge Σ, not even when S and Σ are both function-free.

Thirdly, we contribute a complete discussion of existence and non-existence of greatest specializations in each of the six ordered languages. In particular, we show that any finite set of clauses has a greatest specialization under implication. Combining this with the corollary of our result on LGI's, it follows that a function-free clausal language is a *lattice*.

[5] Note that even for function-free clauses, the subsumption order is still not enough. Consider $D_1 = P(x,y,z) \leftarrow P(y,z,x)$ and $D_2 = P(x,y,z) \leftarrow P(z,x,y)$. D_1 is a resolvent of D_2 and D_2 and D_2 is a resolvent of D_1 and D_1. Hence D_1 and D_2 are logically equivalent. This means that D_1 is an LGI of the set $\{D_1, D_2\}$. However, the LGS of these two clauses is $P(x,y,z) \leftarrow P(u,v,w)$, which is clearly an over-generalization.

4.2 Preliminaries

In this chapter, it will be convenient to ignore the order and possible duplication of literals in a clause. Clearly, this order and duplication does not affect the truth-value of a clause. Thus $P(x) \vee Q(x) \vee P(x)$ and $Q(x) \vee Q(x) \vee P(x)$ can both be considered as the same clause $P(x) \vee Q(x)$. The union $C \cup D$ of two clauses denotes a clause which contains every literal in C and D.

The two languages of clauses that will be considered in this chapter are the following:

Definition 4.1 Let $\mathcal{A}$ be an alphabet of the first-order logic. Then the *clausal language $\mathcal{C}$ by $\mathcal{A}$* is the set of all clauses which can be constructed from the symbols in $\mathcal{A}$. The *Horn language $\mathcal{H}$ by $\mathcal{A}$* is the set of all Horn clauses which can be constructed from the symbols in $\mathcal{A}$. $\diamond$

Here we just presuppose some arbitrary alphabet $\mathcal{A}$, and consider the clausal language $\mathcal{C}$ and Horn language $\mathcal{H}$ based on this $\mathcal{A}$.

Three increasingly strong generality orders on clauses are subsumption, implication and *relative* implication. Below we repeat the definitions of subsumption and implication, and introduce the definition of relative implication.

Definition 4.2 Let C and D be clauses and Σ be a set of clauses. We say that C *subsumes* D, denoted as $C \succeq D$, if there exists a substitution θ such that $C\theta \subseteq D$.[6] C and D are *subsume-equivalent* if $C \succeq D$ and $D \succeq C$.

Σ *(logically) implies* C, denoted as $\Sigma \models C$, if every model of Σ is also a model of C. C *(logically) implies* D, denoted as $C \models D$, if $\{C\} \models D$. C and D are *(logically) equivalent* if $C \models D$ and $D \models C$.

C *implies* D *relative to* Σ, denoted as $C \models_\Sigma D$, if $\Sigma \cup \{C\} \models D$. C and D are *equivalent relative to* Σ if both $C \models_\Sigma D$ and $D \models_\Sigma C$. $\diamond$

If $C \succeq D$, then $C \models D$. The converse does not hold, as the examples in the Introduction showed. Similarly, if $C \models D$, then $C \models_\Sigma D$, and again the converse need not hold. Consider the clauses $C = P(a) \vee \neg P(b)$, $D = P(a)$ and $\Sigma = \{P(b)\}$: then $C \models_\Sigma D$, but $C \not\models D$.

The next lemma was first proved by Gottlob [Got87]. Actually, it is an immediate corollary of the Subsumption Theorem:

Lemma 4.1 (Gottlob) *Let C and D be non-tautologous clauses. If $C \models D$, then $C^+ \succeq D^+$ and $C^- \succeq D^-$.*

Proof Since $C^+ \succeq C$, if $C \models D$, then we have $C^+ \models D$. Since C^+ cannot be resolved with itself, it follows from the Subsumption Theorem that $C^+ \succeq D$. But then C^+ must subsume the positive literals in D, hence $C^+ \succeq D^+$. Similarly $C^- \succeq D^-$. $\square$

An important consequence of this lemma concerns the *depth* of clauses, defined as follows:

[6] Right from the very first applications of subsumption in ILP, there has been some controversy about the symbol used for subsumption: Plotkin [Plo70] used '$\leq$', while Reynolds [Rey70] used '$\geq$'. We use '$\succeq$' here, similar to Reynolds' '$\geq$', because we feel it serves the intuition to view C as somehow "bigger" or "stronger" than D, if $C \succeq D$ holds.

Definition 4.3 Let t be a term. If t is a variable or constant, then the *depth* of t is 1. If $t = f(t_1, \ldots, t_n)$, $n \geq 1$, then the depth of t is 1 plus the depth of the t_i with largest depth. The *depth* of a clause C is the depth of the term with largest depth in C. $\diamond$

Example 4.1 The term $t = f(a, x)$ has depth 2. $C = P(f(x)) \leftarrow P(g(f(x), a))$ has depth 3, since $g(f(x), a)$ has depth 3. $\triangleleft$

It follows from Gottlob's lemma that if $C \models D$, then the depth of C is smaller than or equal to the depth of D, for otherwise C^+ cannot subsume D^+ or C^- cannot subsume D^-. For instance, take $D = P(x, f(x, g(y))) \leftarrow P(g(a), b)$, which has depth 3. Then a clause C containing a term $f(x, g^2(y))$ (depth 4) cannot imply D.

Lemma 4.2 *Let Σ be a set of clauses, C be a clause, and σ be a Skolem substitution for C w.r.t. Σ. Then $\Sigma \models C$ iff $\Sigma \models C\sigma$.*

Proof

$\Rightarrow$: Obvious.

$\Leftarrow$: Suppose C is not a tautology and let $\sigma = \{x_1/a_1, \ldots, x_n/a_n\}$. If $\Sigma \models C\sigma$, it follows from the Subsumption Theorem that there is a D such that $\Sigma \vdash_r D$ and $D \succeq C\sigma$. Thus there is a θ, such that $D\theta \subseteq C\sigma$. Note that since $\Sigma \vdash_r D$ and none of the constants $a_1, \ldots, a_n$ appears in Σ, none of these constants appears in D. Now let θ' be obtained by replacing in θ all occurrences of a_i by x_i, for every $1 \leq i \leq n$. Then $D\theta' \subseteq C$, hence $D \succeq C$. Therefore $\Sigma \vdash_d C$, and hence $\Sigma \models C$. $\square$

4.3 Least generalizations and greatest specializations

In this section, we will define the concepts we need concerning least generalizations and greatest specializations.

Definition 4.4 Let $?$ be a set and R be a binary relation on $?$ (i.e., $R \subseteq ? \times ?$).

1. R is *reflexive on* $?$, if xRx for every $x \in ?$.
2. R is *transitive on* $?$, if for every $x, y, z \in ?$, xRy and yRz implies xRz.
3. R is *symmetric on* $?$, if for every $x, y \in ?$, xRy implies yRx.
4. R is *anti-symmetric on* $?$, if for every $x, y, z \in ?$, xRy and yRx implies $x = y$.

If R is both reflexive and transitive on $?$, we say R is a *quasi-order* on $?$. If R is both reflexive, transitive and anti-symmetric on $?$, we say R is a *partial order* on $?$. If R is reflexive, transitive and symmetric on $?$, R is an *equivalence relation* on $?$. $\diamond$

A quasi-order R on $?$ induces an equivalence-relation $\sim$ on $?$, as follows: we say $x, y \in ?$ are *equivalent* induced by R (denoted $x \sim y$) if both xRy and yRx. Using this equivalence relation, a quasi-order R on $?$ induces a partial order R' on the set of equivalence classes in $?$, defined as follows: if $[x]$ denotes the equivalence class of x (i.e., $[x] = \{y \mid x \sim y\}$), then $[x]R'[y]$ iff xRy.

We first give a general definition of least generalizations and greatest specializations for sets of clauses ordered by some quasi-order, which we then instantiate in different ways.

Definition 4.5 Let $?$ be a set of clauses, $\geq$ be a quasi-order on $?$, $S \subseteq ?$ be a finite set of clauses and $C \in ?$. If $C \geq D$ for every $D \in S$, then we say C is a *generalization* of S under $\geq$. Such a C is called a *least generalization (LG)* of S under $\geq$ in $?$, if we have $C' \geq C$ for every generalization $C' \in ?$ of S under $\geq$.

Dually, C is a *specialization* of S under $\geq$, if $D \geq C$ for every $D \in S$. Such a C is called a *greatest specialization (GS)* of S under $\geq$ in $?$, if we have $C \geq C'$ for every specialization $C' \in ?$ of S under $\geq$. $\diamond$

It is easy to see that if some set S has an LG or GS under $\geq$ in $?$, then this LG or GS will be unique up to the equivalence induced by $\geq$ in $?$. That is, if C and D are both LG's or GS's of some set S, then we have $C \sim D$.

The concepts defined above are instances of the mathematical concepts of (least) upper bounds and (greatest) lower bounds. Thus we can speak of lattice-properties of a quasi- or partially ordered set of clauses:

Definition 4.6 Let $?$ be a set of clauses and $\geq$ a quasi-order on $?$. If for every finite subset S of $?$ there exist both a least generalization and a greatest specialization of S under $\geq$ in $?$, then $?$ ordered by $\geq$ is called a *lattice*. $\diamond$

It should be noted that usually in mathematics, a lattice is defined for a partial order instead of a quasi-order. However, since in ILP we usually have to deal with individual clauses rather than with equivalence classes of clauses, it is convenient for us to define 'lattice' for a quasi-order here. Anyhow, if a quasi-order $\geq$ is a lattice on $?$, then the partial order induced by $\geq$ is a lattice on the set of equivalence classes in $?$.

In ILP, there are two main instantiations for the set of clauses $?$: either we take a clausal language $\mathcal{C}$, or we take a Horn language $\mathcal{H}$. Similarly, there are three interesting choices for the quasi-order $\geq$: we can use either $\succeq$ (subsumption), $\models$ (implication), or $\models_\Sigma$ (relative implication) for some background knowledge Σ. It is easy to see that each of these is indeed a quasi-order on a set of clauses. In the $\succeq$-order, we will sometimes abbreviate the terms 'least generalization of S under subsumption' and 'greatest specialization of S under subsumption' to 'LGS of S' and 'GSS of S', respectively. Similarly, in the $\models$-order we will sometimes speak of an LGI (least generalization under implication) and a GSI. In the $\models_\Sigma$-order, we will use LGR (least generalization under *relative* implication) and GSR.

These two different languages and three different quasi-orders give a total of six combinations. For each combination, we can ask whether an LG or

GS of every finite set S exists. In the next section, we will review the answers for subsumption given by others or by ourselves. Then we devote two sections to least generalizations and greatest specializations under implication, respectively. Finally, we discuss least generalizations and greatest specializations under relative implication. The results of this survey have already been summarized in Table 4.1 in the Introduction.

4.4 Subsumption

First we devote some attention to subsumption. Least generalizations under subsumption have been discussed extensively by Plotkin [Plo70]. The main result in his framework is the following:

Theorem 4.1 (Existence of LGS in $\mathcal{C}$) *Let $\mathcal{C}$ be a clausal language. Then for every finite $S \subseteq \mathcal{C}$, there exists an LGS of S in $\mathcal{C}$.*

If S only contains Horn clauses, then it can be shown that the LGS of S is itself also a Horn clause. Thus the question for the existence of an LGS of every finite set S of clauses is answered positively for both clausal languages and for Horn languages.

Plotkin established the existence of an LGS, but he seems to have ignored the GSS in [Plo70, Plo71b], possibly because it is a very straightforward result. It is in fact fairly easy to show that the GSS of some finite set S of clauses is simply the union of all clauses in S after they are standardized apart.[7] We include the proof here.

Theorem 4.2 (Existence of GSS in $\mathcal{C}$) *Let $\mathcal{C}$ be a clausal language. Then for every finite $S \subseteq \mathcal{C}$, there exists a GSS of S in $\mathcal{C}$.*

Proof Suppose $S = \{D_1, \ldots, D_n\} \subseteq \mathcal{C}$. Without loss of generality, we assume the clauses in S are standardized apart. Let $D = D_1 \cup \ldots \cup D_n$, then $D_i \succeq D$, for every $1 \leq i \leq n$. Now let $C \in \mathcal{C}$ be such that $D_i \succeq C$, for every $1 \leq i \leq n$. Then for every $1 \leq i \leq n$, there is a θ_i such that $D_i\theta_i \subseteq C$ and θ_i only acts on variables in D_i. If we let $\theta = \theta_1 \cup \ldots \cup \theta_n$, then $D\theta = D_1\theta_1 \cup \ldots \cup D_n\theta_n \subseteq C$. Hence $D \succeq C$, so D is a GSS of S in $\mathcal{C}$. □

This establishes that a clausal language $\mathcal{C}$ ordered by $\succeq$ is a lattice.

Proving the existence of a GSS of every finite set of Horn clauses in $\mathcal{H}$ requires a little more work, but here also the result is positive. For example, $D = P(a) \leftarrow P(f(a)), Q(y)$ is a GSS of $D_1 = P(x) \leftarrow P(f(x))$ and $D_2 = P(a) \leftarrow Q(y)$. Note that D can be obtained by applying $\sigma = \{x/a\}$ (the mgu for the heads of D_1 and D_2) to $D_1 \cup D_2$, the GSS of D_1 and D_2 in $\mathcal{C}$. This idea

[7] Note that this has nothing to do with *unification*. For instance, if $S = \{P(a, x), P(y, b)\}$, then the GSS of S in $\mathcal{C}$ would be $P(a, x) \vee P(y, b)$. However, if we would instantiate Γ in Definition 4.5 to the set of atoms, then the greatest specialization of two atoms *in the set of atoms* should itself also be an atom. The GSS of two atoms is then their most general unification [Rey70]. For instance, the GSS of S would in this case be $P(a, b)$.

will be used in the following proof. Here we assume $\mathcal{H}$ contains an artificial bottom element (True) $-$, such that $C \succeq -$ for every $C \in \mathcal{H}$, and $- \not\succeq C$ for every $C \neq -$. Note that $-$ is not subsume-equivalent with other tautologies.

Theorem 4.3 (Existence of GSS in $\mathcal{H}$) *Let $\mathcal{H}$ be a Horn language, with $- \in \mathcal{H}$. Then for every finite $S \subseteq \mathcal{H}$, there exists a GSS of S in $\mathcal{H}$.*

Proof Suppose $S = \{D_1, \ldots, D_n\} \subseteq \mathcal{H}$. Without loss of generality we assume the clauses in S are standardized apart, $D_1, \ldots, D_k$ are the definite program clauses in S and $D_{k+1}, \ldots, D_n$ are the definite goals in S. If $k = 0$ (i.e., if S only contains goals), then it is easy to show that $D_1 \cup \ldots \cup D_n$ is a GSS of S in $\mathcal{H}$. If $k \geq 1$ and the set $\{D_1^+, \ldots, D_k^+\}$ is not unifiable, then $-$ is a GSS of S in $\mathcal{H}$. Otherwise, let σ be an mgu for $\{D_1^+, \ldots, D_k^+\}$, and let $D = D_1\sigma \cup \ldots \cup D_n\sigma$ (note that actually $D_i\sigma = D_i$ for $k+1 \leq i \leq n$, since the clauses in S are standardized apart). Since D has exactly one literal in its head, it is a definite program clause. Furthermore, we have $D_i \succeq D$ for every $1 \leq i \leq n$, since $D_i\sigma \subseteq D$.

To show that D is a GSS of S in $\mathcal{H}$, suppose $C \in \mathcal{H}$ is some clause such that $D_i \succeq C$ for every $1 \leq i \leq n$. For every $1 \leq i \leq n$, let θ_i be such that $D_i\theta_i \subseteq C$ and θ_i only acts on variables in D_i. Let $\theta = \theta_1 \cup \ldots \cup \theta_n$. For every $1 \leq i \leq k$, $D_i^+\theta = D_i^+\theta_i = C^+$, so θ is a unifier of $\{D_1^+, \ldots, D_k^+\}$. But σ is an mgu for this set, so there is a γ such that $\theta = \sigma\gamma$. Now $D\gamma = D_1\sigma\gamma \cup \ldots \cup D_n\sigma\gamma = D_1\theta \cup \ldots \cup D_n\theta = D_1\theta_1 \cup \ldots \cup D_n\theta_n \subseteq C$. Hence $D \succeq C$, so D is a GSS of S in $\mathcal{H}$. See figure 4.1 for illustration of the case where $n = 2$. □

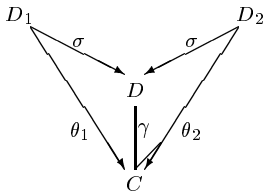


Figure 4.1: D is a GSS of D_1 and D_2

Thus a Horn language $\mathcal{H}$ ordered by $\succeq$ is also a lattice.

We end this section by briefly discussing Plotkin's *relative subsumption* [Plo71b]. This is an extension of subsumption which takes background knowledge into account. This background knowledge is rather restricted: it must be a finite set Σ of *ground literals*. Because of its restrictiveness, we have not included relative subsumption in Table 4.1. Nevertheless, we mention it here, because least generalization under relative subsumption forms the basis of the well-known ILP system GOLEM [MF92].

Definition 4.7 Let C, D be clauses, $\Sigma = \{L_1, \ldots, L_m\}$ be a finite set of ground literals. Then C *subsumes* D *relative to* Σ, denoted by $C \succeq_\Sigma D$, if $C \succeq (D \cup \{\neg L_1, \ldots, \neg L_m\})$. ◇

It is easy to see that $\succeq_\Sigma$ is reflexive and transitive, so it imposes a quasi-order on a set of clauses.

Suppose $S = \{D_1, \ldots, D_n\}$ and $\Sigma = \{L_1, \ldots, L_m\}$. It is easy to see that an LGS of $\{(D_1 \cup \{\neg L_1, \ldots, \neg L_m\}), \ldots, (D_n \cup \{\neg L_1, \ldots, \neg L_m\})\}$ is a least generalization of S under $\succeq_\Sigma$, so every finite set of clauses has a least generalization under $\succeq_\Sigma$ in $\mathcal{C}$. Moreover, if each D_i is a Horn clause and each L_j is a *positive* ground literal (i.e., a ground atom), then this least generalization will itself also be a Horn clause. Accordingly, if Σ is a finite set of positive ground literals, then every finite set of Horn clauses has a least generalization under $\succeq_\Sigma$ in $\mathcal{H}$.

4.5 Least generalizations under implication

Now we turn from subsumption to the implication order. In this section we will discuss LGI's, in the next section we handle GSS's. For Horn clauses, the LGI-question has already been answered negatively in [MDR94].

Let $D_1 = P(f^2(x)) \leftarrow P(x)$, $D_2 = P(f^3(x)) \leftarrow P(x)$, $C_1 = P(f(x)) \leftarrow P(x)$ and $C_2 = P(f^2(y)) \leftarrow P(x)$. Then we have both $C_1 \models \{D_1, D_2\}$ and $C_2 \models \{D_1, D_2\}$. It is not very difficult to see that there are no more specific Horn clauses than C_1 and C_2 that imply both D_1 and D_2. For C_1: no resolvent of C_1 with itself implies D_2 and no clause that is properly subsumed by C_1 still implies D_1 and D_2. For C_2: every resolvent of C_2 with itself is a variant of C_2, and no clause that is properly subsumed by C_2 still implies D_1 and D_2. Thus C_1 and C_2 are both "minimal" generalizations under implication of $\{D_1, D_2\}$. Since C_1 and C_2 are not logically equivalent under implication, there is no LGI of $\{D_1, D_2\}$ in $\mathcal{H}$.

However, the fact that there is no LGI of $\{D_1, D_2\}$ in $\mathcal{H}$, does not mean that D_1 and D_2 have no LGI in $\mathcal{C}$, since a Horn language is a more restricted space than a clausal language. In fact, it is shown in [MP94] that $C = P(f(x)) \vee P(f^2(y)) \leftarrow P(x)$ is an LGI of D_1 and D_2 in $\mathcal{C}$. For this reason, it may be worthwhile for the LGI to consider a clausal language instead of only Horn clauses.

In the next subsection, we show that any finite set of clauses which contains at least one non-tautologous function-free clause, has an LGI in $\mathcal{C}$. An immediate corollary of this result is the existence of an LGI of any finite set of function-free clauses. In our usage of the word, a 'function-free' clause may contain constants, even though constants are sometimes seen as functions of arity 0.

Definition 4.8 A clause is *function-free* if it does not contain function symbols of arity 1 or more. ◇

Note that a clause is function-free iff it has depth 1. In case of sets of clauses which all contain function symbols, the LGI-question remains open.

4.5.1 A sufficient condition for the existence of an LGI

In this subsection, we will show that any finite set S of clauses containing at least one non-tautologous function-free clause, has an LGI in $\mathcal{C}$.

Definition 4.9 Let C be a clause, $x_1, \ldots, x_n$ all distinct variables in C, and K a set of terms. Then the *instance set* of C w.r.t. K is $\mathcal{I}(C,K) = \{C\theta \mid \theta = \{x_1/t_1, \ldots, x_n/t_n\}$, where $t_i \in K$, for every $1 \leq i \leq n\}$. If $\Sigma = \{C_1, \ldots, C_k\}$ is a set of clauses, then the *instance set* of Σ w.r.t. K is $\mathcal{I}(\Sigma,K) = \mathcal{I}(C_1,K) \cup \ldots \cup \mathcal{I}(C_k,K)$. $\diamond$

Example 4.2 If $C = P(x) \vee Q(y)$ and $T = \{a, f(z)\}$, then $\mathcal{I}(C,T) = \{(P(a) \vee Q(a)), (P(a) \vee Q(f(z))), (P(f(z)) \vee Q(a)), (P(f(z)) \vee Q(f(z)))\}$. $\lhd$

Definition 4.10 Let S be a finite set of clauses, and σ be a Skolem substitution for S. Then the *term set* of S by σ is the set of all terms (including subterms) occurring in $S\sigma$. $\diamond$

A term set of S by some σ is a finite set of ground terms.

Example 4.3 The term set of $D = P(f^2(x), y, z) \leftarrow P(y, z, f^2(x))$ by $\sigma = \{x/a, y/b, z/c\}$ is $T = \{a, f(a), f^2(a), b, c\}$. $\lhd$

Our definition of a term set corresponds to what Idestam-Almquist [IA93, IA95] calls a 'minimal term set'. In his definition, if σ is a Skolem substitution for a set of clauses $S = \{D_1, \ldots, D_n\}$ w.r.t. some other set of clauses S', then a *term set* of S is a finite set of terms which contains the minimal term set of S by σ as a subset.

Using his notion of term set, he defines *T-implication* as follows: if C and D are clauses and T is a term set of $\{D\}$ by some Skolem substitution σ w.r.t. $\{C\}$, then C *T-implies* D w.r.t. T, if $\mathcal{I}(C,T) \models D\sigma$. T-implication is decidable, weaker than logical implication and stronger than subsumption. [IA93, IA95] gives the result that any finite set of clauses has a least generalization under T-implication w.r.t. any term set T. However, as he also notes, T-implication is not transitive and hence not a quasi-order. Therefore it does not fit into our general framework here. For this reason, we will not discuss it fully here, and for the same reason we have not included a row for T-implication in Table 4.1.

Let us now begin with the proof of our result concerning the existence of LGI's. Consider $C = P(x,y,z) \leftarrow P(z,x,y)$, and D, σ and T as above. Then $C \models D$ and also $\mathcal{I}(C,T) \models D\sigma$, since $D\sigma$ is a resolvent of $P(f^2(a), b, c) \leftarrow P(c, f^2(a), b)$ and $P(c, f^2(a), b) \leftarrow P(b, c, f^2(a))$, which are in $\mathcal{I}(C,T)$. As we will show in the next lemma, this holds in general: if $C \models D$ and C is function-free, then we can restrict attention to the ground instances of C instantiated to terms in the term set of D by some σ.

The proof of Lemma 4.3 uses the following idea. Consider a derivation of a clause E from a set Σ of ground clauses. Suppose some of the clauses in Σ contain terms not appearing in E. Then any literals containing these terms in Σ must be resolved away in the derivation. This means that if we replace all the terms in the derivation that are not in E, by some other term t, then the result will be another derivation of E. For example, the left of figure 4.2 shows a derivation of length 1 of E. The term $f^2(b)$ in the parent clauses does not appear in E. If we replace this term by the constant a, the result is another derivation of E (right of the figure).

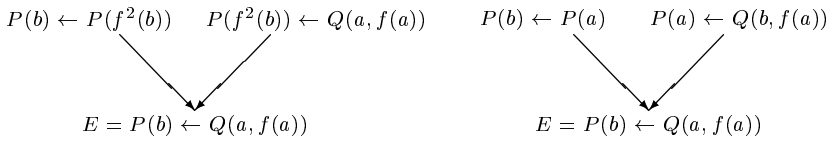


Figure 4.2: Transforming the left derivation yields the right derivation

Lemma 4.3 *Let C be a function-free clause, D be a clause, σ be a Skolem substitution for D w.r.t. $\{C\}$ and T be the term set of D by σ. Then $C \models D$ iff $\mathcal{I}(C,T) \models D\sigma$.*

Proof

$\Leftarrow$: Since $C \models \mathcal{I}(C,T)$ and $\mathcal{I}(C,T) \models D\sigma$, we have $C \models D\sigma$. Now $C \models D$ by Lemma 4.2.

$\Rightarrow$: If D is a tautology, then $D\sigma$ is a tautology, so this case is obvious. Suppose D is not a tautology, then $D\sigma$ is not a tautology. Since $C \models D\sigma$, it follows from Theorem 2.2 that there exists a finite set Σ of ground instances of C, such that $\Sigma \models D\sigma$. By the Subsumption Theorem, there exists a derivation from Σ of a clause E, such that $E \succeq D\sigma$. Since Σ is ground, E must also be ground, so we have $E \subseteq D\sigma$. This implies that E only contains terms from T.

Let t be an arbitrary term in T and let Σ' be obtained from Σ by replacing every term in clauses in Σ which is not in T, by t. Note that since each clause in Σ is a ground instance of the function-free clause C, every clause in Σ' is also a ground instance of C. Now it is easy to see that the same replacement of terms in the derivation of E from Σ results in a derivation of E from Σ': (1) each resolution step in the derivation from Σ can also be carried out in the derivation from Σ', since the same terms in Σ are replaced by the same terms in Σ' and (2) the terms in Σ that are not in T (and hence are replaced by t), do not appear in the conclusion E of the derivation.

Since there is a derivation of E from Σ, we have $\Sigma' \models E$, and hence $\Sigma' \models D\sigma$. Σ' is a set of ground instances of C and all terms in Σ' are terms in T, so $\Sigma' \subseteq \mathcal{I}(C,T)$. Hence $\mathcal{I}(C,T) \models D\sigma$. □

Lemma 4.3 cannot be generalized to the case where C contains function symbols of arity ≥ 1, take $C = P(f(x), y) \leftarrow P(z, x)$ and $D = P(f(a), a) \leftarrow P(a, f(a))$ (from the example given on p. 25 of [IA93]). Then $T = \{a, f(a)\}$ is the term set of D, and we have $C \models D$, yet it can be seen that $\mathcal{I}(C,T) \not\models D$. The argument used in the previous lemma does not work here, because different terms in some ground instance need not relate to different variables. For example, in the ground instance $P(f^2(a), a) \leftarrow P(a, f(a))$ of C, we cannot just replace $f^2(a)$ by some other term, for then the resulting clause would not be an instance of C.

On the other hand, Lemma 4.3 can be generalized to a *set* of clauses instead of a single clause. If Σ is a finite set of function-free clauses, C is an arbitrary clause and σ is a Skolem substitution for C w.r.t. Σ, then we have that $\Sigma \models C$ iff $\mathcal{I}(\Sigma, T) \models C\sigma$. The proof is almost literally the same as above.

This result implies that $\Sigma \models C$ is reducible to an implication $\mathcal{I}(\Sigma, T) \models C\sigma$ between ground clauses. Since, by the next lemma, implication between ground clauses is decidable, it follows that $\Sigma \models C$ is decidable in case Σ is function-free.

Lemma 4.4 *The problem whether $\Sigma \models C$, where Σ is a finite set of ground clauses and C is a ground clause, is decidable.*

Proof Let $C = L_1 \vee \ldots \vee L_n$ and $\mathcal{A}$ be the set of all ground atoms occurring in Σ and C. Now:
$\Sigma \models C$ iff (by Proposition A.1)
$\Sigma \cup \{\neg L_1, \ldots, \neg L_n\}$ is unsatisfiable iff (by Proposition A.4)
$\Sigma \cup \{\neg L_1, \ldots, \neg L_n\}$ has no Herbrand model iff
no subset of $\mathcal{A}$ is an Herbrand model of $\Sigma \cup \{\neg L_1, \ldots, \neg L_n\}$.
Since $\mathcal{A}$ is finite, the last statement is decidable. □

Corollary 4.1 *The problem whether $\Sigma \models C$, where Σ is a finite set of function-free clauses and C is a clause, is decidable.*

The following sequence of results more or less follows the pattern of Lemma 11 to Theorem 14 of Idestam-Almquist's [IA95] (similar to Lemma 3.10 to Theorem 3.14 of [IA93]). There he gives a proof of the existence of a least generalization under T-implication of any finite set of (not necessarily function-free) clauses. We can adjust the proof in such a way that we can use it to establish the existence of an LGI of any finite set of clauses containing at least one non-tautologous and function-free clause.

Lemma 4.5 *Let S be a finite set of non-tautologous clauses, $V = \{x_1, \ldots, x_m\}$ be a set of variables and let $G = \{C_1, C_2, \ldots\}$ be a (possibly infinite) set of generalizations of S under implication. Then the set $G' = \mathcal{I}(C_1, V) \cup \mathcal{I}(C_2, V) \cup \ldots$ is a finite set of clauses.*

Proof Let d be the maximal depth of the terms in clauses in S. It follows from Lemma 4.1 that G (and hence also G') cannot contain terms of depth greater than d, nor predicates, functions or constants other than those in S. The set of literals which can be constructed from predicates in S, and from terms of depth at most d consisting of functions and constants in S and variables in V, is finite. Hence the set of clauses which can be constructed from those literals is also finite. G' is a subset of this set, so G' is a finite set of clauses. □

Lemma 4.6 *Let D be a clause, C be a function-free clause such that $C \models D$, $T = \{t_1, \ldots, t_n\}$ be the term set of D by σ, $V = \{x_1, \ldots, x_m\}$ be a set of variables and $m \geq n$. If E is an LGS of $\mathcal{I}(C, V)$, then $E \models D$.*

Proof Let $\gamma = \{x_1/t_1, \ldots, x_n/t_n, x_{n+1}/t_n, \ldots, x_m/t_n\}$ (it does not matter to which terms the variables $x_{n+1}, \ldots, x_m$ are mapped by γ, as long as they are mapped to terms in T). Suppose $\mathcal{I}(C, V) = \{C\rho_1, \ldots, C\rho_k\}$. Then $\mathcal{I}(C, T) =$

$\{C\rho_1\gamma, \ldots, C\rho_k\gamma\}$. Let E be an LGS of $\mathcal{I}(C, V)$ (note that E must be function-free). Then for every $1 \leq i \leq k$, there are θ_i such that $E\theta_i \subseteq C\rho_i$. This means that $E\theta_i\gamma \subseteq C\rho_i\gamma$ and hence $E\theta_i\gamma \models C\rho_i\gamma$, for every $1 \leq i \leq k$. Therefore $E \models \mathcal{I}(C, T)$.

Since $C \models D$, we know from Lemma 4.1 that constants appearing in C must also appear in D. This means that σ is a Skolem substitution for D w.r.t. $\{C\}$. Then from Lemma 4.3 we know $\mathcal{I}(C, T) \models D\sigma$, hence $E \models D\sigma$. Furthermore, since E is an LGS of $\mathcal{I}(C, V)$, all constants in E also appear in C, hence all constants in E must appear in D. Thus σ is also a Skolem substitution for D w.r.t. $\{E\}$. Therefore $E \models D$ by Lemma 4.2. □

Consider $C = P(x, y, z) \leftarrow P(y, z, x)$ and $D = \leftarrow Q(w)$. Both C and D imply the clause $E = P(x, y, z) \leftarrow P(z, x, y), Q(b)$. Now note that $C \cup D = P(x, y, z) \leftarrow P(y, z, x), Q(w)$ also implies E. This holds for clauses in general, even in the presence of background knowledge Σ. The next lemma is very general, but in this section we only need the special case where C and D are function-free and Σ is empty. We need the general case to prove the existence of a GSR in Section 4.8.

Lemma 4.7 *Let C, D and E be clauses such that C and D are standardized apart, and let Σ be a set of clauses. If $C \models_\Sigma E$ and $D \models_\Sigma E$, then $C \cup D \models_\Sigma E$.*

Proof Suppose $C \models_\Sigma E$ and $D \models_\Sigma E$, and let M be a model of $\Sigma \cup \{C \cup D\}$. Since C and D are standardized apart, the clause $C \cup D$ is equivalent to the formula $\forall(C) \vee \forall(D)$ (where $\forall(C)$ denotes the universally quantified clause C). This means that M is a model of C or a model of D. Furthermore, M is also a model of Σ, so it follows from $\Sigma \cup \{C\} \models E$ or $\Sigma \cup \{D\} \models E$ that M is a model of E. Therefore $\Sigma \cup \{C \cup D\} \models E$, hence $C \cup D \models_\Sigma E$. □

Now we can prove the existence of an LGI of any finite set S of clauses which contains at least one non-tautologous and function-free clause. In fact we can prove something stronger, namely that this LGI is a *special* LGI. This is an LGI that is not only implied, but actually *subsumed* by any other generalization of S:

Definition 4.11 Let $\mathcal{C}$ be a clausal language and S be a finite subset of $\mathcal{C}$. An LGI C of S in $\mathcal{C}$ is called a *special* LGI of S in $\mathcal{C}$, if $C' \succeq C$ for every generalization $C' \in \mathcal{C}$ of S under implication. ◇

Note that if D is an LGI of a set containing at least one non-tautologous function-free clause, then by Lemma 4.1 D is itself function-free, because it should imply the function-free clause(s) in S. For instance, $C = P(x, y, z) \leftarrow P(y, z, x), Q(w)$ is an LGI of $D_1 = P(x, y, z) \leftarrow P(y, z, x), Q(f(a))$ and $D_2 = P(x, y, z) \leftarrow P(z, x, y), Q(b)$. Note that this LGI is properly subsumed by the LGS of $\{D_1, D_2\}$, which is $P(x, y, z) \leftarrow P(x', y', z'), Q(w)$. An LGI may sometimes be the empty clause □, for example if $S = \{P(a), Q(a)\}$.

Theorem 4.4 (Existence of special LGI in $\mathcal{C}$) *Let $\mathcal{C}$ be a clausal language. If S is a finite set of clauses from $\mathcal{C}$, and S contains at least one non-tautologous function-free clause, then there exists a special LGI of S in $\mathcal{C}$.*

Proof Let $S = \{D_1, \ldots, D_n\}$ be a finite set of clauses from $\mathcal{C}$, such that S contains at least one non-tautologous function-free clause. We can assume without loss of generality that S contains no tautologies. Let σ be a Skolem substitution for S, $T = \{t_1, \ldots, t_m\}$ be the term set of S by σ, $V = \{x_1, \ldots, x_m\}$ be a set of variables and $G = \{C_1, C_2, \ldots\}$ be the set of all generalizations of S under implication in $\mathcal{C}$. Note that $\square \in G$, so G is not empty. Since each clause in G must imply the function-free clause(s) in S, it follows from Lemma 4.1 that all members of G are function-free. By Lemma 4.5, the set $G' = \mathcal{I}(C_1, V) \cup \mathcal{I}(C_2, V) \cup \ldots$ is a finite set of clauses. Since G' is finite, the set of $\mathcal{I}(C_i, V)$'s is also finite. For simplicity, let $\{\mathcal{I}(C_1, V), \ldots, \mathcal{I}(C_k, V)\}$ be the set of all distinct $\mathcal{I}(C_i, V)$'s.

Let E_i be an LGS of $\mathcal{I}(C_i, V)$, for every $1 \leq i \leq k$, such that $E_1, \ldots, E_k$ are standardized apart. For every $1 \leq j \leq n$, the term set of D_j by σ is some set $\{t_{j_1}, \ldots, t_{j_s}\} \subseteq T$, such that $m \geq j_s$. From Lemma 4.6, we have that $E_i \models D_j$, for every $1 \leq i \leq k$ and $1 \leq j \leq n$, hence $E_i \models S$. Now let $F = E_1 \cup \ldots \cup E_k$, then we have $F \models S$ from Lemma 4.7 (applying the case of Lemma 4.7 where Σ is empty).

To prove that F is a special LGI of S, it remains to show that $C_j \succeq F$, for every $j \geq 1$. For every $j \geq 1$, there is an i $(1 \leq i \leq k)$, such that $\mathcal{I}(C_j, V) = \mathcal{I}(C_i, V)$. So for this i, E_i is an LGS of $\mathcal{I}(C_j, V)$. C_j is itself also a generalization of $\mathcal{I}(C_j, V)$ under subsumption, hence $C_j \succeq E_i$. Then finally $C_j \succeq F$, since $E_i \subseteq F$. $\square$

As a consequence, we also immediately have the following:

Corollary 4.2 (Existence of LGI for function-free clauses) *Let $\mathcal{C}$ be a clausal language. Then for every finite set of function-free clauses $S \subseteq \mathcal{C}$, there exists an LGI of S in $\mathcal{C}$.*

Proof Let S be a finite set of function-free clauses in $\mathcal{C}$. If S only contains tautologies, any tautology will be an LGI of S. Otherwise, let S' be obtained by deleting all tautologies from S. By the previous theorem, there is a special LGI of S'. Clearly, this is also a special LGI of S itself in $\mathcal{C}$. $\square$

This corollary is not trivial, since even though the number of Herbrand interpretations of a language without function symbols is finite (due to the fact that the number of all possible ground atoms is finite in this case), S may nevertheless be implied by an infinite number of non-equivalent clauses. This may seem like a paradox, since there are only finitely many categories of clauses that can "behave differently" in a *finite* number of finite Herbrand interpretations. Thus it would seem that the number of non-equivalent function-free clauses should also be finite. This is a misunderstanding, since logical implication (and hence also logical equivalence) is defined in terms of *all* interpretations, not just Herbrand interpretations. For instance, define $D_1 = P(a, a)$ and

$P(b,b)$, $C_n = \{P(x_i,x_j) \mid i \neq j, 1 \leq i,j \leq n\}$. Then we have $C_n \models \{D_1, D_2\}$, $C_n \models C_{n+1}$ and $C_{n+1} \not\models C_n$, for every $n \geq 1$, see [LNC94a].

Another interesting consequence of Theorem 4.4 concerns self-saturation (see the Introduction to this chapter for the definition of self-saturation). If C is a special LGI of some set S, then it is clear that C is self-saturated: any clause which implies C also implies S and hence must *subsume* C, since C is a special LGI of S. Now consider $S = \{D\}$, where D is some non-tautologous function-free clause. Then a special LGI C of S will be logically equivalent to D. Moreover, since this C will be self-saturated, it is a self-saturation of D.

Corollary 4.3 *If D is a non-tautologous function-free clause, then there exists a self-saturation of D.*

4.5.2 The LGI is computable

In the previous subsection we proved the *existence* of an LGI in $\mathcal{C}$ of every finite set S of clauses containing at least one non-tautologous function-free clause. In this subsection, we will establish the *computability* of such an LGI. The next algorithm, extracted from the proof of the previous section, computes this LGI of S.

Algorithm 4.1 (LGI-Algorithm)
Input: A finite set S of clauses, containing at least one non-tautologous function-free clause.
Output: An LGI of S in $\mathcal{C}$.

1. Remove all tautologies from S (a clause is a tautology iff it contains literals A and $\neg A$), call the remaining set S'.
2. Let m be the number of distinct terms (including subterms) in S', let $V = \{x_1, \ldots, x_m\}$. (Notice that this m is the same number as the number of terms in the term set T used in the proof of Theorem 4.4.)
3. Let G be the (finite) set of all clauses which can be constructed from predicates and constants in S' and variables in V.
4. Let $\{U_1, \ldots, U_n\}$ be the set of all subsets of G.
5. Let H_i be an LGS of U_i, for every $1 \leq i \leq n$. These H_i can be computed by Plotkin's algorithm [Plo70].
6. Remove from $\{H_1, \ldots, H_n\}$ all clauses which do not imply S' (since each H_i is function-free, by Corollary 4.1 this implication is decidable), and standardize the remaining clauses $\{H_1, \ldots, H_q\}$ apart.
7. Return the clause $H = H_1 \cup \ldots \cup H_q$.

The correctness of this algorithm follows from the proof of Theorem 4.4. First notice that $H \models S$ by Lemma 4.7. Furthermore, note that all $\mathcal{I}(C_i, V)$'s mentioned in the proof of Theorem 4.4, are elements of the set $\{U_1, \ldots, U_n\}$. This means that for every E_i in the set $\{E_1, \ldots, E_k\}$ mentioned in that proof, there is a clause H_j in $\{H_1, \ldots, H_q\}$ such that E_i and H_j are subsume-equivalent. Then it follows that the LGI $F = E_1 \cup \ldots \cup E_k$ of that proof subsumes the clause $H = H_1 \cup \ldots \cup H_q$ that our algorithm returns. On the other hand, F is a special LGI, so F and H must be subsume-equivalent.

Suppose the number of distinct constants in S' is c and the number of distinct variables in step 2 of the algorithm is m. Furthermore, suppose there are p distinct predicate symbols in S', with respective arities $a_1, \ldots, a_p$. Then the number of distinct atoms that can be formed from these constants, variables and predicates, is $l = \sum_{i=1}^{p}(c+m)^{a_i}$, and the number of distinct literals that can be formed, is $2 \cdot l$. The set G of distinct clauses which can be formed from these literals is the power set of this set of literals, so $|G| = 2^{2 \cdot l}$. Then the set $\{U_1, \ldots, U_n\}$ of all subsets of G contains $2^{|G|} = 2^{2^{2 \cdot l}}$ members.

Thus the algorithm outlined above is not very efficient (to say the least). A more efficient algorithm may exist, but since implication is harder than subsumption and the computation of an LGS is already quite expensive, we should not put our hopes too high. Nevertheless, the existence of the LGI-algorithm does establish the theoretical point that the LGI of any finite set of clauses containing at least one non-tautologous function-free clause, is effectively computable.

Theorem 4.5 (Computability of LGI) *Let $\mathcal{C}$ be a clausal language. If S is a finite set of clauses from $\mathcal{C}$, and S contains at least one non-tautologous function-free clause, then the LGI of S in $\mathcal{C}$ is computable.*

4.6 Greatest specializations under implication

Now we turn from least generalizations under implication to greatest specializations. Finding least generalizations of sets of clauses is common practice in ILP. On the other hand, the greatest specialization, which is the dual of the least generalization, is used hardly ever. Nevertheless, the GSI of two clauses D_1 and D_2 might be useful. Suppose that we have one positive example e^+ and two negative examples e_1^- and e_2^-, and suppose that D_1 implies e^+ and e_1^-, while D_2 implies e^+ and e_2^-. Then it might very well be that the GSI of D_1 and D_2 still implies e^+, but is consistent w.r.t. $\{e_1^-, e_2^-\}$. Then we could obtain a correct specialization by taking the GSI of D_1 and D_2.

It is obvious from the previous sections that the existence of an LGI of S is quite hard to establish. For clauses which all contain functions, the existence of an LGI is still an open question and even for the case where S contains at least one non-tautologous function-free clause, the proof was far from trivial. However, the existence of a GSI in $\mathcal{C}$ is much easier to prove. In fact, a GSI of a finite set S is the same as the GSS of S, namely the union of the clauses in S after these are standardized apart.

To see the reason for this dissymmetry, let us take a step back from the clausal framework and consider full first-order logic for a moment. If ϕ_1 and ϕ_2 are two arbitrary first-order formulas, then it can be easily shown that their least generalization is just $\phi_1 \wedge \phi_2$: this conjunction implies ϕ_1 and ϕ_1, and must be implied by any other formula which implies both ϕ_1 and ϕ_2. Dually, the greatest specialization is just $\phi_1 \vee \phi_2$: this is implied by both ϕ_1 and ϕ_2 and must imply any other formula that is implied by both ϕ_1 and ϕ_2. See figure 4.3.

Now suppose ϕ_1 and ϕ_2 are clauses. Then why do we have a problem in finding the LGI of ϕ_1 and ϕ_2? The reason for this is that $\phi_1 \wedge \phi_2$ is not a clause.

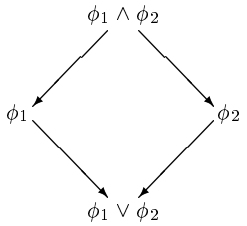


Figure 4.3: Least generalization and greatest specialization in first-order logic

Instead of using $\phi_1 \wedge \phi_2$, we have to find some least clause which implies both clauses ϕ_1 and ϕ_2. Such a clause appears quite hard to find sometimes.

On the other hand, in case of specialization there is no problem. Here we can take $\phi_1 \vee \phi_2$ as GSI, since $\phi_1 \vee \phi_2$ is equivalent to a clause, if we handle the universal quantifiers in front of a clause properly. If ϕ_1 and ϕ_2 are standardized apart, then the formula $\phi_1 \vee \phi_2$ is equivalent to the clause which is the union of ϕ_1 and ϕ_2. This fact was used in the proof of Lemma 4.7.

Suppose $S = \{D_1, \ldots, D_n\}$, and $D'_1, \ldots, D'_n$ are variants of these clauses which are standardized apart. Then clearly $D = D'_1 \cup \ldots \cup D'_n$ is a GSI of S, since it follows from Lemma 4.7 that any specialization of S under implication is implied by D. Thus we have the following result:

Theorem 4.6 (Existence of GSI in $\mathcal{C}$) *Let $\mathcal{C}$ be a clausal language. Then for every finite $S \subseteq \mathcal{C}$, there exists a GSI of S in $\mathcal{C}$.*

The previous theorem holds for clauses in general, so in particular also for function-free clauses. Furthermore, Corollary 4.2 guarantees us that in a function-free clausal language, an LGI of every finite S exists. This means that the set of function-free clauses quasi-ordered by logical implication, is in fact a lattice.

Corollary 4.4 (Lattice-structure of function-free clauses under $\models$) *A function-free clausal language ordered by implication is a lattice.*

In case of a Horn language $\mathcal{H}$, we cannot apply the same proof method as in the case of a clausal language, since the union of two Horn clauses need not be a Horn clause itself. In fact, we can show that not every finite set of Horn clauses has a GSI in $\mathcal{H}$. Here we can use the same clauses that we used to show that sets of Horn clauses need not have an LGI in $\mathcal{H}$, this time from the perspective of specialization instead of generalization.

Again, let $D_1 = P(f^2(x)) \leftarrow P(x)$, $D_2 = P(f^3(x)) \leftarrow P(x)$, $C_1 = P(f(x)) \leftarrow P(x)$ and $C_2 = P(f^2(y)) \leftarrow P(x)$. Then $C_1 \models \{D_1, D_2\}$ and $C_2 \models \{D_1, D_2\}$, and there is no Horn clause D such that $D \models D_1$, $D \models D_2$, $C_1 \models D$ and $C_2 \models D$. Hence there is no GSI of $\{C_1, C_2\}$ in $\mathcal{H}$.

4.7 Least generalizations under relative implication

Implication is stronger than subsumption, but *relative* implication is even more powerful, because background knowledge can be used to model all sorts of useful

properties and relations. In this section, we will discuss least generalizations under implication relative to some given background knowledge Σ (LGR's). In the next section, we treat greatest specializations under relative implication.

First, we will prove the equivalence between our definition of relative implication and a definition given by Niblett [Nib88, p. 133]. He gives the following definition of subsumption relative to a background knowledge Σ (to distinguish it from our notion of subsumption, we will call this 'N-subsumption'):[8]

Definition 4.12 Clause C *N-subsumes* clause D with respect to background knowledge Σ if there is a substitution θ such that $\Sigma \vdash (C\theta \rightarrow D)$ (here '$\rightarrow$' is the implication-connective, and '$\vdash$' is an arbitrary sound and complete proof procedure). $\diamond$

Proposition 4.1 *Let C and D be clauses and Σ be a set of clauses. Then C N-subsumes D with respect to Σ iff $C \models_\Sigma D$.*

Proof C N-subsumes D with respect to Σ iff
There is a θ such that $\Sigma \vdash (C\theta \rightarrow D)$ iff (by sound- and completeness of '$\vdash$')
There is a θ such that $\Sigma \models (C\theta \rightarrow D)$ iff (by Theorem A.1)
There is a θ such that $\Sigma \cup \{C\theta\} \models D$ iff (for the 'if', put $\theta = \varepsilon$)
$\Sigma \cup \{C\} \models D$ iff
$C \models_\Sigma D$. $\square$

Since $\models$ is the special case of $\models_\Sigma$ where Σ is empty, our counterexamples to the existence of LGI's or GSI's in $\mathcal{H}$ are also counterexamples to the existence of LGR's or GSR's in $\mathcal{H}$. In other words, the '$-$'-entries in the second row of Table 4.1 carry over to the third row.

For general clauses, the LGR-question also has a negative answer. We will show here that even if S and Σ are both finite sets of *function-free* clauses, an LGR of S relative to Σ need not exist. Let $D_1 = P(a)$, $D_2 = P(b)$, $S = \{D_1, D_2\}$, and $\Sigma = \{(P(a) \vee \neg Q(x)), (P(b) \vee \neg Q(x))\}$. We will show that this S has no LGR relative to Σ in $\mathcal{C}$.

Suppose C is an LGR of S relative to Σ. Note that if C contains the literal $P(a)$, then the Herbrand interpretation which makes $P(a)$ true and which makes all other ground literals false, would be a model of $\Sigma \cup \{C\}$ but not of D_2, so then we would have $C \not\models_\Sigma D_2$. Similarly, if C contains $P(b)$ then $C \not\models_\Sigma D_1$. Hence C cannot contain $P(a)$ or $P(b)$.

Now let d be a constant not appearing in C. Let $D = P(x) \vee Q(d)$, then $D \models_\Sigma S$. By the definition of an LGR, we should have $D \models_\Sigma C$. Then by the Subsumption Theorem, there must be a derivation from $\Sigma \cup \{D\}$ of a clause E, which subsumes C. The set of all clauses which can be derived (in 0 or more resolution-steps) from $\Sigma \cup \{D\}$ is $\Sigma \cup \{D\} \cup \{(P(a) \vee P(x)), (P(b) \vee P(x))\}$. But none of these clauses subsumes C, because C does not contain the constant d, nor the literals $P(a)$ or $P(b)$. Hence $D \not\models_\Sigma C$, contradicting the assumption that C is an LGR of S relative to Σ in $\mathcal{C}$.

[8] Niblett attributes this definition to Plotkin, though Plotkin gives a rather different definition of relative subsumption, as we have seen in Section 4.4.

As we have seen, in general the LGR of S relative to Σ need not exist. However, we can identify a special case in which the LGR *does* exist. This case might be of practical interest. Suppose $\Sigma = \{L_1, \ldots, L_m\}$ is a finite set of *function-free ground* literals. We can assume Σ does not contain complementary literals (i.e., A and $\neg A$), for otherwise Σ would be inconsistent. Also, suppose $S = \{D_1, \ldots, D_n\}$ is a set of clauses, at least one of which is non-tautologous and function-free. Then $C \models_\Sigma D_i$ iff $\{C\} \cup \Sigma \models D_i$ iff $C \models D_i \vee \neg(L_1 \wedge \ldots \wedge L_m)$ iff $C \models D_i \vee \neg L_1 \vee \ldots \vee \neg L_m$. This means that an LGI of the set of clauses $\{D_1 \vee \neg L_1 \vee \ldots \vee \neg L_m, \ldots, D_n \vee \neg L_1 \vee \ldots \vee \neg L_m\}$ is also an LGR of S relative to Σ. If some $D_k \vee \neg L_1 \vee \ldots \vee \neg L_m$ is non-tautologous and function-free, this LGI exists and is computable. Hence in this special case, the LGR of S relative to Σ exists and is computable.

4.8 Greatest specializations under relative implication

Since the counterexample to the existence of GSI's in $\mathcal{H}$ is also a counterexample to the existence of GSR's in $\mathcal{H}$, the only remaining question in the $\models_\Sigma$-order is the existence of GSR's in $\mathcal{C}$. The answer to this question is positive. In fact, like the GSS and the GSI, the GSR of some finite set S in $\mathcal{C}$ is just the union of the (standardized apart) clauses in S.

Theorem 4.7 (Existence of GSR in $\mathcal{C}$) *Let $\mathcal{C}$ be a clausal language and $\Sigma \subseteq \mathcal{C}$. Then for every finite $S \subseteq \mathcal{C}$, there exists a GSR of S relative to Σ in $\mathcal{C}$.*

Proof Suppose $S = \{D_1, \ldots, D_n\} \subseteq \mathcal{C}$. Without loss of generality, we assume the clauses in S are standardized apart. Let $D = D_1 \cup \ldots \cup D_n$, then $D_i \models_\Sigma D$, for every $1 \leq i \leq n$. Now let $C \in \mathcal{C}$ be such that $D_i \models_\Sigma C$, for every $1 \leq i \leq n$. Then from Lemma 4.7, we have $D \models_\Sigma C$. Hence D is a GSR of S relative to Σ in $\mathcal{C}$. □

4.9 Summary

In ILP, the three main generality orders are subsumption, implication and relative implication. The two main languages are clausal languages and Horn languages. This gives a total of six different ordered sets. In this chapter, we have given a systematic treatment of the existence or non-existence of least generalizations and greatest specializations in each of these six ordered sets. The outcome of this investigation has been summarized in Table 4.1 on p. 55. The only remaining open question is the existence or non-existence of a least generalization under implication in $\mathcal{C}$ for sets of clauses which all contain function symbols.

Table 4.1 makes explicit the trade-off between different generality orders. On the one hand, implication is better suited as a generality order than subsumption, particularly in case of recursive clauses. Relative implication is still

better, because it allows us to take background knowledge into account. On the other hand, we can see from the table that as far as the existence of least generalizations goes, subsumption is more attractive than logical implication, and logical implication is in turn more attractive than relative implication. For subsumption, least generalizations always exist. For logical implication, we can only prove the existence of least generalizations in the presence of a function-free clause. And finally, for relative implication, least generalizations need not even exist in a function-free language. In practice this means that we cannot have it all: if we choose to use a very strong generality order, we have few positive results to go on, whereas if we want to guarantee the existence of least generalizations, we are committed to the weakest generality order: subsumption.

Appendix A

Definitions from Logic

First-order logic was initially conceived by Gottlob Frege [Fre79], and further developed by Alfred North Whitehead and Bertrand Russell [WR27]. Its semantics was developed by Alfred Tarski [Tar36, Tar56].

In this appendix, we include the definitions from first-order logic that are used in this thesis. The appendix is mainly intended to make the thesis self-contained, it does not contain a full discussion with examples. For a more extensive introduction, we refer to [CL73, Llo87, Men87, BJ89].

A.1 Syntax

The syntax of first-order logic defines what constitutes a well-formed formula.

Definition A.1 An *alphabet* consists of the following symbols:

1. A set of *constants*: $a, b, \ldots$, which may be subscripted.
2. A set of *variables*: $u, v, w, x, y, \ldots$, which may be subscripted.
3. A set of *function symbols*: $f, g, \ldots$, which may be subscripted. Each function symbol has a natural number (its *arity*) assigned to it.
4. A non-empty set of *predicate symbols*: $P, Q, \ldots$, which may be subscripted. Each predicate symbol has a natural number (its *arity*) assigned to it.
5. The following five *connectives*: $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$.
6. Two *quantifiers*: $\exists$ (called the *existential* quantifier) and $\forall$ (called the *universal* quantifier).
7. Three *punctuation symbols*: '(', ')' and ','.

$\diamond$

The *arity* of a function or predicate symbol is the number of its arguments.

Definition A.2 *Terms* are defined as follows:

1. A constant is a term.
2. A variable is a term.
3. If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

$\diamond$

Definition A.3 *Well-formed formulas* (or just *formulas*) are defined as follows:

1. If P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula, called an *atom*.
2. If ϕ is a formula, then $\neg\phi$ is a formula.
3. If ϕ and ψ are formulas, then $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are formulas.
4. If ϕ is a formula and x is a variable, then $\exists x\ \phi$ and $\forall x\ \phi$ are formulas.

A formula which is not an atom is called a *composite* formula. ◇

Definition A.4 The *first-order language* given by an alphabet is the set of all formulas which can be constructed from the symbols of the alphabet. ◇

Definition A.5 The *scope* of $\forall x$ (respectively $\exists x$) in $\forall x\ \phi$ (resp. $\exists x\ \phi$) is ϕ. ◇

Definition A.6 A *bound occurrence* of a variable in a formula is an occurrence of this variable immediately following a quantifier or an occurrence within the scope of a quantifier which has the same variable immediately after the quantifier. An occurrence of a variable which is not bound, is called *free*. ◇

Definition A.7 A *closed formula* is a formula which does not contain any free occurrences of variables. ◇

Definition A.8 A *ground term* is a term which does not contain any variables. A *ground formula* is a formula which does not contain any variables. ◇

A.2 Semantics

The semantics of first-order logic is concerned with *interpretations*, which give meaning to the formulas in a language.

A.2.1 Interpretations

A pre-interpretation is a mapping from terms in the language to objects in a *domain*.

Definition A.9 A *pre-interpretation* J of a first-order language L consists of the following:

1. A non-empty set D, called the *domain* of the pre-interpretation.
2. Each constant in L is assigned an element of D.
3. Each n-ary function symbol f in L is assigned a mapping J_f from D^n to D.

◇

The domain D may be either finite or infinite. Here $D^n = \{(d_1, \ldots, d_n) \mid d_i \in D$, for every $1 \leq i \leq n\}$. The mapping from variables to objects in the domain is done by a *variable assignment*:

Definition A.10 Let J be a pre-interpretation with domain D of a first-order language L. A *variable assignment* V with respect to L, is a mapping from the set of variables in L to the domain D of J. We use $V(x/d)$ to denote the variable assignment which maps the variable x to $d \in D$ and maps the other variables according to V. ◇

The combination of a pre-interpretation and a variable assignment assigns an object in the domain to each term in the language:

Definition A.11 Let J be a pre-interpretation with domain D of a first-order language L, and let V be a variable assignment w.r.t. L. The *term assignment* w.r.t. J and V of the terms in L is the following mapping from the set of terms in L to the domain D:

1. Each constant is mapped to an element in D by J.
2. Each variable is mapped to an element in D by V.
3. If $d_1, \ldots, d_n$ are the elements of the domain to which the terms $t_1, \ldots, t_n$ are mapped, respectively, and J_f is the function from D^n to D assigned to the function symbol f by J, then the term $f(t_1, \ldots, t_n)$ is mapped to $J_f(d_1, \ldots, d_n)$.

◇

Given a pre-interpretation, an interpretation is a mapping from formulas to truth-values:

Definition A.12 An *interpretation* I of a first-order language L consists of the following:

1. A pre-interpretation J, with some domain D, of L. I is *based on* J.
2. Each n-ary predicate symbol P in L is assigned a mapping I_P from D^n to $\{T, F\}$.

◇

Definition A.13 Let I be an interpretation, based on the pre-interpretation J with domain D, of the first-order language L, and let V be a variable assignment w.r.t. L. Let Z be the term assignment w.r.t. J and V. Then a formula ϕ in L has a *truth-value under I and V*, as follows:

1. If ϕ is the atom $P(t_1, \ldots, t_n)$ and d_i is the domain-element assigned to t_i by Z $(i = 1, \ldots, n)$, then the truth-value of ϕ under I and V is $I_P(d_1, \ldots, d_n)$.

2. If ϕ is a formula of the form $\neg\psi$, $(\psi \wedge \chi)$, $(\psi \vee \chi)$, $(\psi \rightarrow \chi)$ or $(\psi \leftrightarrow \chi)$, then the truth-value of ϕ is determined by the following truth-table for the five connectives:[1]

[1] As can be seen from this table, the connective '$\neg$' is to be interpreted as 'not', '$\wedge$' as 'and', '$\vee$' as 'or', '$\rightarrow$' as 'if...then' and '$\leftrightarrow$' as 'if, and only if'.

ψ	χ	$\neg\psi$	$(\psi \wedge \chi)$	$(\psi \vee \chi)$	$(\psi \rightarrow \chi)$	$(\psi \leftrightarrow \chi)$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Table A.1: The truth-table for the five connectives

3. If ϕ is a formula of the form $\exists x\ \psi$, then ϕ has truth-value T under I and V if there exists an element $d \in D$ for which ψ has truth-value T under I and $V(x/d)$. Otherwise, ϕ has truth-value F under I and V.

4. If ϕ is a formula of the form $\forall x\ \psi$, then ϕ has truth-value T under I and V if for all elements $d \in D$, ψ has truth-value T under I and $V(x/d)$. Otherwise, ϕ has truth-value F under I and V.

$\diamond$

It is not very difficult to see that the truth-value under some I and V of a *closed* formula does not depend on the variable assignment V. In this thesis, we are only interested in closed formulas, so we can leave out the variable assignment V and speak of "truth-value under I" instead of "truth-value under I and V". Also, when we use the word 'formula' later on, we mean 'closed formula'.

Definition A.14 Let ϕ be a formula in the first-order language L and I an interpretation of L. Then ϕ is said to be *true under* I if its truth-value under I is T. I is then said to *satisfy* ϕ, or to *make* ϕ *true*.

Similarly, ϕ is said to be *false under* I if its truth-value is F under I. I is then said to *falsify* ϕ, or to *make* ϕ *false*. $\diamond$

A.2.2 Models

An interpretation is a *model* of a formula, if it makes that formula true:

Definition A.15 Let ϕ be a formula and I an interpretation. I is said to be a *model* of ϕ if I satisfies ϕ. ϕ is then said to *have* I as a model. $\diamond$

Definition A.16 Let Σ be a set of formulas and I an interpretation. I is said to be a *model* of Σ if I is a model of all formulas $\phi \in \Sigma$. Σ is then said to *have* I as a model. $\diamond$

Definition A.17 Let Σ be a set of formulas and ϕ a formula. Then ϕ is said to be a *logical consequence* of Σ (written as $\Sigma \models \phi$), if every model of Σ is also a model of ϕ. We also sometimes say Σ *(logically) implies* ϕ. If $\Sigma = \{\psi\}$, this can also be written as $\psi \models \phi$. $\diamond$

Definition A.18 Let Σ and ? be sets of formulas. Then ? is said to be a *logical consequence* of Σ (written as $\Sigma \models ?$), if $\Sigma \models \phi$, for every $\phi \in ?$. $\diamond$

If ϕ is not a logical consequence of Σ, we write $\Sigma \not\models \phi$, and similarly $\Sigma \not\models ?$ if not $\Sigma \models ?$.

Definition A.19 Two formulas ϕ and ψ are said to be *(logically) equivalent*, denoted by $\phi \Leftrightarrow \psi$, if both $\phi \models \psi$ and $\psi \models \phi$ (so ϕ and ψ have exactly the same models). Similarly, two sets of formulas Σ and $?$ are said to be *(logically) equivalent*, if both $\Sigma \models ?$ and $? \models \Sigma$. $\diamond$

Definition A.20 Let ϕ be a formula. Then:

1. ϕ is called *valid*, or a *tautology*, if every interpretation is a model of ϕ. This can be written as $\models \phi$. ϕ is called *invalid* otherwise.
2. ϕ is called *satisfiable*, or *consistent*, if some interpretation is a model of ϕ.
3. ϕ is called *inconsistent*, or *unsatisfiable*, or a *contradiction*, if no interpretation is a model of ϕ. In other words, ϕ is inconsistent if it has no models.
4. ϕ is called *contingent* if it is satisfiable, but invalid.

$\diamond$

The above definition subdivides the set of all formulas as pictured in figure A.1.

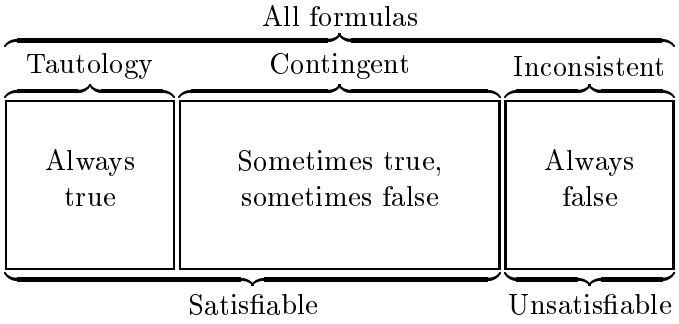


Figure A.1: The class of tautologies, contingent formulas, etc.

These concepts can be defined similarly for a set Σ of formulas. Σ is a tautology if every interpretation is a model of Σ, Σ is satisfiable if it has at least one model, etc. Note that an unsatisfiable set of formulas logically implies anything, since it has no models.

We now state a number of results, whose easy proofs are omitted.

Theorem A.1 (Deduction Theorem) *Let Σ be a set of formulas and ϕ and ψ be formulas. Then $\Sigma \cup \{\phi\} \models \psi$ iff $\Sigma \models (\phi \rightarrow \psi)$.*

Proposition A.1 *Let Σ be a set of formulas and ϕ a formula. Then $\Sigma \models \phi$ iff $\Sigma \cup \{\neg\phi\}$ is unsatisfiable.*

Proposition A.2 *If ϕ and ψ are formulas, then $\phi \Leftrightarrow \psi$ iff $\models (\phi \leftrightarrow \psi)$.*

Proposition A.3 *The following assertions hold.*

1. $\phi \Leftrightarrow \neg\neg\phi$

2. $(\neg\phi \vee \neg\psi) \Leftrightarrow \neg(\phi \wedge \psi)$
3. $(\neg\phi \wedge \neg\psi) \Leftrightarrow \neg(\phi \vee \psi))$
4. $((\phi \vee \psi) \wedge \chi) \Leftrightarrow ((\phi \wedge \chi) \vee (\psi \wedge \chi))$
5. $((\phi \wedge \psi) \vee \chi) \Leftrightarrow ((\phi \vee \chi) \wedge (\psi \vee \chi))$
6. $(\phi \rightarrow \psi) \Leftrightarrow (\neg\phi \vee \psi)$
7. $(\phi \leftrightarrow \psi) \Leftrightarrow ((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$
8. $\forall x\ \phi \Leftrightarrow \neg\exists x\ \neg\phi$
9. $\exists x\ \phi \Leftrightarrow \neg\forall x\ \neg\phi$

For a proof of the following Compactness Theorem, see [BJ89].

Theorem A.2 (Compactness) *If Σ is an infinite, unsatisfiable set of formulas, then there exists a finite, unsatisfiable subset of Σ.*

Note the following consequence of this theorem:

Theorem A.3 *Let Σ be an infinite set of formulas and ϕ be a formula. If $\Sigma \models \phi$, then there is a finite subset Σ' of Σ, such that $\Sigma' \models \phi$.*

Proof If $\Sigma \models \phi$, then by Proposition A.1, $\Sigma \cup \{\neg\phi\}$ is unsatisfiable. By the Compactness Theorem, there is a finite unsatisfiable set $? \subseteq \Sigma \cup \{\neg\phi\}$. Put $\Sigma' = ? \setminus \{\neg\phi\}$. Then $\Sigma' \subseteq \Sigma$, and since $\Sigma' \cup \{\neg\phi\}$ is unsatisfiable, we have $\Sigma' \models \phi$ by Proposition A.1. □

A.2.3 Conventions to simplify notation

In order to avoid an overload of brackets, we can make a number of simplifying conventions. Firstly, we can omit the outer brackets around a formula. Secondly, since $\phi \vee (\psi \vee \chi)$ and $(\phi \vee \psi) \vee \chi$ are equivalent, they can both be written as $\phi \vee \psi \vee \chi$. Such a formula is called a *disjunction*. Similarly, we can write $\phi \wedge \psi \wedge \chi$ (a *conjunction*) instead of $\phi \wedge (\psi \wedge \chi)$ and $(\phi \wedge \psi) \wedge \chi$. Finally, we will sometimes abbreviate iterated function symbols in the following manner: $f^2(a)$ denotes $f(f(a))$, $f^3(a)$ denotes $f(f(f(a)))$, etc.

A.3 Normal forms

In this section we will define two normal forms: prenex conjunctive normal form and Skolem standard form.

A.3.1 Prenex conjunctive normal form

Definition A.21 A *literal* is an atom or the negation of an atom. A *positive literal* is an atom, a *negative literal* is the negation of an atom. ◇

Here we adopt the notational convention that the negation of a negative literal is the atom in that literal: if $L = \neg A$, then $\neg L = A$.

Definition A.22 A *clause* is a finite disjunction of zero or more literals. ◇

A disjunction of *zero* literals is called the *empty clause*, denoted by $\Box$. It represents a contradiction.

Clauses are important, because sets of clauses are commonly used to express theories in Inductive Logic Programming.

Definition A.23 A formula is in *prenex conjunctive normal form* if it has the following form:

$$\underbrace{q_1 x_1 \ldots q_n x_n}_{Prenex} \underbrace{(C_1 \wedge \ldots \wedge C_m)}_{Matrix},$$

where each q_i is either $\exists$ or $\forall$, $x_1, \ldots, x_n$ are all the variables occurring in the formula, and each C_j is a clause. The first part of the formula (the sequence of quantifiers with variables) is called the *prenex* of the formula. The second part is called the *matrix* of the formula[2], which we sometimes abbreviate to $M[x_1, \ldots, x_m]$. $\diamond$

In fact, any formula ϕ can be transformed into an equivalent formula ψ in prenex conjunctive normal form (see pp. 35–39 of [CL73] or Proposition 3.4 of [Llo87]):

Theorem A.4 *Let ϕ be a formula. Then there exists a formula ψ in prenex conjunctive normal form, such that ϕ and ψ are equivalent.*

A.3.2 Skolem standard form

This section discusses the Skolem standard form of a formula. It is obtained from the prenex conjunctive normal form by replacing existentially quantified variables by functional terms.

Definition A.24 Let $\phi = q_1 x_1 \ldots q_n x_n M[x_1, \ldots, x_m]$ be a formula in prenex conjunctive normal form. Then a *Skolemized form* of ϕ is a formula ϕ' obtained by applying the following procedure to ϕ:

1. Set ϕ' to ϕ.
2. If the prenex of ϕ' contains only universal quantifiers, then stop.
3. Let q_i be the first (from the left) existential quantifier in ϕ'. Let $x_{i_1}, \ldots, x_{i_j}$ be the variables on the left of x_i (that is, those variables from $x_1, \ldots, x_{i-1}$ that have not been deleted).
4. Add a new j-ary function symbol, which we denote here by f, to the alphabet. Replace each occurrence of x_i in the matrix of ϕ' by the term $f(x_{i_1}, \ldots, x_{i_j})$. If there are no universal quantifiers to the left of x_i in ϕ', then replace each occurrence of x_i by a new constant which is added to the alphabet.
5. Delete $\exists x_i$ from the prenex of ϕ'.
6. Goto step number 2.

The new function symbols and constants which are added to the alphabet are called *Skolem functions* and *Skolem constants*, respectively. $\diamond$

[2]This term 'matrix' is just a name we use, it does not have very much in common with the mathematical concept of a matrix.

For example, $\forall x \forall y\ P(x, y, f(x, y))$ is a Skolemized form of $\forall x \forall y \exists z\ P(x, y, z)$, obtained by replacing the existentially quantified variable z by the term $f(x, y)$. The standard form of a formula is a conjunction of universally quantified clauses.

Definition A.25 Let ϕ be a (not necessarily closed) formula and $x_1, \ldots, x_n$ be all distinct variables that occur free in ϕ. Then $\forall(\phi)$ denotes the closed formula $\forall x_1 \ldots \forall x_n\ \phi$. $\diamond$

Definition A.26 Let ϕ be a formula, ϕ' be a prenex conjunctive normal form of ϕ and $\phi'' = \forall(C_1 \wedge \ldots \wedge C_n)$ be a Skolemized form of ϕ'. Then $\psi = \forall(C_1) \wedge \ldots \wedge \forall(C_n)$ is called a *Skolem standard form* (or just a *standard form*) of ϕ. We say ϕ *has* ψ as a standard form. $\diamond$

The standard form of a set $\{\phi_1, \ldots, \phi_n\}$ of formulas is simply the standard form of the formula $\phi_1 \wedge \ldots \wedge \phi_n$.

A standard form $\forall(C_1) \wedge \ldots \wedge \forall(C_n)$ can also be written as a set $\{C_1, \ldots, C_n\}$ of clauses. When we are dealing with clauses, we will assume each clause to be universally quantified. So for instance, if $\Sigma = \{C_1, \ldots, C_n\}$ is a set of clauses and C is a clause, we use $\Sigma \models C$ to abbreviate $\forall(C_1) \wedge \ldots \wedge \forall(C_n) \models \forall(C)$.

Putting a formula in standard from does not preserve logical equivalence. For instance, $P(a)$ is a standard form of $\exists x\ P(x)$, but $\exists x\ P(x) \not\Leftrightarrow P(a)$, because $\exists x\ P(x) \not\models P(a)$. However, by the following theorem (Theorem 4.1 of [CL73]), standard form does preserve unsatisfiability.

Theorem A.5 *Let ϕ be a formula and ψ be a standard form of ϕ. Then ϕ is unsatisfiable iff ψ is unsatisfiable.*

A.4 Herbrand interpretations

Herbrand interpretations are interpretations which have the set of ground terms as domain and which interpret each ground term in the language as that same ground term in the domain.

Definition A.27 Let L be a first-order language. The *Herbrand universe* U_L for L is the set of all ground terms which can be formed out of the constants and function symbols in L. In case L does not contain any constants, we add one arbitrary constant to the alphabet to be able to form ground terms. $\diamond$

Definition A.28 Let L be a first-order language. The *Herbrand base* B_L for L is the set of all ground atoms which can be formed out of the predicate symbols in L and the terms in the Herbrand universe U_L. $\diamond$

Definition A.29 Let L be a first-order language. The *Herbrand pre-interpretation* for L is the pre-interpretation consisting of the following:

1. The domain of the pre-interpretation is the Herbrand universe U_L.
2. Constants in L are assigned themselves in U_L.

3. Each n-ary function symbol f in L is assigned the mapping J_f from U_L^n to U_L, defined by $J_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$.

$\diamond$

Definition A.30 Let L be a first-order language and J an Herbrand pre-interpretation. Any interpretation based on J is called an *Herbrand interpretation.* $\diamond$

An Herbrand interpretation I can be identified with the set of ground atoms that are true under I.

Definition A.31 Let I be an Herbrand interpretation of a first-order language L. If I is a model of Σ, it is called an *Herbrand model* of Σ. $\diamond$

In this thesis, we will simply assume some fixed language L, and speak of *interpretations*, instead of interpretations *of* L.

The following result (see [CL73, Theorem 4.2] or [Llo87, Proposition 3.3]) shows that when we are dealing with clauses, we can restrict attention to Herbrand models.

Proposition A.4 *A set of clauses Σ has a model iff Σ has an Herbrand model.*

A.5 Horn clauses

Horn clauses are a restricted, but very useful kind of clauses.

Definition A.32 A *definite program clause* is a clause containing one positive, and zero or more negative literals. A *(definite) goal* is a clause containing only negative literals. A *Horn clause* is either a definite program clause, or a definite goal. $\diamond$

If a definite program clause consists of the positive literal A and the negative literals $\neg B_1, \ldots \neg B_n$, then such a clause can be written as the following implication:

$$(B_1 \wedge \ldots \wedge B_n) \rightarrow A.$$

In most papers and books about Logic Programming, this is written as:

$$A \leftarrow B_1, \ldots, B_n.$$

A is called the *head* of the clause, $B_1, \ldots, B_n$ is called the *body* of the clause. It will be convenient to denote the head of a clause C by C^+ and the body by C^-. In case of an atom A (that is, if $n = 0$), we can omit the '$\leftarrow$'-symbol. A definite goal can be written as

$$\leftarrow B_1, \ldots, B_n.$$

The empty clause $\square$ is also considered to be a goal.

Definition A.33 A *definite program* is a finite set of definite program clauses. ◇

Proposition A.5 (Proposition 6.1 of [Llo87]) *Let* Π *be a definite program. If* $\{M_1, M_2, \ldots, M_k, \ldots\}$ *is a (possibly infinite) set of Herbrand models of* Π*, then their intersection* $M = \cap_i M_i$ *is also an Herbrand model of* Π*.*

It follows from the previous proposition that the intersection of *all* Herbrand models of Π, which will be called the *least* Herbrand model, is itself also an Herbrand model of Π.

Definition A.34 Let Π be a definite program. The intersection of all Herbrand models of Π is called the *least Herbrand model* of Π, and is denoted by M_Π. ◇

Theorem A.6 (Theorem 6.2 of [Llo87]) *If* Π *is a definite program, then* $M_\Pi = \{A \in B_\Pi \mid \Pi \models A\}$*.*

A.6 Substitution and unification

A.6.1 Substitution

A substitution replaces variables by terms.

Definition A.35 A *substitution* θ is a finite set $\{x_1/t_1, \ldots, x_n/t_n\}$, $n \geq 0$, where the x_i are distinct variables and the t_i are terms. We say t_i is *substituted for* x_i. x_i/t_i is called a *binding* for x_i. The substitution θ is called a *ground substitution* if every t_i is ground.

The substitution given by the empty set ($n = 0$) is called the *identity substitution*, or the *empty substitution*, and is denoted by ε. ◇

Definition A.36 An *expression* is either a term, a literal, or a conjunction or disjunction of literals. A *simple expression* is a term or a literal. ◇

Definition A.37 Let $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution, and E an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of x_i by t_i, $1 \leq i \leq n$. $E\theta$ is called a *ground instance* of E if $E\theta$ is ground. ◇

Definition A.38 Let $\theta = \{x_1/s_1, \ldots, x_m/s_m\}$ and $\sigma = \{y_1/t_1, \ldots, y_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ is the substitution obtained from $\{x_1/(s_1\sigma), \ldots, x_m/(s_m\sigma), y_1/t_1, \ldots, y_n/t_n\}$ by deleting any binding $x_i/(s_i\sigma)$ for which $x_i = (s_i\sigma)$, and any binding y_j/t_j for which $y_j \in \{x_1, \ldots, x_m\}$. ◇

Definition A.39 Let θ and σ be substitutions. We say θ is an *instance* of σ if there exists a substitution γ such that $\sigma\gamma = \theta$. ◇

Proposition A.6 (Proposition 4.1 of [Llo87]) *Let E be an expression and θ, σ and γ be substitutions. Then the following hold:*

1. $\theta = \theta\varepsilon = \varepsilon\theta$.
2. $(E\theta)\sigma = E(\theta\sigma)$.
3. $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

Since $(\theta\sigma)\gamma = \theta(\sigma\gamma)$, we can omit brackets between substitutions.

Definition A.40 Let E be an expression and $\theta = \{x_1/y_1, \ldots, x_n/y_n\}$ be a substitution. We say θ is a *renaming substitution for* E if each x_i occurs in E, and $y_1, \ldots, y_n$ are distinct variables such that each y_i is either equal to some x_j in θ, or y_i does not occur in E. ◇

Definition A.41 Let E and F be expressions. We say E and F are *variants*, or E is a variant of F, if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. ◇

It is easy to show that if E and F are variants, then there are *renaming* substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$.

We will sometimes need a *Skolem substitution*, which substitutes new constants for the variables in a clause.

Definition A.42 Let Σ be a set of clauses, C be a clause, $x_1, \ldots, x_n$ be all the variables appearing in C and $a_1, \ldots, a_n$ be distinct constants not appearing in Σ or C. Then the substitution $\{x_1/a_1, \ldots, x_n/a_n\}$ is called a *Skolem substitution* for C w.r.t. Σ. ◇

A.6.2 Unification

A *unifier* for the set of expressions $\{E_1, E_2, \ldots, E_n\}$ is a substitution θ such that $E_1\theta = E_2\theta = \ldots = E_n\theta$.

Definition A.43 Let Σ be a finite set of expressions. A substitution θ is called a *unifier for* Σ if $\Sigma\theta$ is a singleton (a set containing exactly one element). If there exists a unifier for Σ, we say Σ is *unifiable*. ◇

Definition A.44 If θ is a unifier for Σ and if for any unifier σ for Σ there exists a substitution γ such that $\sigma = \theta\gamma$, then θ is called a *most general unifier* (abbreviated to *mgu*) for Σ. ◇

It can be shown that any finite unifiable set of simple expressions has an mgu (see Theorem 4.3 of [Llo87] or Theorem 5.2 of [CL73]).

Bibliography

[AGB95] Zoltán Alexin, Tibor Gyimóthy, and Henrik Boström. Integrating algorithmic debugging and unfolding transformation in an interactive learner. In De Raedt [DR95], pages 437–453.

[Aha92] David W. Aha. Relating relational learning algorithms. In Muggleton [Mug92b], pages 233–254.

[Ari60] Aristotle. *Posterior Analytics.* Harvard University Press, Cambridge (MA), 1960. Edited and translated by Hugh Tredennick.

[Bac20] Francis Bacon. *Novum Organum.* 1620.

[Ban64] Ranan B. Banerji. A language for the description of concepts. *General Systems*, 9:135–141, 1964.

[BG96] Francesco Bergadano and Daniele Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1996.

[BGA56] J. S. Bruner, J. J. Goodnow, and G. A. Austin. *A Study of Thinking.* Wiley, New York, 1956.

[BIA94] Henrik Boström and Peter Idestam-Almquist. Specialization of logic programs by pruning SLD-trees. In Wrobel [Wro94], pages 31–48.

[BJ89] George S. Boolos and Richard C. Jeffrey. *Computability and Logic.* Cambridge University Press, Cambridge (UK), third edition, 1989.

[BM92] Michael Bain and Stephen Muggleton. Non-monotonic learning. In Muggleton [Mug92b], pages 145–153.

[Bos95a] Henrik Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1194–1200. Morgan Kaufmann, 1995.

[Bos95b] Henrik Boström. Specialization of recursive predicates. In N. Lavrač and S. Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning (ECML-95)*, volume 912 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, 1995.

[Bra93] Pavel B. Brazdil, editor. *Proceedings of the 6th European Conference on Machine Learning (ECML-93)*, volume 667 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1993.

[Bun86] Wray Buntine. Generalized subsumption. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-86)*, 1986.

[Bun88] Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36:149–176, 1988.

[Car50] Rudolf Carnap. *Logical Foundations of Probability*. Routledge & Kegan Paul, London, 1950.

[Car52] Rudolf Carnap. *The Continuum of Inductive Methods*. The University of Chicago Press, Chicago, 1952.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, San Diego, 1973.

[DK96] Yannis Dimopoulos and Antonis Kakas. Abduction and inductive learning. In De Raedt [DR96], pages 144–171.

[DR95] Luc De Raedt, editor. *Proceedings of the 5th International Workshop on Inductive Logic Programming (ILP-95)*. Katholieke Universiteit Leuven, 1995.

[DR96] Luc De Raedt, editor. *Advances in Inductive Logic Programming*. IOS Press, Amsterdam, 1996.

[DRB93] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1058–1063. Morgan Kaufmann, 1993.

[DRD94] Luc De Raedt and Sašo Džeroski. First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[Fla92] Peter A. Flach. A framework for Inductive Logic Programming. In Muggleton [Mug92b], pages 193–211.

[Fla94] Peter A. Flach. Inductive Logic Programming and philosophy of science. In Wrobel [Wro94], pages 71–84.

[Fla95] Peter A. Flach. *Conjectures: An Inquiry Concerning the Logic of Induction*. PhD thesis, Tilburg University, 1995.

[Fre79] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, 1879. English translation in [Hei77].

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Palo Alto (CA), 1987.

[Goo83] Nelson Goodman. *Fact, Fiction, and Forecast*. Harvard University Press, Cambridge (MA), fourth edition, 1983.

[Got87] Georg Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.

[Hei77] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge (MA), 1977.

[Hel89] Nicolas Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.

[Hem45a] Carl G. Hempel. Studies in the logic of confirmation (part I). *Mind*, 54(213):1–26, 1945.

[Hem45b] Carl G. Hempel. Studies in the logic of confirmation (part II). *Mind*, 54(214):97–121, 1945.

[Hem66] Carl G. Hempel. *Philosophy of Natural Science*. Prentice-Hall, Englewood Cliffs, 1966.

[Hom24] Homer. *The Iliad*. Harvard University Press, Cambridge (MA), 1924. Translated by A. T. Murray. Two volumes.

[Hum56] David Hume. *An Enquiry Concerning Human Understanding*. Gateway edition, Chicago, 1956. Originally 1748.

[Hum61] David Hume. *A Treatise of Human Nature*. Dolphin Books. Doubleday, 1961. Originally 1739–1740.

[IA93] Peter Idestam-Almquist. *Generalization of Clauses*. PhD thesis, Stockholm University, 1993.

[IA95] Peter Idestam-Almquist. Generalization of clauses under implication. *Journal of Artificial Intelligence Research*, 3:467–489, 1995.

[Ino92] Katsumi Inoue. Linear resolution for consequence finding. *Artificial Intelligence*, 56:301–353, 1992.

[Jev74] Stanley Jevons. *The Principles of Science: A Treatise*. MacMillan, 1874.

[KK71] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kow70] Robert A. Kowalski. The case for using equality axioms in automatic demonstration. In *Proceedings of the Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 112–127. Springer-Verlag, 1970.

[Kow74] Robert A. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.

[Kuh77] Thomas Kuhn. Second thoughts on paradigms. In Frederic Suppe, editor, *The Structure of Scientific Theories*, pages 459–482. University of Illinois Press, second edition, 1977.

[LD92a] Nada Lavrač and Sašo Džeroski. Inductive learning of relations from noisy examples. In Muggleton [Mug92b], pages 495–516.

[LD92b] Nada Lavrač and Sašo Džeroski. Refinement graphs for FOIL and LINUS. In Muggleton [Mug92b], pages 319–333.

[LD94] Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[LDG91] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning non-recursive definitions of relations with LINUS. In Y. Kodratoff, editor, *Proceedings of the 6th European Working Sessions on Learning (EWSL-91)*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 265–281. Springer-Verlag, 1991.

[Lee67] Richard Char-Tung Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms*. PhD thesis, University of California, Berkeley, 1967.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[LNC94a] Patrick R. J. van der Laag and Shan-Hwei Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning (ECML-94)*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 307–322. Springer-Verlag, 1994.

[LNC94b] Patrick R. J. van der Laag and Shan-Hwei Nienhuys-Cheng. A note on ideal refinement operators in inductive logic programming. In Wrobel [Wro94], pages 247–262.

[Lov70] Donald W. Loveland. A linear format for resolution. In *Proceedings of the IRIA Symposium on Automatic Demonstration, Versailles, France, 1968*, pages 147–162. Springer-Verlag, 1970.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. North Holland, New York, 1978.

[Luc70] Donald Luckham. Refinements in resolution theory. In *Proceedings of the IRIA Symposium on Automatic Demonstration, Versailles, France, 1968*, pages 163–190. Springer-Verlag, 1970.

[MB88] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In John Laird, editor, *Proceedings of the 5th International Conference on Machine Learning (ICML-88)*, pages 339–352. Morgan Kaufmann, 1988.

[MDR94] Stephen Muggleton and Luc De Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19–20:629–679, 1994.

[Men87] Elliott Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks, Belmont (CA), third edition, 1987.

[MF92] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In Muggleton [Mug92b], pages 281–298.

[Mil58] John Stuart Mill. *A System of Logic, Ratiocinative and Inductive*. Harper, New York, 1858.

[Min68] Marvin L. Minsky, editor. *Semantic Information Processing*. The MIT Press, Cambridge (MA), 1968.

[Mit82] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[MP92] Jerzy Marcinkowski and Leszek Pacholski. Undecidability of the horn-clause implication problem. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 354–362, Pittsburg, 1992.

[MP94] Stephen Muggleton and C. David Page. Self-saturation of definite clauses. In Wrobel [Wro94], pages 161–174.

[MR72] Eliana Minicozzi and Raymond Reiter. A note on linear resolution strategies in consequence-finding. *Artificial Intelligence*, 3:175–180, 1972.

[Mug87] Stephen Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 287–292. Morgan Kaufmann, 1987.

[Mug90] Stephen Muggleton. Inductive Logic Programming. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.

[Mug91] Stephen Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–317, 1991.

[Mug92a] Stephen Muggleton. Inductive logic programming. In *Inductive Logic Programming* [Mug92b], pages 3–27.

[Mug92b] Stephen Muggleton, editor. *Inductive Logic Programming*, volume 38 of *APIC Series*. Academic Press, 1992.

[Mug92c] Stephen Muggleton. Inverting implication. In S. Muggleton and K. Furukawa, editors, *Proceedings of the 2nd International Workshop on Inductive Logic Programming (ILP-92)*, Tokyo, 1992. ICOT Research Center. ICOT Technical Memorandum TM-1182.

[MWKE93] K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications*. Academic Press, London, 1993.

[NCLT93] Shan-Hwei Nienhuys-Cheng, Patrick R. J. van der Laag, and Leon van der Torre. Constructing refinement operators by deconstructing logical implication. In Pietro Torasso, editor, *Proceedings of the 3rd Conference of the Italian Association for Artificial Intelligence (AI*IA-93)*, volume 728 of *Lecture Notes in Artificial Intelligence*, pages 178–189. Springer-Verlag, 1993.

[NCW95a] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The equivalence of the subsumption theorem and the refutation-completeness for unconstrained resolution. In K. Kanchanasut and J.-J. Lévy, editors, *Proceedings of the Asean Computer Science Conference (ACSC-95)*, volume 1023 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, 1995.

[NCW95b] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The specialization problem and the completeness of unfolding. In J. C. van Vliet, editor, *Proceedings of Computing Science in the Netherlands (CSN-95)*, pages 155–169. SION, 1995.

[NCW95c] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Specializing definite programs by unfolding. In *Proceedings of Benelearn'95*. Université Libre de Bruxelles, 1995.

[NCW95d] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The subsumption theorem in Inductive Logic Programming: Facts and fallacies. In De Raedt [DR95], pages 147–160.

[NCW95e] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The subsumption theorem revisited: Restricted to SLD-resolution. In J. C. van Vliet, editor, *Proceedings of Computing Science in the Netherlands (CSN-95)*, pages 143–154. SION, 1995.

[NCW95f] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Tidying up the mess around the subsumption theorem in Inductive Logic Programming. In J. C. Bioch and Y.-H. Tan, editors, *Proceedings of the Sev-*

enth Dutch Conference on Artificial Intelligence (NAIC-95), pages 221–230. Erasmus University Rotterdam, 1995.

[NCW96a] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. A complete method for program specialization based on unfolding. In Wolfgang Wahlster, editor, *Proceedings of the European Conference on Artificial Intelligence (ECAI-96)*, pages 438–442, 1996. In press.

[NCW96b] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Least generalizations and greatest specializations of sets of clauses. *Journal of Artificial Intelligence Research*, 4:341–363, 1996.

[NCW96c] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The specialization problem and the completeness of unfolding. *Machine Learning*, 1996. Submitted.

[NCW96d] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. The subsumption theorem in Inductive Logic Programming: Facts and fallacies. In De Raedt [DR96], pages 265–276.

[Nib88] Tim Niblett. A study of generalisation in logic programs. In D. Sleeman, editor, *Proceedings of the 3rd European Working Sessions on Learning (EWSL-88)*, pages 131–138, 1988.

[Pei58] Charles Sanders Peirce. *Collected Papers.* Harvard University Press, Cambridge (MA), 1958. Edited by Charles Harstshorne and Paul Weiss. Volumes I–VII.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[Plo71a] Gordon D. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, 1971.

[Plo71b] Gordon D. Plotkin. A further note on inductive generalization. *Machine Intelligence*, 6:101–124, 1971.

[Pop59] Karl R. Popper. *The Logic of Scientific Discovery.* Hutchinson, London, 1959.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In Brazdil [Bra93], pages 3–20.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Rei49] Hans Reichenbach. *The Theory of Probability.* University of California Press, Berkeley and Los Angeles, 1949.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[Rob65] J. Alan Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Rou92] Céline Rouveirol. Extensions of inversion of resolution applied to theory completion. In Muggleton [Mug92b], pages 63–92.

[Rus48] Bertrand Russell. *Human Knowledge: It's Scope and Limits.* George Allen and Unwin Ltd., London, 1948.

[Rus80] Bertrand Russell. *The Problems of Philosophy.* Oxford University Press, 1980. Originally 1912.

[SA93] Taisuke Sato and Sumitaka Akiba. Inductive resolution. In *Proceedings of the 4th International Workshop on Algorithmic Learning Theory (ALT-93)*, volume 744 of *Lecture Notes in Artificial Intelligence*, pages 101–110. Springer-Verlag, 1993.

[Sam81] Claude A. Sammut. *Learning Concepts by Performing Experiments.* PhD thesis, University of New South Wales, 1981.

[Sam93] Claude A. Sammut. The origins of Inductive Logic Programming: A prehistoric tale. In Stephen Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming (ILP-93)*, pages 127–147, Ljubljana, 1993. Jožef Stefan Institute. Technical Report IJS-DP-6706.

[SB86] C. A. Sammut and R. B. Banerji. Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 167–192. Morgan Kaufmann, Los Altos (CA), 1986.

[SCL69] J. R. Slagle, C. L. Chang, and R. C. T. Lee. Completeness theorems for semantic resolution in consequence-finding. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI-69)*, pages 281–285, 1969.

[Sha81a] Ehud Y. Shapiro. An algorithm that infers theories from facts. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 446–451, Vancouver, 1981. Morgan Kaufmann.

[Sha81b] Ehud Y. Shapiro. Inductive inference of theories from facts. Research Report 192, Yale University, 1981.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[Tar36] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, pages 261–405, 1936. English translation in [Tar56].

[Tar56] Alfred Tarski. *Logic, Semantics, Metamathematics. Papers from 1923 to 1938.* Oxford University Press, New York, 1956.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of the 2nd International Logic Programming Conference*, pages 127–138, Uppsala, 1984. Uppsala University.

[UM82] P. Utgoff and T. M. Mitchell. Acquisition of appropriate bias for inductive concept learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 414–417, Los Altos (CA), 1982. Morgan Kaufmann.

[Ver75] Steven Vere. Induction of concepts in the predicate calculus. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI-75)*, pages 351–356, 1975.

[Ver77] Steven Vere. Induction of relational productions in the presence of background information. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, 1977.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, 1927. Originally 1910–1913.

[Wro93] Stefan Wrobel. On the proper definition of minimality in specialization and theory revision. In Brazdil [Bra93], pages 65–82.

[Wro94] Stefan Wrobel, editor. *Proceedings of the 4th International Workshop on Inductive Logic Programming (ILP-94)*, volume 237 of *GMD-Studien*, Bad Honnef/Bonn, 1994. Gesellschaft für Mathematik und Datenverarbeitung.

Author Index

Subject Index