Science of Computational Logic
— Working Material[1] —

Steffen Hölldobler
International Center for Computational Logic
Technische Universität Dresden
D–01062 Dresden
sh@iccl.tu-dresden.de

December 11, 2012

[1] The working material is incomplete and may contain errors. Any suggestions are greatly appreciated.

Contents

Notation

In this book we will make the following notational conventions:

a	constant
b	constant
C	unary relation symbol denoting a concept
$\mathcal{C}$	set of concept formulas
$\mathcal{D}$	non-empty domain of an interpretaion
$\mathcal{E}$	set of equations
$E\lceil s \rceil$	expression containing an occurrence of the term s
$E\lceil s/t \rceil$	expression where an occurrence of the term s has been replaced by t
$\mathcal{E}_{\mathcal{R}}$	equational system obtained from the term rewriting system $\mathcal{R}$
$\mathcal{E}_{\approx}$	axioms of equality
ε	empty substitution
g	function symbol
f	function symbol
F	formula
$\mathcal{F}$	set of formulas
g	function symbol
G	formula
H	formula
I	interpretation
$\mathcal{K}$	a set of formulas often called knowledge base
l	term; left-hand side of an equation or rewrite rule
L	literal
p	relation symbol
r	term; the right-hand side of equations or rewrite rules
R	binary relation symbol denoting a role
$\mathcal{R}$	term rewriting system
s	term
t	term
θ	substitution
u	term
U	variable
V	variable
W	variable
X	variable
Y	variable
Z	variable

In addition, we will consider the following precedence hierarchy among connectives:

$$\{\forall, \exists\} \succ \neg \succ \{\wedge, \vee\} \succ \rightarrow \succ \leftrightarrow .$$

Chapter 1

Description Logic

In the late 1960s and early 1970s, it was recognized that knowledge representation and reasoning is at the heart of any intelligent system. Heavily influenced by the work of Quillian on so-called semantic networks [Qui68] and the work of Minsky on so-called frames [Min75] simple graphs and structured objects were used to represent knowledge and many algorithms were developed which manipulated these data structures. At first sight, these systems were quite attractive because they apparently admitted an intuitive semantics, which was easy to understand. For example, a graph like the one shown in Figure 1.1 seems to represent the following short story.

> *Dogs, cats and mice are mammals. Dogs dislike cats and, in particular, the dog Rudi, which is a German shepherd, has bitten the cat Tom while Tom was chasing the mouse Jerry.*

Simple algorithms operating on this graph can be applied to conclude that, for example, German shepherds are mammals, Rudi dislikes Tom, etc.

Shortly afterwards, however, it was recognized that systems based on these techniques lack a formal semantics (see e.g. [Woo75]). What precisely is denoted by a link? What precisely is denoted by a vertex? It was also observed that the algorithms which operated on these data structures did not always yield the intended results. This led to a formal reconstruction of semantic networks as well as frame systems within logic (see e.g. [Sch76, Hay79]). At around the same time, Brachman developed the idea that formally defined concepts should be interrelated and organized in networks such that the structure of these networks allows reasoning about possible conclusions [Bra78]. This line of research led to the knowledge representation and reasoning system KL-ONE [BS85], which is the ancester of a whole family of systems. Such systems have been used in a wide range of practical applications including financial management systems, computer configuration systems, software information systems and database interfaces. KL-ONE has also led to a thorough investigation of the semantics of the representations used in these systems and the development of correct and complete algorithms for computing with these representations. Today the field is called *description logic* and this chapter gives an introduction into such logics.

Description logics focus on descriptions of concepts and their interrelationships in certain domains. Based on so-called *atomic concepts* and relations between concepts, which are traditionally called *roles*, more complex *concepts* are formed with the help of certain

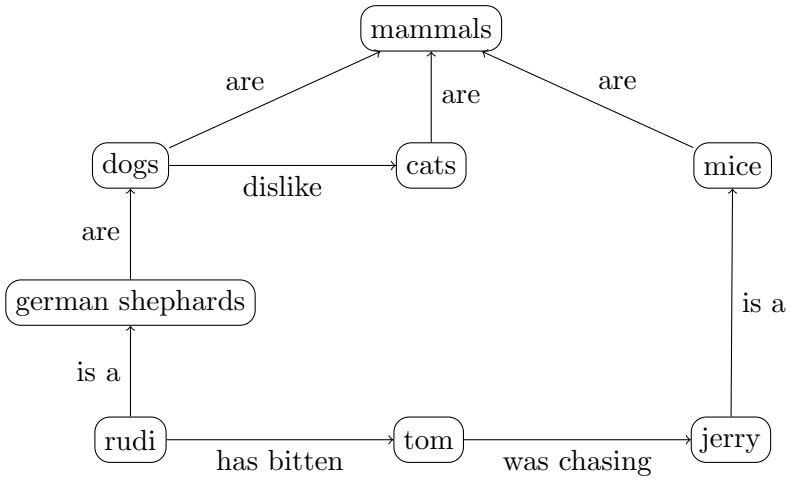


Figure 1.1: A simple semantic network with apparently obvious intended meaning.

operators. Furthermore, assertions about certain aspects of the world can be made. For example, a certain individual may be an instance of a certain concept or two individuals are connected via a certain role. The basic inference tasks provided by description logics are *subsumption* and *unsatisfiability testing*. Subsumption is used to check whether a category is a subset of another category. As we shall see in the next paragraph, description logics do not allow the specification of subsumption hierarchies explicitly but these hierarchies depend on the definitions of the concepts. The unsatisfiability check allows the determination of whether an individual belongs to a certain concept. A formal account of these notions will be developed in the following sections.

1.1 Terminologies

We consider an alphabet with constant symbols, the variables X, Y, ..., the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, the quantifiers $\forall$ and $\exists$, and the special symbols $($, ,, $)$. For notational convenience, C shall denote a unary relation symbols and R a binary relation symbol in the sequel. Informally, C denotes a concept whereas R denotes a role.

C
R

Terms are defined as usual, ie., the set of terms is the union of the set of constant symbols and the set of variables. The set of *role formulas* consists of all strings of the form $R(X, Y)$. The set of *atomic concept formulas* consists of all strings of the form $C(X)$. As we will see shortly, each concept formula contains precisely one free variable. Hence, concept formulas will be denoted by $F(X)$ and $G(X)$, where X is the only free variable occurring in F and G. The set of *concept formulas* is the smallest set $\mathcal{C}$ satisfying the following conditions:

concept formula

1. All atomic formulas are in $\mathcal{C}$.
2. If $F(X)$ is in $\mathcal{C}$, so is $\neg F(X)$.
3. If $F(X)$ and $G(X)$ are in $\mathcal{C}$, so are $F(X) \wedge G(X)$ and $F(X) \vee G(X)$.

4. if $R(X,Y)$ is a role formula and $F(Y)$ is in $\mathcal{C}$, then $(\exists Y)(R(X,Y) \wedge F(Y))$ and $(\forall X)(R(X,Y) \rightarrow F(Y))$ are in $\mathcal{C}$ as well.

The set of *concept axioms* consists of all strings of the form $(\forall X)(C(X) \rightarrow F(X))$ or $(\forall X)(C(X) \leftrightarrow F(X))$. A *terminology* or *T-box* is a finite set $\mathcal{K}_T$ of concept axioms such that

concept axioms
terminology
T-box
$\mathcal{K}_T$

1. each atomic concept C occurs at most once as left-hand side of an axiom and
2. the set does not contain any cycles.[1]

The set of *generalized concept axioms* consists of all strings of the form $(\forall X)\,(F(X) \rightarrow G(X))$ or $(\forall X)\,(F(X) \leftrightarrow G(X))\,.$

gerneralized concept axiom

An example of a T-box is shown in Table 1.1. Informally, the concepts *woman* and *man* are not completely defined but a necessary condition is stated, viz. that both are persons. The remaining concepts are completely defined. For example, a *father* is a man who has a child which is a person. By inspection we observe that all axioms are universally closed in a T-box. Hence, the universal quantifiers can be omitted. Likewise, because each concept formula has precisely one free variable, this variable can be omitted as well. Furthermore, the structure of remaining quantified formulas like $(\exists Y)\,(\mathit{child}(X,Y) \wedge \mathit{parent}(Y))$ and $(\forall Y)\,(\mathit{child}(X,Y) \rightarrow \neg \mathit{man}(Y))$ is also quite regular, which allows for further abbreviations like $\exists \mathit{child} : \mathit{parent}$ and $\forall \mathit{child} : \neg \mathit{man}$, respectively. Alltogether, Table 1.1 depicts the simple terminology also in abbreviated form, where the usage of the symbols $\sqsubseteq$, $=$, $\sqcap$ and $\sqcup$ instead of $\rightarrow$, $\leftrightarrow$, $\wedge$ and $\vee$, respectively, is motivated by the following semantics.

The semantics for terminologies is the usual semantics for first order logic formulas. However, the restricted form of concept formulas and concept axioms allows the representation of the semantics in a more convenient and intuitive form. Let I be an interpretation with finite, non-empty domain $\mathcal{D}$.

- I assigns to each constant a an element a^I of $\mathcal{D}$.
- I assigns to each unary predicate symbol C a subset $C^I \subseteq \mathcal{D}$. This subset contains precisely the individuals from $\mathcal{D}$ which belong to C^I.
- Let F^I and G^I be the subsets of $\mathcal{D}$ assigned to the concept formulas $D(X)$ and $E(X)$, respectively. Then, I assigns $\mathcal{D} \setminus F^I$, $F^I \cap G_1$, and $F^I \cup G^I$ to the concept formulas $\neg F(X)$, $F(X) \wedge G(X)$, and $F(X) \vee G(X)$, respectively.
- I assigns to each binary relation symbol symbol R a set $R^I \subseteq \mathcal{D} \times \mathcal{D}$. Let $R^I(d)$ denote the set of all $d' \in \mathcal{D}$ obtained from R^I by selecting all tuples whose first argument is d and projecting this selection onto the second argument, i.e.,

$$R^I(d) = \{d' \in \mathcal{D} \mid (d,d') \in R^I\}.$$

Then, I assigns

$$\{d \in \mathcal{D} \mid R^I(d) \cap F^I \neq \emptyset\}$$

[1] A concept C *depends on* the concept C' wrt the T-box $\mathcal{K}_T$ iff $\mathcal{K}_T$ contains a concept axiom of the form $(\forall X)(C(X) \rightarrow F(X))$ or $(\forall X)(C(X) \leftrightarrow F(X))$ such that C' occurs in F. A T-box is said to be *cyclic* iff it contains a concept which recursively depends on itself.

$$
\begin{aligned}
(\forall X)\,(woman(X) &\rightarrow person(X)),\\
(\forall X)\,(man(X) &\rightarrow person(X)),\\
(\forall X)\,(mother(X) &\leftrightarrow (woman(X) \wedge (\exists Y)\,(child(X,Y) \wedge person(Y)))),\\
(\forall X)\,(father(X) &\leftrightarrow (man(X) \wedge (\exists Y)\,(child(X,Y) \wedge person(Y)))),\\
(\forall X)\,(parent(X) &\leftrightarrow (mother(X) \vee father(X))),\\
(\forall X)\,(grandparent(X) &\leftrightarrow (parent(X) \wedge (\exists Y)\,(child(X,Y) \wedge parent(Y)))),\\
(\forall X)\,(father\_without\_son(X) &\leftrightarrow (father(X) \wedge (\forall Y)\,(child(X,Y) \rightarrow \neg man(Y)))).
\end{aligned}
$$

$$
\begin{aligned}
woman &\sqsubseteq person,\\
man &\sqsubseteq person,\\
mother &= woman \sqcap \exists child : person,\\
father &= man \sqcap \exists child : person,\\
parent &= mother \sqcup father,\\
grandparent &= parent \sqcap \exists child : parent,\\
father\_without\_son &= father \sqcap \forall child : \neg man.
\end{aligned}
$$

Table 1.1: A simple terminology as set of first-order concept axioms (top) and in abbreviated form (bottom).

and

$$\{d \in \mathcal{D} \mid R^I(d) \subseteq F^I\}$$

to the concept formulas

$$(\exists X)\,(R(X,Y) \wedge F(Y))$$

and

$$(\forall X)\,(R(X,Y) \rightarrow F(Y)),$$

respectively.

The meaning of a generalized concept axiom under I is defined as follows, where $F(X)$ and $G(X)$ are concept formulas

$$
\begin{aligned}
I \models (\forall X)\,(F(X) \rightarrow G(X)) \quad &\text{iff} \quad F^I \subseteq G^I.\\
I \models (\forall X)\,(F(X) \leftrightarrow G(X)) \quad &\text{iff} \quad F^I = G^I.
\end{aligned}
$$

I is said to be a *model* for a terminology $\mathcal{K}_T$ iff it satisfies all concept axioms in $\mathcal{K}_T$.

In other words, the semantics of any concept formula is simply a subset of the domain of the interpretation. The meaning of implications and equivalences between concept formulas is the subset and equality relation respectively.

1.2 Assertions

Having specified the terminology, the next step is to model the individuals and the facts known about these individuals along with their relationships and roles. We will call these facts assertions and we need a language for expressing assertions. This language will use the concepts defined in $\mathcal{K}_T$. More formally, let C be a unary relation symbol, R a binary relation symbol, and a as well as b be constants. Then an *assertion* is an expression of

assertion

$$parent(carl),\ parent(conny),$$
$$child(conny, joe),\ child(conny, carl),$$
$$man(joe),\ man(carl),\ woman(conny).$$

Table 1.2: A simple A-box.

the form $F(a)$ or $R(a, b)$. An *A-box* is a finite set of assertions and will be denoted by $\mathcal{K}_A$. Whereas concept formulas provide the terminology for certain aspects of the world, assertions describe the actual state of the world.

A-Box $\mathcal{K}_A$

The semantics of assertions is defined in the usual way. Let I be an interpretation with finite, non-empty domain $\mathcal{D}$ then

$$\begin{aligned} I \models C(a) \quad &\text{iff} \quad a^I \in C^I, \\ I \models R(a, b) \quad &\text{iff} \quad b^I \in R^I(a). \end{aligned}$$

I is said to be a *model* for $\mathcal{K}_A$ iff I satisfies each assertion occurring in $\mathcal{K}_A$. As an example consider the assertions shown in Table 1.2.

There are two basic inferences provided by description logics, viz. subsumption and unsatisfiability testing. All other inferences can be reduced to these two as shown below.

1.3 Subsumption

Let G and F be two concept formulas (in abbreviated form) and $\mathcal{F}_T$ a T-box. G is said to *subsume* F *wrt* $\mathcal{K}_T$ iff $\mathcal{F}_T \models F \sqsubseteq G$. Equivalently, G subsumes F wrt $\mathcal{F}_T$ iff for all models I of $\mathcal{K}_T$ we find that $F^I \subseteq G^I$. For example, let $\mathcal{F}_T$ be the T-box given in Table 1.1, then the concept *person* subsumes both, *man* and *woman*. Similarly, *parent* subsumes *grandparent*. One should observe that the latter subsumption is not explicitly contained in $\mathcal{K}_T$ and has to be computed by comparing the concept.

subsumption

The subsumption relation for the simple description logic presented in this section is decidable [NS90] but intractable[2] [Neb90]. In [LB87] a restricted description logic without negation and disjunction was shown to be tractable.

Several other questions of interest concerning terminologies can be reduced to subsumption. For example, if a knowledge engineer has defined a complex concept based on simpler concepts, he or she should be interested in whether the complex concept is meaningful in the sense that there is at least one object in the real world which belongs to that concept. This can be expressed formally by requiring that a concept is satisfiable by some model of the given T-box $\mathcal{K}_T$, ie. some model of $\mathcal{K}_T$ assigns a non-empty subset of the domain to the concept formula. Alternatively, a concept F is said to be *unsatisfiable* iff $\mathcal{K}_T \models F = \bot$, where $\bot$ denotes an unsatisfiable formula. Unsatisfiability can be reduced to subsumption with the help of the law $F \sqsubseteq G \equiv F \sqcap \neg G = \bot$.

unsatisfiability

Other interesting problems are disjointness and equivalence of concepts:

[2] A problem is said to be *tractable* iff it can be solved in polynomial time wrt the size of the problem. A relation is said to be *tractable* iff the problem of whether a given tuple belongs to the relation is tractable.

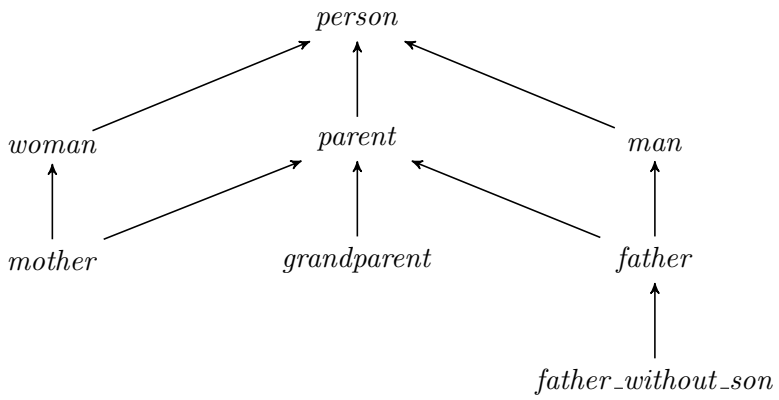


Figure 1.2: The taxonomy defined by the T-box given in Table 1.1, where each arrow from concept F to concept G denotes $F \rhd_T G$.

disjointness

- Two concepts F and G are said to be *disjoint wrt* $\mathcal{K}_T$ iff $\mathcal{K}_T \models F \sqcap G = \bot$.

equivalence

- Two concepts F and G are said to be *equivalent wrt* $\mathcal{K}_T$ iff $\mathcal{K}_T \models F = G$.

Both, disjointness and equivalence can be reduced to subsumption.

Each T-box $\mathcal{K}_T$ represents a taxonomy. In fact, the subsumption relation can be used to compute this taxonomy. Let $\mathcal{C}$ denote the set of concepts and let F as well as G be

$\equiv_T$ elements of $\mathcal{C}$. We define

$$F \equiv_T G \text{ iff } \mathcal{K}_T \models F = G$$

$\sqsubseteq_T$ and

$$F \sqsubseteq_T G \text{ iff } \mathcal{K}_T \models F \sqsubseteq G.$$

By definition $\equiv_T$ is an equivalence relation on $\mathcal{C}$. Consequently, $\mathcal{C}$ can be partitioned into its equivalence classes wrt $\equiv_T$. Let $\mathcal{C}|_{\equiv_T}$ be the quotient of $\mathcal{C}$ under $\equiv_T$. One should observe that $\sqsubseteq_T$ is reflexiv, transitive, and antisymmetric on $\mathcal{C}|_{\approx_T}$, i.e.

$$\begin{array}{ll} F \sqsubseteq_T F, & \text{(reflexivity)} \\ F \sqsubseteq_T G \text{ and } G \sqsubseteq_T H \text{ implies } F \sqsubseteq_T H, & \text{(transitivity)} \\ F \sqsubseteq_T G \text{ and } G \sqsubseteq_T F \text{ implies } F \equiv_T G, & \text{(antisymmetry)} \end{array}$$

$\rhd_T$ where $F, G, H \in \mathcal{C}|_{\approx_T}$. Thus, $\sqsubseteq_T$ is a partial order on $\mathcal{C}|_{\approx_T}$. Let $\rhd_T$ be the unique minimal binary relation on $\mathcal{C}$ such that $\sqsubseteq_T$ is its reflexive and transitive closure. The

taxonomy restriction of $\rhd_T$ to the set of atomic concept formulas is called the *taxonomy defined by* $\mathcal{K}_T$. Figure 1.2 shows the taxonomy defined by the T-box specified in Table 1.1. Such a taxonomy can be computed using a subsumption algorithm.

1.4 Unsatisfiability Testing

Given a T-box and an A-box like the ones depicted in Tables 1.1 and 1.2, respectively, we may want to reason about assertions wrt the given terminology. For example, we may want to know whether Conny is a grandparent, ie.

$$\mathcal{K}_T \cup \mathcal{K}_A \models \mathit{grandparent}(\mathit{conny}),$$

whether Carl is a person, ie.

$$\mathcal{K}_T \cup \mathcal{K}_A \models person(carl),$$

whether Carl is a father without sons, ie.

$$\mathcal{K}_T \cup \mathcal{K}_A \models father\_without\_son(carl),$$

or whether Joe is a child of Conny, ie.

$$\mathcal{K}_T \cup \mathcal{K}_A \models child(conny, joe).$$

To answer these questions, we apply a well-known theorem from classical logic, viz. that $\mathcal{F} \models G$ iff $\mathcal{F} \cup \{\neg G\}$ is unsatisfiable. With an appropriate calculus for testing unsatisfiable we are able to conclude that Conny is a grandparent and Carl is a person, but we cannot conclude Carl is a father without sons or that Joe is a child of Conny.

Other questions can be reduced to unsatisfiability testing as well, for example, the question of whether there are parents:

$$\mathcal{F}_T \cup \mathcal{F}_A \models (\exists X)\, parent(X).$$

Another example is the so-called *realisation problem*: Given a T-box $\mathcal{K}_T$, and A-box $\mathcal{K}_A$, and an individual a, what are the most specific concepts defined in $\mathcal{K}_T$ to which a belongs? In this problem, specificity is defined wrt the subsumption relation, where the concept F is said to be *more specific* than the concept G iff G is subsumed by F. In the example T-box and A-box shown in Tables 1.1 and 1.2, *grandparent* is the most specific concept to which Conny belongs.

realisation problem

1.5 Final Remarks

As we have seen in the examples of the previous section, we were unable to conclude that Carl is a father without sons although the A-box shown in Figure 1.2 does not mention any son of Carl. Description logics specify a so-called *open world*. Additional assertions like

open world

$$man(fritz),\ \ child(carl, fritz)$$

may be added without the need to withdraw previously derived conclusions. In other words, description logics are usually classical logics and are monotonic.

Description logics may be extended to include role restrictions, complex and transitive roles, cyclic concept definitions, or concrete domains like the reals. But sometimes these logics are more restricted like, for example, disallowing universally quantified concept formulas.

The Description Logic Handbook [BCM+03] provides a thorough account of description logics covering all aspects from theory over implementations to applications. A more recent account of developments can be found in [Baa11].

Chapter 2

Equational Logic

The equality relation plays an important role in mathematics, computer science, artificial intelligence, operations research, and many other areas. For example, many mathematical structures like monoids, groups, or rings involve equality. Common data structures like lists, stacks, sets, or multisets can be described with the help of the equality relation. Functional programming is programming with equations. These are just a few applications.

2.1 Equational Systems

In this chapter we consider a first-order language over an alphabet which contains the binary relation symbol $\approx$. Usually, $\approx$ is written infix and called *equality*. An *equation* is an expression of the form $s \approx t$, where s and t are terms. An *equational system* $\mathcal{E}$ is a set of universally closed equations. For example, the equational system given in Table 2.1 specifies a group, where the universal quantifiers are omitted. If equations are negated, then instead of $\neg s \approx t$ we write the more common $s \not\approx t$.

$\approx$
equation
equational system
$\mathcal{E}$

So far, the equality symbol is just an ordinary relation symbol. But usually we expect equality to have the properties reflexivity, symmetry, transitivity and substitutivity. This can be expressed within a first-order logic by the equational system $\mathcal{E}_\approx$ given in Table 2.2, which consists of the so-called *axioms of equality*. One should observe that the substitutivity laws are in fact schemata, which have to be instantiated by every function and relation symbol occurring in the underlying alphabet. One should also note that $\mathcal{E}_\approx$ is not minimal in the sense that axioms may be removed without changing the semantics

$\mathcal{E}_\approx$
axioms of equality

$(X \cdot Y) \cdot Z \approx X \cdot (Y \cdot Z),$	(associativity)
$1 \cdot X \approx X,$	(left unit)
$X \cdot 1 \approx X,$	(right unit)
$X^{-1} \cdot X \approx 1,$	(left inverse)
$X \cdot X^{-1} \approx 1.$	(right inverse)

Table 2.1: An equational system $\mathcal{E}$ specifying a group with binary function symbol $\cdot$ written infix, unary (inverse) function $^{-1}$ written postfix and unit element or constant 1. All equations are assumed to be universally closed.

$X \approx X,$	(reflexivity)
$X \approx Y \to Y \approx X,$	(symmetry)
$X \approx Y \wedge Y \approx Z \to X \approx Z,$	(transitivity)
$\bigwedge_{i=1}^{n} X_i \approx Y_i \to f(X_1, \ldots, X_n) \approx f(Y_1, \ldots, Y_n),$	(f–substitutivity)
$\bigwedge_{i=1}^{n} X_i \approx Y_i \wedge r(X_1, \ldots, X_n) \to r(Y_1, \ldots, Y_n).$	(r–substitutivity)

Table 2.2: The equational system $\mathcal{E}_{\approx}$ specifying the axioms of equality, where the substitutivity axioms are defined for each function symbol f and each relation symbol r in the underlying alphabet.

of $\mathcal{E}_{\approx}$.

As usual we are interested in the logical consequences of an equational system. Formally, let $\mathcal{E}$ be an equational system and F a formula. Then we are interested in the relation

$$\mathcal{E} \cup \mathcal{E}_{\approx} \models F.$$

For example, let $\mathcal{E}$ be the equational systems given in Tables 2.1. Suppose we would like to show that a group which additionally satisfies the equation $X \cdot X \approx 1$ for all X is commutative. This can be expressed as

$$\mathcal{E} \cup \mathcal{E}_{\approx} \cup \{X \cdot X \approx 1\} \models (\forall X, Y)\, X \cdot Y \approx Y \cdot X. \tag{2.1}$$

Sometimes we are also interested in existentially closed equations. For example, let a be a constant, then we may be interested to find a substitution for the variable X such that $X \cdot a \approx 1$, i.e.

$$\mathcal{E} \cup \mathcal{E}_{\approx} \models (\exists X)\, X \cdot a \approx 1.$$

Equational systems are sets of definite formulas and, hence, admit a least (Herbrand) model. For example, suppose that the only function symbols are the constants a, b, and the binary symbol g. Now, consider $\mathcal{E} = \{a \approx b\}$. The least model of $\mathcal{E} \cup \mathcal{E}_{\approx}$ is the set

$$\begin{array}{ll} & \{t \approx t \mid t \text{ is a ground term}\} \\ \cup & \{a \approx b,\ b \approx a\} \\ \cup & \{g(a,a) \approx g(b,a),\ g(a,a) \approx g(a,b),\ g(a,a) \approx g(b,b),\ \ldots\} \end{array}$$

$\approx_{\mathcal{E}}$ We define

$$s \approx_{\mathcal{E}} t \ \text{ iff } \ \mathcal{E} \cup \mathcal{E}_{\approx} \models \forall s \approx t,$$

where s and t are terms and $\forall$ denotes the universal closure. $\approx_{\mathcal{E}}$ is the *least congruence relation on terms generated by* $\mathcal{E}$.

The relation $\approx_{\mathcal{E}}$ is defined semantically and we would like to find syntactic characterizations of this relation in order to mechanize the computation of $\approx_{\mathcal{E}}$. As all formulas occurring in (2.12) are first-order and in clause form we could apply resolution to determine whether commutativity is entailed. If we do so, however, it becomes all too obvious that the single resolution steps are awkward and do not correspond to the way mathematicians would solve such a problem. Moreover the search space is extremely large. In fact, if the search space is traversed in a breadth-first way then 10^{21} deduction steps are needed (see [Bun83]). That this technique is clearly impractical was observed almost as soon as the resolution principle was discovered. The clauses which cause the trouble are mainly the axioms of equality. J. Alan Robinson proposed to remove these and similar

troublesome clauses from the given set of formulas and to build them into the deductive machinery [Rob67].

Where shall we insert the troublesome axioms? Basically there are two possibilities. Either a new inference rule is added to the resolution calculus or the resolution rule itself is modified by building the equational theory into the unification computation. Whereas the latter idea is investigated in Section 2.4, the former possibility is presented in the next section.

2.2 Paramodulation

Paramodulation extends resolution in the case of equality. The most important principle behind equality is that we may replace equals by equals. For example, given any expression over the natural numbers, we may replace $1+1$ by 2 as both terms denote the same object, viz. the natural number 2. This principle can directly be applied to compute the logical consequences of equational systems. The rule of inference capturing this principle is called *paramodulation* and is not restricted to equations but can be applied to general clauses. *paramodulation*

Let $L\lceil s\rceil$ denote a literal L which contains an occurrence of the term s and $L[s/t]$ the literal L where this occurrence has been replaced by t. Let

$$C_1 = [L\lceil s\rceil, L_1, \ldots, L_n]$$

and

$$C_2 = [l \approx r, L_{n+1}, \ldots, L_m]$$

be two clauses, where $0 \leq n \leq m$. If s and l are unifiable with most general unifier θ, then

$$[L\lceil s/r\rceil, L_1 \ldots, L_m]\theta$$

is called *paramodulant of* C_1 *and* C_2. We also say that *paramodulation was applied to* C_1 *using* C_2. The notions of *derivation* and *refutation* defined for the resolution calculus can be straightforwardly extended to paramodulation and resolution. One should observe, that in a derivation the parent clauses of a resolvent must be variable-disjoint. This condition applies to paramodulants as well. In linear derivations–like the ones considered in the sequel of this section–this can be achieved by considering new variants of the input clauses. *paramodulant* *derivation* *refutation*

As equations are first-order expressions we recall that

$$\begin{array}{ll}
& \mathcal{E} \cup \mathcal{E}_{\approx} \models \forall s \approx t \\
\text{iff} & \bigwedge_{C_{\mathcal{E}} \cup \mathcal{E}_{\approx}} \rightarrow \forall s \approx t \text{ is valid} \\
\text{iff} & \neg(\bigwedge_{C_{\mathcal{E}} \cup \mathcal{E}_{\approx}} \rightarrow \forall s \approx t) \text{ is unsatisfiable} \\
\text{iff} & \neg(\neg \bigwedge_{C_{\mathcal{E}} \cup \mathcal{E}_{\approx}} \vee \forall s \approx t) \text{ is unsatisfiable} \\
\text{iff} & \neg\neg \bigwedge_{C_{\mathcal{E}} \cup \mathcal{E}_{\approx}} \wedge \neg \forall s \approx t \text{ is unsatisfiable} \\
\text{iff} & C_{\mathcal{E}} \cup \mathcal{E}_{\approx} \cup \{\exists s \not\approx t\} \text{ is unsatisfiable.}
\end{array}$$

The existential quantifiers can be removed by Skolemization. It can be shown that each paramodulation step can be simulated by resolution steps using the axioms of equality: Intuitively, the substitutivity axioms may be applied to move the term s upon which

1	$[\neg p(g(f(b,a)))]$	(goal)
2	$[f(W,Z) \approx f(Z,W)]$	(commutativity of f)
3	$[\neg p(g(f(a,b)))]$	(par,1,2,$\{W \mapsto b, Z \mapsto a\}$)
4	$[p(g(f(a,b)))]$	(fact)
5	$[\,]$	(res,3,4,ε)

Table 2.3: A proof of (2.2) by resolution and paramodulation, where *par* denotes a paramodulation step followed by the numbers of the parent clauses and the most general unifier used in this step. Likewise, *res* denotes a resolution step.

1	$[\neg p(g(f(b,a)))]$	(goal)
2	$[p(Y),\ \neg p(X),\ X \not\approx Y]$	(r-substitutivity)
3	$[\neg p(X),\ X \not\approx g(f(b,a))]$	(res,1,2,$\{Y \mapsto f(b,a)\}$)
4	$[g(U) \approx g(V),\ U \not\approx V]$	(f-substitutivity)
5	$[\neg p(g(U)),\ U \not\approx f(b,a)]$	(res,3,4,$\{X \mapsto g(U),\ V \mapsto f(b,a)\}$)
6	$[f(W,Z) \approx f(Z,W)]$	(commutativity of f)
7	$[\neg p(g(f(a,b)))]$	(res,5,6,$\{U \mapsto f(a,b),\ Z \mapsto b,\ W \mapsto a\}$)
8	$[p(g(f(a,b)))]$	(fact)
9	$[\,]$	(res,7,8,ε)

Table 2.4: A proof of (2.2) by resolution using the substitutivity axioms.

paramodulation was applied to the top level such that it can be unified with the term l. The following example shall illustrate this intuition. Suppose, we want to show that

$$\{p(g(f(a,b)))\} \cup \{f(X,Y) \approx f(Y,X)\} \cup \mathcal{E}_{\approx} \models p(g(f(b,a))) \tag{2.2}$$

Table 2.3 shows a proof by resolution and paramodulation, whereas Table 2.4 shows a corresponding proof by resolution using the substitutivity axioms. Formally, Brand has proven in [Bra75] that resolution, factoring, and paramodulation are sound and complete if the axiom of reflexivity is added.

Theorem 2.1 *$\mathcal{E} \cup \mathcal{E}_{\approx} \cup \{\exists s \not\approx t\}$ is unsatisfiable if and only if there is a refutation of $\mathcal{E} \cup \{X \approx X,\ \exists s \not\approx t\}$ with respect to paramodulation, resolution and factoring.*

In other words, all equational axioms except the axiom of reflexivity are built into paramodulation.[1] We can now apply this theorem to show that (2.12) holds. In particular, (2.12) holds iff it can be shown that

$$\bigwedge_{\mathcal{E} \cup \mathcal{E}_{\approx} \cup \{X \cdot X \approx 1\}} \rightarrow (\forall X, Y)\, X \cdot Y \approx Y \cdot X$$

is valid iff

$$(\mathcal{E} \cup \mathcal{E}_{\approx} \cup \{X \cdot X \approx 1\}) \cup \{\exists X, Y)\, X \cdot Y \not\approx Y \cdot X\} \tag{2.3}$$

is unsatisfiable. Skolemizing (2.3) we obtain

$$\mathcal{E} \cup \mathcal{E}_{\approx} \cup \{X \cdot X \approx 1\} \cup \{a \cdot b \not\approx b \cdot a\}, \tag{2.4}$$

[1] One should observe that, strictly speaking, the clauses occurring in $\mathcal{E}$ are not axioms with respect to the resolution and paramodulation calculus. The only axiom in this calculus is the empty clause $[\,]$.

1	$a \cdot b \not\approx \underline{b} \cdot a$	(initial query)
2	$1 \cdot X_1 \approx X_1$	(left unit)
3	$\underline{X_2} \approx X_2$	(reflexivity)
4	$X_1 \approx 1 \cdot X_1$	(par,2,3,$\{X_2 \mapsto 1 \cdot X_1\}$)
5	$a \cdot b \not\approx (\underline{1} \cdot b) \cdot a$	(par,1,4,$\{X_1 \mapsto b\}$)
6	$X_3 \cdot X_3 \approx 1$	(hypothesis)
7	$\underline{X_4} \approx X_4$	(reflexivity)
8	$1 \approx X_3 \cdot X_3$	(par,6,7,$\{X_4 \mapsto X_3 \cdot X_3\}$)
9	$a \cdot b \not\approx ((X_3 \cdot X_3) \cdot b) \cdot \underline{a}$	(par,5,8,ε)
	$\vdots$	(right unit)
	$a \cdot b \not\approx ((X_3 \cdot X_3) \cdot b) \cdot (a \cdot \underline{1})$	
	$\vdots$	(hypothesis)
	$a \cdot b \not\approx \underline{((X_3 \cdot X_3) \cdot b) \cdot (a \cdot (X_4 \cdot X_4))}$	
	$\vdots$	(associativity)
	$a \cdot b \not\approx (X_3 \cdot \underline{((X_3 \cdot b) \cdot (a \cdot X_4))}) \cdot X_4$	
	$\vdots$	(hypothesis)
	$a \cdot b \not\approx \underline{(a \cdot 1)} \cdot b$	
	$\vdots$	(right unit)
n	$a \cdot b \not\approx a \cdot b$	
n'	$X_5 \approx X_5$	(reflexivity)
n''	$[\,]$	(res,n,n',$\{X_5 \mapsto a \cdot b\}$)

Table 2.5: Fragment of a refutation using paramodulation and resolution to show that groups satisfying the law $(\forall X)\, X \cdot X \approx 1$ are commutative. The subterm whereupon paramodulation is applied is underlined. One should observe that steps 2 to 4 show how symmetry is captured by paramodulation. In the application of paramodulation upon the subterm $((X_3 \cdot b) \cdot (a \cdot X_4))$ using a new variant $Z \cdot Z \approx 1$ of the hypotheses the most general unifier is $\{Z \mapsto a \cdot b,\ X_3 \mapsto a,\ X_4 \mapsto b\}$.

where a and b are new Skolem constants. We can now apply Theorem 2.1 and obtain the refutation shown in Table 2.5. The refutation still looks clumsy but Table 2.6 shows a shorthand notation which can always be used if only equation are involved and which is very close to the way mathematicians transform expressions using equalities. One should observe that mathemeticians prove universal statement like $(\forall X, Y)\, X \cdot Y \approx Y \cdot X$ usually by selecting arbitrary but fixed elements a and b replacing X and Y, respectively, and showing that $a \cdot b \approx b \cdot a$. Arbitrary but fixed elements correspond precisely to the Skolem constants introduced in the process of turning a formula into clause form.

The search space which has to be investigated by a simple breadth-first search procedure based on resolution, factoring, and paramodulation is still huge. In the example, it consists of about 10^{11} nodes. Many steps are redundant and useless. For example, an equation may be used from left to right, replacing an instance of the left subterm by the instance of the right one, and some steps later, the equation may be used the other way around, replacing an instance of the right subterm by the instance of the left one. If we could somehow restrict the use of these equations so that they are used in one direction only, then many useless steps could be avoided. This idea has led to term rewriting systems. On the other hand, if we restrict the use of equations, then we should be prepared to pay a price in that the expressive power of the restricted system is less than the expressive

$$
\begin{array}{lcll}
\underline{b} \cdot a & \approx & (\underline{1} \cdot b) \cdot a & \text{(left unit)} \\
& \approx & ((X_3 \cdot X_3) \cdot b) \cdot \underline{a} & \text{(hypothesis)} \\
& \approx & ((X_3 \cdot X_3) \cdot b) \cdot (a \cdot \underline{1}) & \text{(right unit)} \\
& \approx & \underline{((X_3 \cdot X_3) \cdot b) \cdot (a \cdot (X_4 \cdot X_4))} & \text{(hypothesis)} \\
& \approx & \overline{(X_3 \cdot \underline{((X_3 \cdot b) \cdot (a \cdot X_4))}) \cdot X_4} & \text{(associativity)} \\
& \approx & \underline{(a \cdot 1)} \cdot b & \text{(hypothesis)} \\
& \approx & \overline{a \cdot b} & \text{(right unit)}
\end{array}
$$

Table 2.6: Shorthand notation for the refutation shown in Table 2.5.

$$
\begin{array}{lcl}
\mathit{append}([\,], X) & \to & X, \\
\mathit{append}([X|Y], Z) & \to & [X|\mathit{append}(Y, Z)], \\
\mathit{reverse}([\,]) & \to & [\,], \\
\mathit{reverse}([X|Y]) & \to & \mathit{append}(\mathit{reverse}(Y), [X]).
\end{array}
$$

Table 2.7: A term rewriting system for the functions *append* and *reverse*.

power of equational systems.

2.3 Term Rewriting Systems

term rewriting system

The idea of term rewriting systems is to orient equations $s \approx t$ into so-called *rewrite rules* $s \to t$ indicating that instances of s may be replaced by instances of t but not vice versa. A *term rewriting system* is a finite set of rewrite rules. As an example consider the term rewriting system shown in Table 2.7, in which the functions *append* and *reverse* are defined. Informally, *append* concatenates two lists and *reverse* reverses a list. Lists are represented using a binary function symbol : and the constant $[\,]$. $[\,]$ denotes the empty list. If Y is a list and X a term then $:(X, Y)$ denotes a list whose head is X and whose tail is Y. To ease the notation it is common to abbreviate lists as follows: $[X|Y]$ is an abbreviation for $:(X, Y)$, where X is a term and Y is a list; furthermore, $[a_1, a_2, \ldots, a_n]$ is an abbreviation for $:(a_1, :(a_2 \ldots :(a_n, [\,]) \ldots))$.

The study of term rewriting systems is concerned with how to orient equations into rewrite rules and what conditions guarantee that term rewriting systems have the same computational power as the equational system they were derived from. Moreover, term rewriting systems can be regarded as the logical basis for a restricted class of functional programs as will be demonstrated later in this section.

rewriting $\to_{\mathcal{R}}$ $\overset{*}{\to}_{\mathcal{R}}$

What are term rewriting systems good for? Of course, they shall be used to replace equals by equals. Let $\mathcal{R}$ be a term rewriting system. Let $s\lceil u \rceil$ denote a term s which contains an occurrence of the (sub-)term u and $s\lceil u/v \rceil$ the term s where this occurrence has been replaced by v.[2] A term $s\lceil u \rceil$ *rewrites* to a term t, in symbols $s \to_{\mathcal{R}} t$, iff there exists a rewrite rule $l \to r \in \mathcal{R}$ and a substitution θ such that $u = l\theta$ and $t = s\lceil u/r\theta \rceil$. Let $\overset{*}{\to}_{\mathcal{R}}$ be the reflexive and transitive closure of $\to_{\mathcal{R}}$. Thus, $s \overset{*}{\to}_{\mathcal{R}} t$ iff there is a sequence $u_1, \ldots, u_n$ of terms such that $s = u_1$, $u_i \to_{\mathcal{R}} u_{i+1}$, for all $1 \leq i < n$, and

[2] One should note that only one occurrence of u in s is replaced even if u occurs several times in s.

$u_n = t$. Furthermore, $s \leftrightarrow_{\mathcal{R}} t$ iff $s \rightarrow_{\mathcal{R}} t$ or $s \leftarrow_{\mathcal{R}} t$. $\overset{*}{\leftrightarrow}_{\mathcal{R}}$ is the reflexive and transitive closure of $\leftrightarrow_{\mathcal{R}}$. For ease of notation we sometimes omit the subscript $\mathcal{R}$ if it is obvious from the context which term rewriting system is meant. Recalling the example shown in Table 2.7 we find that:

$\leftrightarrow_{\mathcal{R}}$
$\overset{*}{\leftrightarrow}_{\mathcal{R}}$
!

$$\begin{array}{ll} & \underline{append([1,2],[3,4])} \\ \rightarrow & [1 \mid \underline{append([2],[3,4])}] \\ \rightarrow & [1,2 \mid \underline{append([\,],[3,4])}] \\ \rightarrow & [1,2,3,4], \end{array} \tag{2.5}$$

where the rewritten (sub-)terms are underlined.

The substitution θ used in a rewriting step is only applied to the rewrite rule used in a rewriting step, but not to the term which is rewritten. Given two terms u and l, the problem of whether there exists a substitution θ such that $u = l\theta$ is called a *matching problem*, and if such a substitution exists, then θ is called a *matcher for l against u*. Matching is a restricted form of unification and all notions and notations concerning unification hold for matching problems as well. In particular, if there exists a matcher θ such that $u = l\theta$ then there exists also a most general one and it suffices to consider such a most general matcher in computing the rewrite relation $\rightarrow_{\mathcal{R}}$.

matching
matcher

In the literature term rewriting systems are often defined such that for all rules $l \rightarrow r$ occurring in $\mathcal{R}$ it is required that $var(l) \supseteq var(r)$, where $var(t)$ denotes the set of variables occurring in t. As an immediate consequence of such a condition we obtain that if $s \rightarrow_{\mathcal{R}} t$ then $var(s) \supseteq var(t)$. This can be examplified by recalling the term rewriting system shown in Table 2.7 and considering the term $append([V],[W])$, where V and W are variables:

var

$$\underline{append([V],[W])} \rightarrow [V|\underline{append([\,],[W])}] \rightarrow [V,W]$$

and we find that

$$var(append([V],[W])) = \{V,W\} = var([V, append([\,],W)) = var([V,W]).$$

As another example consider the term rewriting system

$$\mathcal{R} = \{projection1\,(X,Y) \rightarrow X\}.$$

It specifies a function *projection1* which projects onto its first argument. Here,

$$projection1\,(f(V),W) \rightarrow f(V)$$

and we find that

$$var(projection1\,(f(V),W)) = \{V,W\} \supset \{V\} = var(f(V)).$$

Let $\mathcal{E}_{\mathcal{R}}$ be the equational system obtained from the rewriting system $\mathcal{R}$ by replacing each rule $l \rightarrow r \in \mathcal{R}$ by the equation $l \approx r$ and adding the axioms of equality. It is not too difficult to see that

$\mathcal{E}_{\mathcal{R}}$

$$\text{if } s \rightarrow_{\mathcal{R}} t \text{ then } s \approx_{\mathcal{E}_{\mathcal{R}}} t.$$

In other words, if s rewrites to t then in each model of $\mathcal{E}_{\mathcal{R}}$ and, in particular, in the least model of $\mathcal{E}_{\mathcal{R}}$ the terms s and t denote the same element of the domain. In fact, an even stronger result can be shown, viz.

$$s \approx_{\mathcal{E}_{\mathcal{R}}} t \text{ iff } s \overset{*}{\leftrightarrow}_{\mathcal{R}} t. \tag{2.6}$$

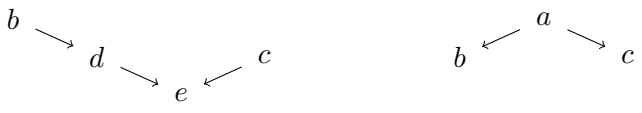


Figure 2.1: Two rewriting derivations for $b \overset{*}{\leftrightarrow} c$. The one of the left-hand side is in valley form.

This gives another syntactic characterization of logical consequence: In order to show that two terms s and t are equal under $\mathcal{E}_\mathcal{R}$, we have to find a derivation from s to t wrt $\leftrightarrow$. As an example consider the term rewriting system

$$\mathcal{R} = \{a \to b,\ a \to c,\ b \to d,\ c \to e,\ d \to e\}.$$

Then $b \approx_{\mathcal{E}_\mathcal{R}} c$ because

$$b \to d \to e \leftarrow c$$

or, alternatively,

$$b \leftarrow a \to c.$$

Such derivations are often depicted graphically as shown in Figure 2.1. The derivation on the left is in so-called *valley form*, whereas this is not the case for the derivation shown on the right. A derivation in valley-form is desirable because in such a derivation rewriting has been applied only to the terms b and c and their successors.

valley form

Unfortunately, the latter characterization of logical consequence is still unsatisfactory because in order to determine whether $s \approx_{\mathcal{E}_\mathcal{R}} t$ we cannot simply apply rewriting to s and t (and their successors). Can we find conditions such that rewriting applied to s and t is complete?

A term s is said to be *reducible* with respect to $\mathcal{R}$ iff there exists a term t such that $s \to_\mathcal{R} t$, otherwise it is said to be *irreducible*. If $s \overset{*}{\to}_\mathcal{R} t$ and t is irreducible, then t is a *normal form* of s. We also say that t is obtained from s by *normalization*. For example, in (2.13) the term $[1,2,3,4]$ is irreducible and, thus, it is the normal form of $\mathit{append}([1,2],[3,4])$.

reducible
irreducible
normal form

One should also observe that the term rewriting system $\mathcal{R}$ shown in Table 2.7 is in fact a functional program defining the functions *append* and *reverse*. In this view, (2.13) is an evaluation of the function *append* called with the arguments $[1,2]$ and $[3,4]$, and the normal form $[1,2,3,4]$ is the value of this function call. Equivalently, this evaluation of the function append can be seen as the desired answer to the question of whether $\mathcal{E}_\mathcal{R} \models (\exists X)\ \mathit{append}([1,2],[3,4]) \approx X$ holds. From a logic programming point of view, the answer substitution

$$\sigma = \{X \mapsto \mathit{append}([1,2],[3,4])\}$$

is also correct, but in most cases it is not the intended one. This is $\{X \mapsto [1,2,3,4]\}$, which can be obtained from σ by normalizing the terms occurring in the codomain of σ with respect to $\mathcal{R}$.

Rewrite rules of the form $X \to r$ can be used to rewrite each subterm. Semantically such a rule specifies that each term is equal to r and therefore the whole domain of any interpretation satisfying this rule effectively collapses to a singleton set. Because such systems are not very interesting, one often disallows such rules in term rewriting systems.

$$\begin{array}{lcl} not(not(X)) & \to & X, \\ not(or(X,Y)) & \to & and(not(X), not(Y)), \\ not(and(X,Y)) & \to & or(not(X), not(Y)), \\ and(X, or(Y,Z)) & \to & or(and(X,Y), and(X,Z)), \\ and(or(X,Y), Z) & \to & or(and(Y,Z), and(Z,X)). \end{array}$$

Table 2.8: A non-confluent but terminating term rewriting system for propositional logic.

In each step of (2.13) there was only one way to rewrite the term. Unfortunately, this is not always the case. As another example, consider the term rewriting system shown in Table 2.8 which can be applied to convert propositional logic expressions into normal form. Here, the term

$$and(or(X,Y), or(U,V))$$

has two normal forms, viz.

$$or(or(and(X,U), and(Y,U)), or(and(X,V), and(Y,V)))$$

and

$$or(or(and(Y,U), and(Y,V)), or(and(V,X), and(X,U))).$$

Recall that our goal was to find restrictions such that the question whether two terms s and t are equal under a given equational theory can be decided by using the equations only from left to right. To this end we need to introduce two more notions, viz. the notion of a confluent and terminating term rewriting system.

For terms s and t we write $s \downarrow_{\mathcal{R}} t$ iff there exists a term u such that $s \stackrel{*}{\to}_{\mathcal{R}} u \stackrel{*}{\leftarrow}_{\mathcal{R}} t$. We write $s \uparrow_{\mathcal{R}} t$ iff there exists a term u such that $s \stackrel{*}{\leftarrow}_{\mathcal{R}} u \stackrel{*}{\to}_{\mathcal{R}} t$. As before, we will omit the index $\mathcal{R}$ if $\mathcal{R}$ can be determined from the context. Returning to Figure 2.1 we find that $b \downarrow c$ and $b \uparrow c$ because of the derivations shown on the left and the right, respectively. $\downarrow_{\mathcal{R}}$ $\uparrow_{\mathcal{R}}$

A term rewriting system $\mathcal{R}$ is said to be *confluent* iff for all terms s and t we find $s \uparrow t$ implies $s \downarrow t$. It is said to be *ground confluent* if it is confluent for ground terms. In other words, if a term rewriting system is confluent, then any two different rewritings originating from a term will eventually converge. *confluent*

A term rewriting system $\mathcal{R}$ has the *Church-Rosser property* iff for all terms s and t, we find $s \stackrel{*}{\leftrightarrow} t$ iff $s \downarrow t$. It can be shown that $\mathcal{R}$ has the Church-Rosser property iff $\mathcal{R}$ is confluent. Combining this result with (2.6) we learn that rewriting need only be applied in one direction if the term rewriting system is confluent. In this case $s \approx_{\mathcal{E}_{\mathcal{R}}} t$ holds iff we find a term u such that both, s and t, rewrite to u. *Church-Rosser*

A term rewriting system $\mathcal{R}$ is *terminating* iff it admits no infinite rewriting sequences. In other words, each rewriting process applied to a term will eventually stop. For example, the term rewriting systems shown in the Tables 2.7 and 2.8 are terminating. Unfortunately, it is undecidable whether a term rewriting system is terminating. However, if the system is terminating then confluence is decidable. *terminating*

Terminating and confluent term rewriting systems are said to be *canonical* or *convergent*. The question of whether two terms s and t are equal under an equational system $\mathcal{E}$ can be decided if we find a canonical term rewriting system $\mathcal{R}$ such that the finest congruence *canonical* *convergent*

relations generated by $\mathcal{E}$ and $\mathcal{E}_{\mathcal{R}}$ coincide. In this case $s \approx_{\mathcal{E}} t$ iff $s \downarrow t$. In other words, for a canonical term rewriting system $\mathcal{R}$ the corresponding equational theory $\mathcal{E}_{\mathcal{R}}$ is decidable. In this case, all we have to do in order to decide whether

$$s \approx_{\mathcal{E}_{\mathcal{R}}} t \tag{2.7}$$

is to normalize both terms s and t. If their normal forms are syntactically equal, then (2.7) holds, otherwise it does not.

Thus, it is desirable that a given term rewriting system is both, terminating and confluent. In the following two sections techniques for showing that a term rewriting system has these properties will be discussed.

2.3.1 Termination

We now consider the question of how to determine whether a given term rewriting system is terminating. The problem is undecidable as shown by [HL78]. Hence, we cannot expect to find an algorithm which proves termination even if the term rewriting system is terminating. All what we can hope for is to develop techniques such that for large classes of term rewriting systems these techniques help to find out whether a system is terminating. These techniques are not confined to term rewriting systems but can be applied to programs in general.

Let $\succeq$ be a partial order on terms, i.e. $\succeq$ is reflexive, transitive, and antisymmetric. Let $\succ$ be defined on terms as follows:

$$s \succ t \text{ iff } s \succeq t \text{ and } s \neq t.$$

well-founded ordering $\succ$ is said to be *well-founded* iff there is no infinite descending sequence $s_1 \succ s_2 \succ \ldots$. All techniques presented in this section make use of a well-founded order $\succ$ on terms having the property that

$$s \to t \text{ implies } s \succ t.$$

termination ordering Formally, a *termination ordering* $\succ$ is a well-founded, transitive, and antisymmetric relation on the set of terms satisfying the following properties:

full invariance property
1. *Full invariance property*: If $s \succ t$ then $s\theta \succ t\theta$ for all substitutions θ.

replacement property
2. *Replacement property*: if $s \succ t$ then $u\lceil s \rceil \succ u\lceil s/t \rceil$ for all terms u containing s.

One should observe that if $s \succ t$ and $\succ$ is a termination ordering then all variables occurring in t must also occur in s.

Theorem 2.2 *Let $\mathcal{R}$ be a term rewriting system and $\succ$ a termination ordering. If for all rules $l \to r \in \mathcal{R}$ we find that $l \succ r$ then $\mathcal{R}$ is terminating.*

Thus, one way to show that a term rewriting system is terminating is to find a termination ordering for this system. One of the simplest termination ordering is based on the *term size* *size* of a term. Let $|s|$ denote the size of a term s, viz. the length of the string s. We can define a termination ordering $\succ$ as follows: $s \succ t$ iff for all grounding substitutions θ we find that

$$|s\theta| > |t\theta|.$$

With the help of such an ordering we find, for example, that

$$f(X,Y) \succ g(X),$$

but there is no such ordering such that

$$f(X,Y) \succ g(X,X).$$

The latter observation limits the applicability of such an ordering and more complex termination orderings have been considered in the literature.

The just mentioned ordering based on the size of the term can be modified by weighting the symbols so that $|s|$ is the weighted sum of the number of occurrences of the symbols. Another class of termination orderings are so-called *polynomial orderings*: Each function symbol is interpreted as a polynomial with coefficients taken from the set of natural numbers. The domain of such an interpretation is the set of polynomials and each variable assignment assigns each variable to itsself. Thus, each term is interpreted as a polynomial on natural numbers. For example, we could define an interpretation I such that

polynomial ordering

$$[f(X,Y)]^{I,\mathcal{Z}} = 2X + Y$$

and

$$[g(X,Y)]^{I,\mathcal{Z}} = X + Y,$$

where the variable assignment $\mathcal{Z}$ is the identity. In this case the ordering

$$s \succ t \text{ iff } s^{I,\mathcal{Z}} > t^{I,\mathcal{Z}}$$

is a termination ordering, where $>$ is the greater-than ordering on natural numbers.

There are other widely used orderings such as the *recursive path ordering* or the *lexicographic path ordering* (see e.g. [Pla93]). But it would be beyond the scope of this introduction to mention all of them. These orderings are often combined with a variety of other methods to determine termination of term rewriting systems. For example, in [FGM+07] SAT-solvers are applied for termination analysis with polynomial interpretations.

This subsection will close with a brief discussion of *incrementality*. An ordering $\succ'$ is *more powerful than* (or *extends*) $\succ$ iff $s \succ t$ implies $s \succ' t$, but not vice versa. This issue will be important in the next subsection. There, we will see that sometimes terminating non-confluent term rewriting systems can be turned into a confluent ones by adding additional rewrite rules. These rules, however, need not comply with the termination ordering used to show that the given term rewriting system is terminating. However, if the incremental property holds, then the termination ordering can be gradually extended with each new rule that is added to a term rewriting system.

incrementality
more powerful than

2.3.2 Confluence

As already mentioned if a term rewriting system is terminating, then confluence is decidable. In this section, an algorithm for deciding confluency is developed.

Following the definition of confluency, we have to consider all terms s and t for which $s \uparrow t$ holds. This can be reformulated as to consider all terms u, s and t such that

u rewrites to s and to t. Fortunately, in case of a terminating term rewriting system we do not have to consider arbitrary long rewriting sequences. Rather, we may restrict our attention to single step rewritings from u to s and t.

locally confluent A term rewriting system is said to be *locally confluent* iff for all terms u, s and t the following holds: If $u \to s$ and $u \to t$ then $s \downarrow t$. The following result was establish by Newman in [New42]:

Theorem 2.3 *Let $\mathcal{R}$ be a terminating term rewriting system. $\mathcal{R}$ is confluent iff $\mathcal{R}$ is locally confluent.*

This result is still insufficient to decide confluency as we have to consider all terms u, and there are infinitely many. Wouldn't it be nice if we could focus on the term rewriting system itself or, more precisely, on the left-hand sides of the rules occurring in the term rewriting system as there are only finitely many? In order to answer this question let us study cases where a term u rewrites to two different terms. How can this happen? Let *redex* $\mathcal{R}$ be a term rewriting system and u a term. A subterm w of u is called a *redex* if w is an instance of the left-hand-side of a rule $l \to r \in \mathcal{R}$, i.e., if there exists a substitution θ such that $w = l\theta$. Now let $l_1 \to r_1$ and $l_2 \to r_2$ be two rules occurring in $\mathcal{R}$ which are both applicable to the term u, i.e., we find two redeces in t corresponding to the left-hand sides of the two applicable rules. In general there are exactly three possibilities of rewriting u in two different ways:

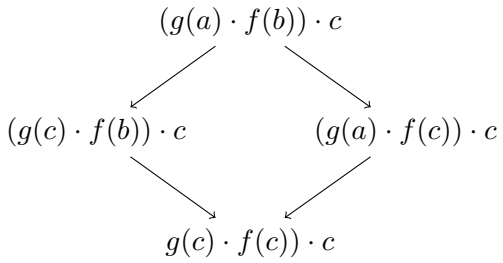
1. The two redeces are disjoint.
2. One redex is a subterm of the other one and corresponds to a variable position in the left-hand side of the other rule.
3. One redex is a subterm of the other one but does not correspond to a variable *overlap* position in the left-hand side of the other rule. In this case the redeces are said to *overlap*.

Examples may help to better understand the three cases. Let u be the term

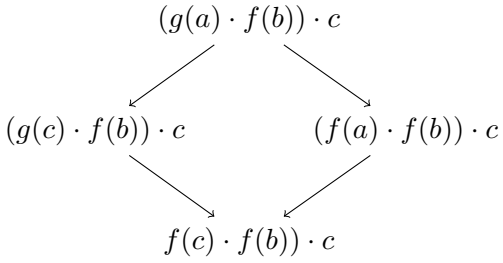
$$(g(a) \cdot f(b)) \cdot c,$$

where $\cdot$ is a binary function symbol written infix, f and g are unary function symbols, and a, b, and c are constants.

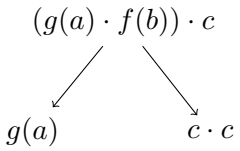
1. Let
$$\mathcal{R} = \{a \to c,\ b \to c\}.$$
Then u contains two redeces, viz. a and b. These redeces are disjoint. In this case it does not matter which rule we apply first because we can always apply the other rule afterwards. After applying both rules we will always end up with the term
$$(g(c) \cdot f(c)) \cdot c.$$
Alltogether, we obtain the following commuting diagram:



2. Let
$$\mathcal{R} = \{a \to c,\ g(X) \to f(X)\}.$$
In this case u contains the redeces a and $g(a)$. Moreover, a corresponds to the variable position in $g(X)$. As in the first case it does not matter which rule is applied first. In any case the rewritings commute to
$$(f(c) \cdot f(b)) \cdot c.$$
Alltogether, the following commuting diagram is obtained:



3. Let
$$\mathcal{R} = \{(X \cdot Y) \cdot Z \to X,\ g(a) \cdot f(b) \to c\}. \tag{2.8}$$
In this case u contains the redeces
$$(g(a) \cdot f(b)) \cdot c, \tag{2.9}$$
i.e., u itself is a redex, and
$$g(a) \cdot f(b). \tag{2.10}$$
Applying the first rule of $\mathcal{R}$ to t at redex (2.9) yields
$$g(a),$$
whereas the application of the second rule of $\mathcal{R}$ at redex (2.10) yields
$$c \cdot c.$$
Both terms are in normal form and they are different. One should observe that redex (2.10) does not correspond to a variable position in the left-hand side of the first rule in $\mathcal{R}$. Alltogether we obtain the following non-commuting diagram:



These examples illustrate that the interesting case for determining whether a term rewriting system is locally confluent is last one and we have to discuss it further. Let us abstract from the example: Suppoese the term rewriting system $\mathcal{R}$ contains the rules $l_1 \to r_1$ and $l_2 \to r_2$ without common variables. Suppose l_2 is unifiable with a non-variable subterm u of l_1 using the most general unifier θ. Then the pair

$$\langle (l_1\lceil u/r_2 \rceil)\theta,\ r_1\theta \rangle$$

critical pair is said to be *critical*.[3] It is obtained by *superposing* l_1 and l_2.

superposition

Recalling the previous example we see that the rules

$$(X \cdot Y) \cdot Z \to X$$

and

$$g(a) \cdot f(b) \to c$$

form a critical pair: The left-hand side of the second rule is unifiable with the subterm

$$(X \cdot Y)$$

of the left-hand side of the first rule using the most general unifier

$$\{X \mapsto g(a),\ Y \mapsto f(b)\}.$$

Thus, we obtain the critical pair

$$\langle c \cdot Z, g(a) \rangle. \tag{2.11}$$

The analysis has shown that in order to decide whether a term rewriting system is locally confluent we have to look at all critical pairs. In fact, it is now easy to see that the following holds:

Theorem 2.4 *A term rewriting system $\mathcal{R}$ is locally confluent iff for all critical pairs $\langle s, t \rangle$ of $\mathcal{R}$ we find that $s \downarrow t$.*

One should observe that in a finite term rewriting system, i.e., a system with finitely many rewrite rules, there may be only finitely many critical pairs and these pairs can be computed in polynomial time. Furthermore, if the term rewriting system is additionally terminating, then all normal forms of each element of a critical pair can be computed in finite time. Hence, we find that the problem of determining whether a given terminating term rewriting system is (locally) confluent is decidable.

Returning to the previous example we find that the elements of the critical pair (2.11) are already in normal form with respect to the term rewriting system $\mathcal{R}$ shown in (2.8). Because these normal forms are different, this system is not (locally) confluent. However, in many cases a terminating and non-confluent term rewriting system can be turned into a confluent one by a so-called completion procedure.

[3] One should observe that if the two rules are variants, and u is equal to l_1 then the critical pair contains identical elements. This is a so-called *trivial* critical pair and need not be considered for obvious reasons.

Given a term rewriting system $\mathcal{R}$ together with a termination ordering $\succ$:

1. If for all critical pairs $\langle s, t\rangle$ of $\mathcal{R}$ we find that $s \downarrow t$ then return "success"; $\mathcal{R}$ is a canonical term rewriting system.
2. If $\mathcal{R}$ has a critical pair whose elements do not rewrite to a common term then transform the elements of the critical pair to some normal form. Let $\langle s, t\rangle$ be the normalized critical pair:
   (a) If $s \succ t$ then add the rule $s \to t$ to $\mathcal{R}$ and goto 1.
   (b) If $t \succ s$ then add the rule $t \to s$ to $\mathcal{R}$ and goto 1.
   (c) If neither $s \succ t$ nor $t \succ s$ then return "fail".

Table 2.9: The completion procedure.

2.3.3 Completion

The question considered in this subsection is whether a terminating term rewriting system $\mathcal{R}$ which is not confluent can be turned into a confluent one. As we will see in a moment this is possible in some cases by adding new rules to the given term rewriting system. Of course, we should require that the added rules do not change the equational theory defined by $\mathcal{R}$. We call two term rewriting systems equivalent if they have the same set of logical consequences. More formally, the term rewriting systems $\mathcal{R}$ and $\mathcal{R}'$ are said to be *equivalent* iff $\approx_{\mathcal{E}_\mathcal{R}} = \approx_{\mathcal{E}_{\mathcal{R}'}}$. *equivalence*

The *completion* procedure is a transformation which adds rules to a terminating term rewriting system while preserving termination and gaining confluence. The idea is that if $\langle s, t\rangle$ is a critical pair, then the rules $s \to t$ or $t \to s$ can be added without changing the equational theory. With such a rule the terms s and t rewrite to a common term. If a procedure adds enough such rules while preserving termination, then it yields a canonical term rewriting system. This idea goes back to Knuth and Bendix [KB70] and can also be found in [Buc87]. *completion*

Such a completion procedure has to cope with several cases.

- The added rules have to preserve termination. Hence, if the elements of a critical pair cannot be oriented into a rule preserving termination, then the completion procedure is said to *fail*. *failure*
- The added rules may lead to new critical pairs, which must be considered. This process may go on forever, in which case the completion procedure is said to *loop*. *loop*

The completion procedure itself is specified in Table 2.9. It can be modified such that it turns a given equational system into a canonical term rewriting system.

A very simple example taken from [Pla93] will illustrate the completion procedure. Consider the term rewriting system

$$\mathcal{R} = \{c \to b,\ f \to b,\ f \to a,\ e \to a,\ e \to d\}$$

and the alphabetic ordering, i.e.

$$f \succ e \succ d \succ c \succ b \succ a.$$

$\mathcal{R}$ is terminating but not confluent because the elements of the critical pairs

$$\langle b, a \rangle \tag{2.12}$$

(obtained by superposing the rules $f \to b$ and $f \to a$) and

$$\langle d, a \rangle$$

(obtained by superposing the rules $e \to a$ and $e \to d$) are already in normal form. Both critical pairs can be oriented with respect to $\succ$ into the rules

$$b \to a \tag{2.13}$$

and

$$d \to a, \tag{2.14}$$

respectively. We obtain the term rewriting system

$$\mathcal{R}' = \{c \to b,\ f \to b,\ f \to a,\ e \to a,\ e \to d,\ b \to a,\ d \to a\}$$

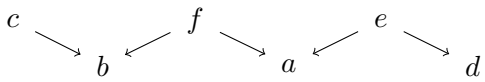
which is canonical because now every term rewrites to a. One should observe that

$$s \approx_{\mathcal{E}_{\mathcal{R}}} t \ = \ s \approx_{\mathcal{E}_{\mathcal{R}'}} t.$$

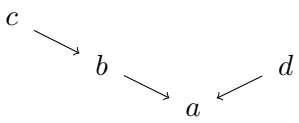
To understand the completion procedure we consider its effects on the rewrite proof of

$$c \approx_{\mathcal{E}_{\mathcal{R}}} d.$$

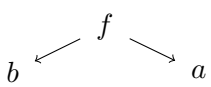
Given $\mathcal{R}$ this proof is:



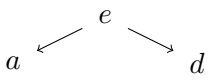
However, with $\mathcal{R}'$ the shorter proof



is obtained. The critical pair (2.12) covers the part



of the original sequence which is replaced by (2.13). Likewise, the critical pair (2.13) covers the part



of the original sequence which is replaced by (2.14). One should observe that the final proof is in valley form.

Various extensions of the completion procedure have been developed to overcome its limitations. An excellent overview is given in [Pla93]. [BN98] is an excellent textbook on term rewriting systems and other reduction systems. Good German introductions to the field can be found in [Ave95] and [Bün98].

2.4 Unification Theory

Unification theory is concerned with problems of the following kind: Let a and b be constants, f and g binary function symbols, X and Y variables, and $\mathcal{E}$ an equational system. Does *unification theory*

$$\mathcal{E} \cup \mathcal{E}_{\approx} \models (\exists X, Y)\, f(X, g(a, b)) \approx f(g(Y, b), X) \tag{2.15}$$

hold? Such *decision problems* have a solution iff we find a substitution θ (often called an *$\mathcal{E}$-unifier*) such that *$\mathcal{E}$-unifier*

$$f(X, g(a, b))\theta \approx_{\mathcal{E}} f(g(Y, b), X)\theta$$

holds. In addition to the decision problem there is also the problem of finding a *unification algorithm*, i.e., a procedure which enumerates the $\mathcal{E}$-unifiers, given $\mathcal{E}$ and the two terms to be unified under $\mathcal{E}$. Let us consider some examples:

- If $\mathcal{E}$ is empty, then the decision problem (2.15) is the well-known unificiation problem and is decidable. The most general unifier of the two terms to be unified is the unique (modulo variable renaming) minimal solution. Several unification algorithms are known [Rob65, PW78, MM82]. For example,

  $$\theta_1 = \{X \mapsto g(a, b),\ Y \mapsto a\}$$

  is a solution for (2.15).

- If

  $$\mathcal{E} = \{f(X) \approx X\}$$

  then

  $$\{Y \mapsto a\}$$

  is an $\mathcal{E}$-unifier for $g(f(a), a)$ and $g(Y, Y)$. One should observe that the terms $g(f(a), a)$ and $g(Y, Y)$ are not unifiable (under the empty equational theory).

- If $\mathcal{E}$ states that f is commutative, i.e., if

  $$\mathcal{E} = \{f(X, Y) \approx f(Y, X)\},$$

  then θ_1 is still a solution for (2.15). However, it is no longer a minimal one because, for example,

  $$\theta_2 = \{Y \mapsto a\}$$

is also a solution for (2.15). This is because

$$f(X, g(a, b))\theta_2 = f(X, g(a, b)) \approx_{\mathcal{E}} f(g(a, b), X) = f(g(Y, b), X)\theta_2.$$

Moreover, θ_2 is more general than θ_1 because

$$\theta_1 = \theta_2\{X \mapsto g(a, b)\}.$$

Whereas under the empty equational system there is at most one most general unifier, this does not hold any longer for unification under commutativity. There exist terms such that the decision problem under commutativity has more than one most general unifier, but it can be shown that their maximum number is always finite.

- The problem becomes entirely different if we assume that

$$\mathcal{E} = \{f(X, f(Y, Z)) \approx f(f(X, Y), Z)\},$$

i.e., if we assume that f is associative. In this case θ_1 is still a solution for (2.15), but

$$\theta_3 = \{X \mapsto f(g(a, b), g(a, b)),\ Y \mapsto a\}$$

is also a solution because

$$\begin{array}{rcl} f(X, g(a, b))\theta_3 & = & f(f(g(a, b), g(a, b)), g(a, b)) \\ & \approx_{\mathcal{E}} & f(g(a, b), f(g(a, b), g(a, b))) \\ & = & f(g(Y, b), X)\theta_3. \end{array}$$

One should observe that neither is θ_1 more general than θ_3 nor is θ_3 more general than θ_1. In addition,

$$\theta_4 = \{X \mapsto f(g(a, b), f(g(a, b), g(a, b))),\ Y \mapsto a\}$$

is yet another independent solution, and it is easy to see that there are infinitely many independent solutions for (2.15).

- Finally, the situation changes once again if we assume that f is associative and commutative. In this case for any pair of terms, the number of independent solutions is either zero, in which case the terms are not unifiable, or finite.

2.4.1 Unification under Equality

As shown before, any equational system $\mathcal{E}$ over some alphabet induces a finest congruence relation $\approx_{\mathcal{E}}$ on the set of terms over the alphabet. An *$\mathcal{E}$-unification problem* consists of an equational system $\mathcal{E}$ and an equation $s \approx t$ and involves the question of whether

$\mathcal{E}$-unification

$$\mathcal{E} \cup \mathcal{E}_{\approx} \models \exists\, s \approx t,$$

where the existential quantifier denotes the existential closure of $s \approx t$. An *$\mathcal{E}$-unifier* for this problem is a substitution θ such that

$\mathcal{E}$-unifier

$$s\theta \approx_{\mathcal{E}} t\theta$$

and is a solution for the $\mathcal{E}$-unification problem. The set of all $\mathcal{E}$-unifiers for this problem is denoted by $U_{\mathcal{E}}(s,t)$.

$U_{\mathcal{E}}(s,t)$

Two substitutions η and θ are said to be *$\mathcal{E}$-equal* on a set $\mathcal{V}$ of variables iff $X\eta \approx_{\mathcal{E}} X\theta$ for all $X \in \mathcal{V}$. As an example let

$\mathcal{E}$-equal substitutions

$$\mathcal{E} = \{f(X) \approx X\}$$

and consider the substitutions

$$\{Y \mapsto a\}$$

and

$$\{Y \mapsto f(a)\}.$$

They are $\mathcal{E}$-equal on $\{X, Y\}$.

As in the case where $\mathcal{E}$ is empty, one does not need to consider the set of all $\mathcal{E}$-unifiers in most applications. It is usually sufficient to consider a complete set of $\mathcal{E}$-unifiers, i.e., a set of $\mathcal{E}$-unifiers from which all $\mathcal{E}$-unifiers can be generated by instantiation and equality modulo $\mathcal{E}$. Let $\mathcal{V}$ be a set of variables and θ and η be two substitutions. η is called an *$\mathcal{E}$-instance* of θ on $\mathcal{V}$, in symbols $\eta \leq_{\mathcal{E}} \theta[\mathcal{V}]$, iff there exists a substitution τ such that $X\eta \approx_{\mathcal{E}} X\theta\tau$ for all $X \in \mathcal{V}$. Obviously, if θ is a solution for an $\mathcal{E}$-unification problem and η is an $\mathcal{E}$-instance of θ, then η is a solution for this problem as well. η is called a *strict $\mathcal{E}$-instance of θ on $\mathcal{V}$*, in symbols $\eta <_{\mathcal{E}} \theta[\mathcal{V}]$ iff $\eta \leq_{\mathcal{E}} \theta$ and η and θ are not $\mathcal{E}$-equal. If neither $\theta \leq_{\mathcal{E}} \eta[\mathcal{V}]$ nor $\eta \leq_{\mathcal{E}} \theta[\mathcal{V}]$ then θ and η are said to be *incomparable*.

$\mathcal{E}$-instance $\leq_{\mathcal{E}}$

strict $\mathcal{E}$-instance $<_{\mathcal{E}}$

incomparable unifiers

As an example let

$$\mathcal{E} = \{f(X,Y) \approx f(Y,X)\},$$
$$\theta = \{X \mapsto f(a,Y)\},$$

and

$$\eta = \{X \mapsto f(b,a),\ Y \mapsto b\}.$$

In this case,

$$\eta \leq_{\mathcal{E}} \theta[\{X,Y\}]$$

because we find a substitution

$$\tau = \{Y \mapsto b\}$$

such that

$$X\eta = f(b,a) \approx_{\mathcal{E}} f(a,b) = X\theta\tau$$

and

$$Y\eta = b = Y\theta\tau.$$

Moreover, θ and η are not $\mathcal{E}$-equal on $\{X,Y\}$ because

$$Y\eta = b \not\approx_{\mathcal{E}} Y = Y\theta$$

and, hence,

$$\eta <_{\mathcal{E}} \theta[\{X,Y\}].$$

The substitutions θ_3 and θ_4 discussed in the introductory example where f was associative are incomparable $\mathcal{E}$-unifiers.

Recall that $U_{\mathcal{E}}(s,t)$ denotes the set of all $\mathcal{E}$-unifiers for the terms s and t. A set $\mathcal{S}$ of substitutions is said to be a *complete set of $\mathcal{E}$-unifiers for s and t* if it satisfies the following conditions:

complete set of unifiers

1. $\mathcal{S} \subseteq U_{\mathcal{E}}(s,t)$ and
2. for all $\eta \in U_{\mathcal{E}}(s,t)$ there exists $\theta \in \mathcal{S}$ such that $\eta \leq_{\mathcal{E}} \theta[var(s) \cup var(t)]$.

In other words, a set of substitutions is complete for two terms iff each element of this set is an $\mathcal{E}$-unifier for the terms and each $\mathcal{E}$-unifier for the terms is an $\mathcal{E}$-instance of some element of this set. Often, complete sets of $\mathcal{E}$-unifiers for s and t are denoted by $cU_{\mathcal{E}}(s,t)$.

$cU_{\mathcal{E}}(s,t)$

minimal complete set of unifiers

For reasons of efficiency a complete set of $\mathcal{E}$-unifiers should be as small as possible. Thus, we are interested in *minimal complete sets of $\mathcal{E}$-unifiers for s and t*. Such a set $\mathcal{S}$ is complete and satisfies the additional condition:

3. for all $\theta, \eta \in \mathcal{S}$ we find that $\theta \leq_{\mathcal{E}} \eta[var(s) \cup var(t)]$ implies $\theta = \eta$.

$\mu U_{\mathcal{E}}(s,t)$

$\equiv_{\mathcal{E}}$

Often, minimal complete sets of $\mathcal{E}$-unifiers for s and t are denoted by $\mu U_{\mathcal{E}}(s,t)$. Let $\theta \equiv_{\mathcal{E}} \eta[\mathcal{V}]$ iff $\eta \leq \theta[\mathcal{V}]$ and $\theta \leq \eta[\mathcal{V}]$. A minimal complete set of $\mathcal{E}$-unifiers for s and t is unique modulo $\equiv_{\mathcal{E}} [var(s) \cup var(t)]$, if it exists.

As an example consider the terms $s = f(X,a)$ and $t = f(a,Y)$. Let

$$\mathcal{E} = \{f(X, f(Y,Z)) = f(f(X,Y),Z)\}$$

and suppose that the constant symbol a and the binary function symbol f are the only function symbols in the underlying alphabet. The substitution

$$\theta = \{X \mapsto a,\ Y \mapsto a\}$$

is an $\mathcal{E}$-unifier for s and t, and so is

$$\eta = \{X \mapsto f(a,Z),\ Y \mapsto f(Z,a)\}.$$

It is easy to see that the set $\{\theta, \eta\}$ is a complete set of $\mathcal{E}$-unifiers. Moreover, because θ and η are incomparable under $\leq_{\mathcal{E}}$, this set is minimal.

Whenever there exists a finite complete set of $\mathcal{E}$-unifiers and the relation $\leq_{\mathcal{E}}$ is decidable, then there exists also a minimal one. This set can be obtained from the complete set of $\mathcal{E}$-unifiers by removing each unifier which is an $\mathcal{E}$-instance of some other unifier.

In general, however, we must be aware of the following result, which is due to Fages and Huet [FH83, FH86]:

Theorem 2.5 *Minimal complete sets of $\mathcal{E}$-unifiers do not always exist.*

To prove this theorem we consider the term rewriting system

$$\mathcal{R} = \{f(a,X) \to X,\ g(f(X,Y)) \to g(Y)\}$$

and show that $\mu U_{\mathcal{E}_{\mathcal{R}}}(g(X), g(a))$ does not exist. It should be noted that $\mathcal{R}$ is canonical. We define

$$\begin{array}{lcll} \sigma_0 & = & \{X \mapsto a\} & \\ \sigma_1 & = & \{X \mapsto f(X_1, a)\} & = \{X \mapsto f(X_1, X\sigma_0) \\ \vdots & & & \\ \sigma_i & = & \{X \mapsto f(X_i, X\sigma_{i-1})\} & \end{array}$$

and

$$\mathcal{S} = \{\sigma_i \mid i \geq 0\}.$$

It is not too difficult to show that $\mathcal{S}$ is a complete set of $\mathcal{E}_\mathcal{R}$-unifiers for $g(X)$ and $g(a)$. With

$$\rho_i = \{X_i \mapsto a\}$$

we find for all $i > 0$ that

$$X\sigma_i\rho_i = f(a, X\sigma_{i-1}) \approx_{\mathcal{E}_\mathcal{R}} X\sigma_{i-1}.$$

Hence,

$$\sigma_{i-1} \leq_{\mathcal{E}_\mathcal{R}} \sigma_i[\{X\}]$$

for all $i > 0$. Because

$$X\sigma_i = f(X_i, X\sigma_{i-1}) \not\approx_{\mathcal{E}_\mathcal{R}} X\sigma_{i-1}$$

we conclude

$$\sigma_{i-1} <_{\mathcal{E}_\mathcal{R}} \sigma_i[\{X\}]$$

for all $i > 0$.

Now assume that $\mathcal{S}'$ is a minimal and complete set of $\mathcal{E}_\mathcal{R}$-unifiers for $g(X)$ and $g(a)$. Because $\mathcal{S}$ is complete, we find that for all $\theta \in \mathcal{S}'$ there exists a $\sigma_i \in \mathcal{S}$ such that

$$\theta \leq_{\mathcal{E}_\mathcal{R}} \sigma_i[\{X\}].$$

Because

$$\sigma_i <_{\mathcal{E}_\mathcal{R}} \sigma_{i+1}[\{X\}]$$

we learn that

$$\theta <_{\mathcal{E}_\mathcal{R}} \sigma_{i+1}[\{X\}].$$

Conversely, because $\mathcal{S}'$ is complete we find that there exists $\sigma \in \mathcal{S}'$ such that

$$\sigma_{i+1} \leq_{\mathcal{E}_\mathcal{R}} \sigma[\{X\}].$$

Hence,

$$\theta <_{\mathcal{E}_\mathcal{R}} \sigma[\{X\}]$$

and, consequently, $\mathcal{S}'$ is not minimal. Figure 2.2 illustrates the situation. This contradicts our assumption and completes the proof.

Based on these observations, the *unification type* of an equational theory can be defined as follows. It is

unification type

- *unitary* iff a set $\mu U_\mathcal{E}(s,t)$ exists for all s, t and has cardinality 0 or 1,
- *finitary* iff a set $\mu U_\mathcal{E}(s,t)$ exists for all s, t and is finite,
- *infinitary* iff a set $\mu U_\mathcal{E}(s,t)$ exists for all s, t, and there are terms u and v such that $\mu U_\mathcal{E}(u,v)$ is infinite,
- *zero* iff there are terms s and t such that a set $\mu U_\mathcal{E}(s,t)$ does not exist.

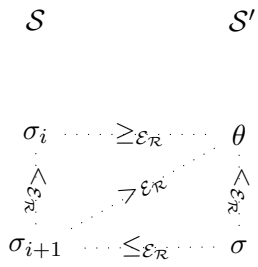


Figure 2.2: The situation leading to the contradiction in the proof of Theorem 2.5.

$\mathcal{E}$-unification procedure

An *$\mathcal{E}$-unification procedure* is a procedure which takes an equation $s \approx t$ as input and generates a subset of the set of $\mathcal{E}$-unifiers for s and t as output. It is said to be:

- *complete* iff it generates a complete set of $\mathcal{E}$-unifiers,
- *minimal* iff it generates a minimal complete set of $\mathcal{E}$-unifiers.

A *universal $\mathcal{E}$-unification procedure* is a procedure which takes an equational system $\mathcal{E}$ and an equation $s \approx t$ as input and generates a subset of the set of $\mathcal{E}$-unifiers for s and t as output. The notions of completeness and minimal unification procedures extend to universal unification procedures in the obvious way.

For a given equational system $\mathcal{E}$, unification theory is mainly concerned with finding answers for the following questions:

- Is it decidable whether an $\mathcal{E}$-unification problem is solvable?
- What is the unification type of $\mathcal{E}$?
- How can we obtain an efficient $\mathcal{E}$-unification algorithm or a preferably minimal $\mathcal{E}$-unification procedure?

It is important to note that the answers to these questions depend on the underlying alphabet or, more generally, the environment in which the unification problems have to be solved. Let $\mathcal{E}$ be an equational system. $\mathcal{E}$-unification problems are classified as follows. They are called:

- *elementary* iff the terms of the problem may contain only symbols that appear in $\mathcal{E}$,
- *with constants* iff the terms of the problem may contain additional free constants,
- *general* iff the terms of the problem may contain additional free function symbols of arbitrary arity.

For example, there exists an equational system for which elementary unification is decidable whereas unification with constants is undecidable [Bür86].

2.4.2 Examples

In this subsection the $\mathcal{E}$-unification problems for several equational theories are discussed. Table 2.10 taken from [BS94] shows some results concerning unification with constants.

$$\mathcal{E}_A = \{f(X, f(Y, Z)) \approx f(f(X, Y), Z)\}$$

defines the associativity of the function symbol f. Unification under $\mathcal{E}_A$ is needed for solving string unification problems or, equivalently, word problems.

associativity $\mathcal{E}_A$

$$\mathcal{E}_C = \{f(X, Y) \approx f(Y, X)\}$$

defines the commutativity of the function symbol f and

commutativity $\mathcal{E}_C$

$$\mathcal{E}_{AC} = \mathcal{E}_A \cup \mathcal{E}_C$$

defines an Abelian semi-group. This equational system is of particular importance because many mathematical operations such as addition or multiplication are associative and commutative. $\mathcal{E}_{AC}$ cannot be oriented into a terminating term rewriting system and consequently many questions have to be solved modulo $\mathcal{E}_{AC}$.

Abelian semi-group $\mathcal{E}_{AC}$

$$\mathcal{E}_{AG} = \mathcal{E}_{AC} \cup \{f(X, 1) \approx X,\ f(X, X^{-1}) \approx 1\}$$

defines an Abelian group. Unification problems under $\mathcal{E}_{AG}$ are equivalent to solving Diophantine equations over the set of integers.

Abelian group $\mathcal{E}_{AG}$

$$\mathcal{E}_{AI} = \mathcal{E}_A \cup \{f(X, X) \approx X\}$$

defines idempotent semi-groups.

idempotent semi-groups $\mathcal{E}_{AI}$

$$\begin{aligned}
\mathcal{E}_{CR1} = \{\ & f(X, f(Y, Z)) \approx f(f(X, Y), Z), \\
& f(X, 0) \approx X, \\
& f(X, X^{-1}) \approx 0, \\
& f(X, Y) \approx f(Y, X), \\
& g(X, g(Y, Z)) \approx g(g(X, Y), Z), \\
& g(X, Y) \approx g(Y, X), \\
& g(X, 1) \approx 1, \\
& g(X, f(Y, Z)) \approx f(g(X, Y), g(X, Z)), \\
& g(f(X, Y), Z) \approx f(g(X, Z), g(Y, Z)) \quad \}
\end{aligned}$$

defines a commutative ring with identity. The unification problem under $\mathcal{E}_{CR1}$ is equivalent to Hilbert's 10th problem, i.e., the problem of Diophantine solvability of polynomial equations.

commutative ring with identity $\mathcal{E}_{CR1}$

$$\begin{aligned}
\mathcal{E}_{DL} &= \{g(f(X, Y), Z) \approx f(g(X, Z), g(Y, Z))\} \\
\mathcal{E}_{DR} &= \{g(X, f(Y, Z)) \approx f(g(X, Y), g(X, Z))\} \\
\mathcal{E}_{D} &= \mathcal{E}_{DL} \cup \mathcal{E}_{DR} \\
\mathcal{E}_{DA} &= \mathcal{E}_{D} \cup \mathcal{E}_{A}
\end{aligned}$$

define left and right distributivity, both-sided distributivity as well as distributivity and

distributivity $\mathcal{E}_{DL}, \mathcal{E}_{DR}, \mathcal{E}_{D}, \mathcal{E}_{DA}$

Equational System	Unification Type	Unification decidable	Complexity of the decision problem
$\mathcal{E}_A$	infinitary	yes	NP-hard
$\mathcal{E}_C$	finitary	yes	NP-complete
$\mathcal{E}_{AC}$	finitary	yes	NP-complete
$\mathcal{E}_{AG}$	unitary	yes	polynomial
$\mathcal{E}_{AI}$	zero	yes	NP-hard
$\mathcal{E}_{CR1}$	zero	no	–
$\mathcal{E}_{DL}, \mathcal{E}_{DR}$	unitary	yes	polynomial
$\mathcal{E}_D$	infinitary	?	NP-hard
$\mathcal{E}_{DA}$	infinitary	no	–
$\mathcal{E}_{BR}$	unitary	yes	NP-complete

Table 2.10: Results on unification types and the decision problem for unification with constants.

associativity respectively. Finally,

$$\begin{aligned}\mathcal{E}_{BR} = \{ \quad & f(X,1) \approx 1, \\ & f(X,X) \approx X, \\ & f(X,Y) \approx f(Y,X), \\ & f(X,f(Y,Z)) \approx f(f(X,Y),Z), \\ & g(X,0) \approx 0, \\ & g(X,X) \approx X, \\ & g(X,Y) \approx g(Y,X), \\ & g(X,g(Y,Z)) \approx g(g(X,Y),Z), \\ & g(X,1) \approx X, \\ & g(X,f(Y,Z)) \approx f(g(X,Y),g(X,Z)) \quad \}\end{aligned}$$

Boolean ring $\mathcal{E}_{BR}$ defines Boolean rings. Unification modulo $\mathcal{E}_{BR}$ can be used to build Boolean expressions into programming languages, which then can be applied to, for example, the verification of circuit switches.

2.4.3 Remarks

$\mathcal{E}$-matching An *$\mathcal{E}$-matching problem* consists of an equational system $\mathcal{E}$ and an equation $s \approx t$ and is the question of whether there exists a substitution θ such that

$$s \approx_{\mathcal{E}} t\theta.$$

Hence, it differs from $\mathcal{E}$-unification problems in that the substitution θ is only applied to one term. All concepts relating to $\mathcal{E}$-unification can be defined for $\mathcal{E}$-matching as well.

general $\mathcal{E}$-unification Besides unification under a specific equational theory, one is often interested in so-called *general* $\mathcal{E}$-unification problems, i.e. problems, where the equational system is also part of the input. Such problems arise naturally within equational programming, where the program is a set of equations. Paramodulation, narrowing and rewriting may be applied in these cases as discussed in the previous section.

combination problem Another problem which has received much attention is the so-called *combination problem*: given two equational systems $\mathcal{E}_1$ and $\mathcal{E}_2$, can the results and unification algorithms

for $\mathcal{E}_1$ and $\mathcal{E}_2$ be combined to handle unification problems under $\mathcal{E}_1 \cup \mathcal{E}_2$?

applications

Unification problems occur in many application areas such as the following: databases and information retrieval, computer vision, natural language processing and text manipulation systems, knowledge based systems, planning and scheduling systems, pattern-directed programming languages, logic programming systems, computer algebra systems, deduction systems and non-classical reasoning systems. Excellent overviews are presented in [BS94] and [BS99].

2.4.4 Multisets

Multisets are an important data structure for many applications in Computer Science and Artificial Intelligence. They are particularly appropriate whenever production and consumption of resources are to be modeled.

multiset

Informally, multisets are sets in which each element can occur more than once. Formally, let $\dot{\emptyset}$ denote the empty multiset and let the parentheses $\dot{\{}$ and $\dot{\}}$ be used to enclose the elements of a multiset. Analogously to the case of sets, the following relations and operations on multisets are defined: membership, union, difference, intersection, submultiset and equality. Let $\mathcal{M}$, $\mathcal{M}_1$, and $\mathcal{M}_2$ be finite multisets. Then these relations and operations apply as follows:

- *Membership*: $X \in_k \mathcal{M}$ iff X occurs precisely k-times in $\mathcal{M}$, for $k \geq 0$. *membership*

  For example, if $\mathcal{M}$ is the multiset

  $$\dot{\{}a, b, c, a, b, a\dot{\}},$$

  then $a \in_3 \mathcal{M}$, $b \in_2 \mathcal{M}$, $c \in_1 \mathcal{M}$ and $d \in_0 \mathcal{M}$.

- *Equality*: $\mathcal{M}_1 \doteq \mathcal{M}_2$ iff for all X we find $X \in_k \mathcal{M}_1$ iff $X \in_k \mathcal{M}_2$. *equality*

  For example,

  $$\dot{\{}a, b, a\dot{\}} \doteq \dot{\{}a, a, b\dot{\}}.$$

- *Union*: $X \in_m \mathcal{M}_1 \dot{\cup} \mathcal{M}_2$ iff there exist $k, l \geq 0$ such that $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, and $m = k + l$. *union*

  For example, if

  $$\mathcal{M}_1 \doteq \dot{\{}a, b, c\dot{\}}$$

  and

  $$\mathcal{M}_2 \doteq \dot{\{}a, b, a\dot{\}},$$

  then

  $$\mathcal{M}_1 \dot{\cup} \mathcal{M}_2 \doteq \dot{\{}a, b, c, a, b, a\dot{\}}.$$

- *Difference*: $X \in_m \mathcal{M}_1 \dot{\setminus} \mathcal{M}_2$ iff there exist $k, l \geq 0$ such that either $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, $k > l$, and $m = k - l$ or $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, $k \leq l$, and $m = 0$. *difference*

  For example, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are as above, then

  $$\mathcal{M}_1 \dot{\setminus} \mathcal{M}_2 \doteq \dot{\{}c\dot{\}}$$

  and

  $$\mathcal{M}_2 \dot{\setminus} \mathcal{M}_1 \doteq \dot{\{}a\dot{\}}.$$

intersection

- *Intersection*: $X \in_m \mathcal{M}_1 \dot{\cap} \mathcal{M}_2$ iff there exist $k, l \geq 0$ such that $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, and $m = \mathtt{min}\{k, l\}$, where $\mathtt{min}$ maps $\{k, l\}$ to its minimal element.

  For example, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are as above, then

$$\mathcal{M}_1 \dot{\cap} \mathcal{M}_2 \doteq \dot{\{}a, b\dot{\}}.$$

submultiset

- *Submultiset*: $\mathcal{M}_1 \dot{\subseteq} \mathcal{M}_2$ iff $\mathcal{M}_1 \dot{\cap} \mathcal{M}_2 \doteq \mathcal{M}_1$.

  For example,

$$\dot{\{}a, b, a\dot{\}} \dot{\subseteq} \dot{\{}a, b, c, a, b, a\dot{\}}.$$

Multisets can be represented (extensionally) with the help of a binary function symbol $\circ$ (written infix) which is associative, commutative, and admits a unit element (constant) 1. Formally, consider an alphabet with set $\mathcal{V}$ of variables and a set $\mathcal{F}$ of function symbols which contains $\circ$ and 1. Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ be the set of terms built over $\mathcal{F}$ and $\mathcal{V}$, and $\mathcal{F}^- = \mathcal{F} \setminus \{\circ, 1\}$

fluent

Let us call the non-variable elements of $T(\mathcal{F}^-, \mathcal{V})$ *fluents*.[4] These are the terms with a *leading* function symbol like $f(X, a)$ or c. In the following we will consider multisets of fluents.

fluent term

The set of *fluent terms* is the smallest set meeting the following conditions

1. 1 is a fluent term,
2. each fluent is a fluent term, and
3. if s and t are fluent terms, then $s \circ t$ is a fluent term.

As the sequence of fluents occurring in a fluent term is not important, we consider the following equational system:

$$\mathcal{E}_{AC1} = \{ \quad \begin{array}{l} X \circ (Y \circ Z) \approx (X \circ Y) \circ Z \\ X \circ Y \approx Y \circ X \\ X \circ 1 \approx X \end{array} \quad \}$$

For example,

$$on(a, b) \circ on(b, c) \circ ontable(c) \circ clear(a)$$

is a fluent term which, informally, can be interpreted to denote the state shown in Figure 2.3. $on(X, Y)$ states that block X is on block Y, $ontable(X)$ states that block X is on the table, and $clear(X)$ states that block X is clear, i.e., that nothing is on top of it. This example is taken from the so-called *blocks world*, which is often used in Artificial Intelligence to exemplify actions and causality (see also Chapter 3). Alternatively, the table can be interpreted as a container terminal and the blocks as containers. The fluent term

blocks world

$$clear(X) \circ on(X, Y)$$

can informally be interpreted as the precondition of a move action which states that block or container X can be moved if it is on top of some other block Y and is clear.

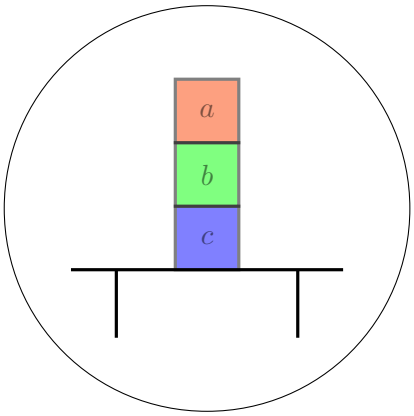


Figure 2.3: The blocks a, b, and c form a tower standing on a table. Block a is clear.

There is a straightforward mapping from fluent terms to multisets of fluents and vice versa. The mapping $\cdot^I$ from fluent terms to multisets of fluents is defined as follows. Let $\cdot^I$ t be a fluent term:

$$t^I = \begin{cases} \dot{\emptyset} & \text{if } t = 1, \\ \dot{\{}t\dot{\}} & \text{if } t \text{ is a fluent, and} \\ u^I \dot{\cup} v^I & \text{if } t = u \circ v \end{cases}$$

The inverse mapping $\cdot^{-I}$ from multisets of fluents to fluent terms exists and is defined as $\cdot^{-I}$ follows. Let $\mathcal{M}$ be a multiset of fluents:

$$\mathcal{M}^{-I} = \begin{cases} 1 & \text{if } \mathcal{M} \doteq \dot{\emptyset}, \\ s \circ \mathcal{N}^{-I} & \text{if } \mathcal{M} \doteq \dot{\{}s\dot{\}} \dot{\cup} \mathcal{N}. \end{cases}$$

It is easy to see that for a fluent term t and a multiset $\mathcal{M}$ of fluents, the equations

$$t \approx_{AC1} (t^I)^{-I}$$

and

$$\mathcal{M} \doteq (\mathcal{M}^{-I})^I$$

hold. In other words, there is a one-to-one correspondence between fluent terms and multisets of fluents.

Returning to the blocks world example we find that

$$\begin{aligned} &(on(a,b) \circ on(b,c) \circ ontable(c) \circ clear(a))^I \\ &\doteq \dot{\{}on(a,b), on(b,c), ontable(c), clear(a)\dot{\}} \end{aligned} \tag{2.16}$$

and

$$(clear(X) \circ on(X,Y))^I \doteq \dot{\{}clear(X), on(X,Y)\dot{\}}. \tag{2.17}$$

Having defined a representation for multisets of fluents, we are interested in the operations on this representation. Leaving the definition of the operations union, intersection and difference on fluent terms to the interested reader, we concentrate on the following problems:

submultiset matching problem

- The *submultiset matching problem* consists of a multiset $\mathcal{M}$ and a ground multiset $\mathcal{N}$. It is the question of whether there exists a substitution θ such that $\mathcal{M}\theta \dot{\subseteq} \mathcal{N}$.

submultiset unification problem

- The *submultiset unification problem* consists of two multisets $\mathcal{M}$ and $\mathcal{N}$. It is the question of whether there exists a substitution θ such that $\mathcal{M}\theta \dot{\subseteq} \mathcal{N}\theta$.

For example, to determine whether block (or container) a can be moved in the state depicted in Figure 2.3 we have to solve the submultiset matching problem of the multiset occurring in (2.17) against the multiset occurring in (2.16). It is easy to see that the substitution

$$\theta = \{X \mapsto a,\ Y \mapsto b\}$$

solves this problem.

With the help of the mapping $\cdot^{-I}$ these problems can be transformed into $\mathcal{E}_{AC1}$-matching and $\mathcal{E}_{AC1}$-unification problems:

fluent matching problem

- The *fluent matching problem* consists of a fluent term s, a ground fluent term t and a variable X not occurring in s. It is the question of whether there exists a substitution θ such that $(s \circ X)\theta \approx_{AC1} t$.

fluent unification problem

- The *fluent unification problem* consists of two fluent terms s and t and a variable X not occurring in s or t. It is the question of whether there exists a substitution θ such that $(s \circ X)\theta \approx_{AC1} t\theta$.

It is easy to see that θ is a solution for the fluent matching problem consisting of s, t, and X iff $\theta|_{var(s)}$ is a solution for the submultiset matching problem consisting of s^I and t^I. Moreover, we find that in this case

$$(X\theta)^I \doteq t^I \dot{\setminus} (s\theta)^I .$$

Similarly, θ is a solution for the fluent unification problem consisting of s, t, and X iff $\theta|_{var(s)}$ is a solution for the submultiset unification problem consisting of s^I and t^I. Moreover, we find that in this case

$$(X\theta)^I \doteq (t\theta)^I \dot{\setminus} (s\theta)^I .$$

The fluent matching and the fluent unification problem are decidable, finitary, and there always exists a minimal complete set of matchers and unifiers. Table 2.11 shows an algorithm for computing minimal complete sets of matchers for fluent matching problems.[5]

Fluent unification and matching problems will play a major rule in reasoning about situations, actions and causality as will be demonstrated in Chapter 3.

[4] These elements are called fluents because they will denote resources that may or may not be available in a certain state, and may be produced and consumed by actions (see Chapter 3).

[5] A selection step in a procedure is said to be *don't-care non-deterministic* iff there is no need to reconsider; a selection step in a procedure is said to be *don't-know non-deterministic* iff all possible choices must eventually be taken into account. In other words, one never has to return to a don't-care non-deterministic selection, whereas a don't know non-deterministic selection defines a branching point of the procedure and all branches need to be investigated.

`Input:` A fluent matching problem $(\exists\theta)\,(s \circ X)\theta \approx_{AC1} t$?
(where t is ground and X does not occur in s).
`Output:` A solution θ of the fluent matching problem, if it is solvable;
failure, otherwise.

1. $\theta = \varepsilon$;
2. if $s \approx_{AC1} 1$ then return $\theta\{X \mapsto t\}$;
3. don't-care non-deterministically select a fluent u from s and remove u from s ;
4. don't-know non-deterministically select a fluent v from t such that there exists a substitution η with $u\eta = v$;
5. if such a fluent exists then apply η to s, delete v from t and let $\theta := \theta\eta$, otherwise stop with failure;
6. goto 2;

Table 2.11: An algorithm for the fluent matching problem consisting of s, t, and X. A complete set of matchers is obtained by considering all possible choices in step 4. This set is always finite because s contains only finitely many fluents and in step 3 an element is deleted from s. A complete minimal set is obtained by removing redundant elements.

2.5 Final Remarks

Paramodulation has been introduced in [Bra75]. The section on term rewriting is based on [Pla93], whereas the section on unification theory is based on [BS94]. Fluent matching and unification problems were considered in [HST93].

Chapter 3

Actions and Causality

The design of rational agents which perceive and act upon their environment is one of the main goals of Intellectics, i.e., Artificial Intelligence and Cognition [Bib92]. Inevitably, such rational agents need to represent and reason about states, actions, and causality, and it comes as no surprise that these topics have a long history in Intellectics. Already in 1963 John McCarthy proposed a predicate logic formalization, viz. the situation calculus [McC63, MH69], which has been extensively studied and extended ever since (see e.g. [Lif90, Rei91]). The core idea underlying this line of research is that a state is a snapshot of the world and that actions mapping states onto states are the only means for changing states.

States are characterized by multisets of fluents, which may or may not be present in certain states.[1] Figure 2.3 shows a state where three blocks form a tower. The fluents are the terms $on(a, b)$, $on(b, c)$, $ontable(c)$, and $clear(a)$. Moving block a from the tower to the table t leads to another state which can be obtained from the initial state by deleting the fluent $on(a, b)$ and adding the fluents $ontable(a)$ and $clear(b)$.

Because it is impossible to completely describe the world at a particular time or to completely specify an action, each state and each action can only be partially known. This gives rise to several difficult and hence interesting problems like the frame, ramification, qualification, and prediction problems.

- The *frame problem* is the question of which fluents are unaffected by the execution of an action. For example, if we move block a from the tower as described before, then we typically assume that the blocks b and c are unaffected by this action. *frame problem*
- The *ramification problem* is the question of which fluents are really present after the execution of an action. For example, if we move block b in the situation shown in Figure 2.3, then we typically assume that block a goes with it. *ramification problem*
- The *qualification problem* is the question of which preconditions have to be satisfied such that an action is executable. For example, block a may be too heavy so that two robots are needed for moving it around. *qualification problem*
- The *prediction problem* is the question of how long fluents are present in certain *precondition problem*

[1] There are arguments over whether states should be regarded as sets or multisets. Sometimes, it is more adequate to think of states as sets, whereas sometimes it is not. For example, properties are typically modeled as sets, whereas resources are modeled as multisets.

situations. For example, if you have parked your bycicle outside of the lecture hall before the lecture, then you typically assume that it is still parked there after the lecture. Occasionally however, it is not.

All these problems have a cognitive as well as a technical aspect. We are cognitively interested in how humans solve these problems (because we are faced with them as well) and we are technically interested in how we can handle these problems on a computer. As far as the latter aspect is concerned, we are particularly interested in finding a formalism which allows us to adequately represent these problems and to adequately compute solutions for these problems.

We take the position that computation requires representation and reasoning. Following [McC63], we intend to build a system which meets the following specification:[2]

- General properties of causality and facts about the possibility and results of actions are given as formulas.
- It is a logical consequence of the facts of a state and the general axioms that goals can be achieved by performing certain actions.

In this chapter, conjunctive planning problems are considered. Examples are taken from the so-called simple blocks world. It is shown how these problems can be represented and solved within the fluent calculus. It is also demonstrated how the technical aspects of the frame problem can be dealt within the fluent calculus. In doing so, we will use the fluent matching algorithm developed in Subsection **??** and built it into SLD-resolution.

3.1 Conjunctive Planning Problems

The planning problems considered in this section consist of a multiset

$$\mathcal{I}: \dot{\{}i_1, \ldots, i_m\dot{\}}$$

of ground fluents called the *initial state*, a multiset

$$\mathcal{G}: \dot{\{}g_1, \ldots, g_n\dot{\}}$$

action of ground fluents called the *goal state* and a finite set of *actions* of the form

$$\dot{\{}c_1, \ldots, c_l\dot{\}} \Rightarrow \dot{\{}e_1, \ldots, e_k\dot{\}},$$

condition where $\dot{\{}c_1, \ldots, c_l\dot{\}}$ and $\dot{\{}e_1, \ldots, e_k\dot{\}}$ are multisets of fluents called *conditions* and *effects*,
effect respectively. We further assume that each variable occurring in the effects of an action occurs also in its conditions, i.e., in at least one of its fluents. A *conjunctive planning*
conjunctive planning problem *problem* is the question of whether there exists a sequence of actions such that its execution transforms the initial state into the goal state.

Let $\mathcal{S}$ be a multiset of ground fluents. An action

$$\dot{\{}c_1, \ldots, c_l\dot{\}} \Rightarrow \dot{\{}e_1, \ldots, e_k\dot{\}}$$

applicable actions is *applicable* in $\mathcal{S}$ iff there is a substitution θ such that

applicable action

$$\dot{\{}c_1\theta, \ldots, c_l\theta\dot{\}} \;\dot{\subseteq}\; \mathcal{S}.$$

One should observe that if θ is restricted to the variables occurring in $\dot{\{}c_1, \ldots, c_l\dot{\}}$ and $\mathcal{S}$ is ground then $range(\theta)$ contains only ground terms. The *application of an action* leads to the state

application of action

$$(\mathcal{S} \dot{\setminus} \dot{\{}c_1\theta, \ldots, c_l\theta\dot{\}}) \dot{\cup} \dot{\{}e_1\theta, \ldots, e_k\theta\dot{\}}.$$

As a consequence of the assumption that each variable occurring in the effects of an action occurs also in the condition of an action, the new state is ground whenever $\mathcal{S}$ is ground. A sequence $[a_1, \ldots, a_n]$ of actions, also called a *plan*, transforms state $\mathcal{S}$ into $\mathcal{S}'$ iff $\mathcal{S}'$ is the result of successively applying the actions in $[a_1, \ldots, a_n]$ to $\mathcal{S}$.

plan

Finally, a goal $\mathcal{G}$ is *satisfied* iff there is a *plan* p, i.e., a sequence of actions $[a_1, \ldots, a_n]$, which transforms the initial state $\mathcal{I}$ into a state $\mathcal{S}$ such that $\mathcal{G} \dot{\subseteq} \mathcal{S}$. If there exists such a plan p, then p is called a *solution* for the planning problem.

satisfied goal

solution

In the next subsection these notions are exemplified in a particular scenario, the so-called blocks worlds.

3.2 Blocks World

The simple blocks world is a toy domain, where blocks can be moved around with the help of a robot. Alternatively, you may think of a container terminal, where containers are loaded from trucks to trains or ships and vice versa. There are four actions:

- The *pickup* action picks up a block V from the table if the block is clear, and the arm of the robot is empty.

  pickup

  $$pickup(V): \quad \dot{\{}clear(V), ontable(V), empty\dot{\}} \Rightarrow \dot{\{}holding(V)\dot{\}}$$

- The *unstack* action unstacks a block V from another block W if the former block is clear and the arm of the robot is empty.

  unstack

  $$unstack(V,W): \quad \dot{\{}clear(V), on(V,W), empty\dot{\}} \Rightarrow \dot{\{}holding(V), clear(W)\dot{\}}$$

- The *putdown* action puts a block V held by the robot onto the table.

  putdown

  $$putdown(V): \quad \dot{\{}holding(V)\dot{\}} \Rightarrow \dot{\{}clear(V), ontable(V), empty\dot{\}}$$

- The *stack* action stacks a block V held by the robot on another block W if the latter block is clear.

  stack

  $$stack(V,W): \quad \dot{\{}holding(V), clear(W)\dot{\}} \Rightarrow \dot{\{}on(V,W), clear(V), empty\dot{\}}$$

Figure 3.1 shows a simple planning problem known as Sussman's anomaly [Sus75] with

Sussman's anomaly

[2] In [McC63] it is also required that the formal descriptions of states should correspond as closely as possible to what people may reasonably be presumed to know about them when deciding what to do. Although this is probably the most interesting and challenging requirement in the context of common sense reasoning, we do not consider it at the moment.

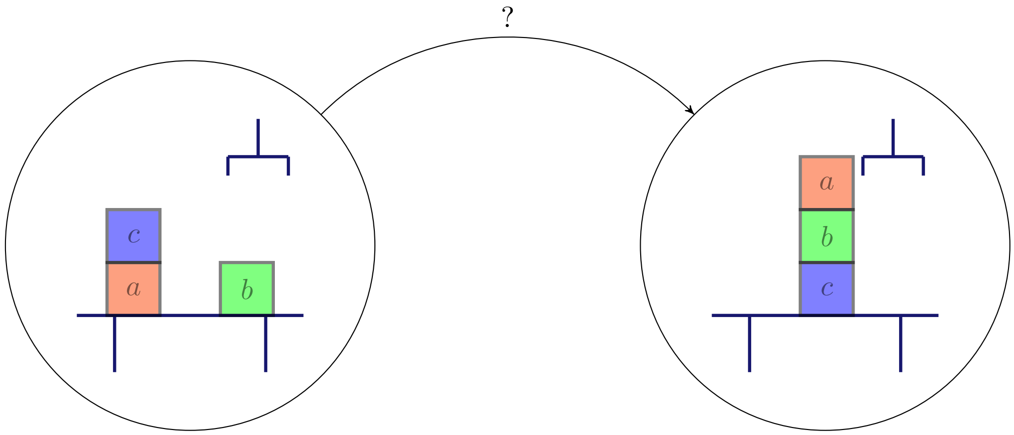


Figure 3.1: A blocks world example: Sussman's anomaly.

initial state

$$\dot{\{} ontable(a), ontable(b), on(c,a), clear(b), clear(c), empty \dot{\}}$$

and goal state

$$\dot{\{} ontable(c), on(b,c), on(a,b), clear(a), empty \dot{\}}.$$

It can be solved by the plan

$$[unstack(c,a), putdown(c), pickup(b), stack(b,c), pickup(a), stack(a,b)]. \quad (3.1)$$

One should observe that the various subgoals of the goal state cannot be achieved independently and one after the other. The interested reader is encouraged to see what happens if she first attempts to find the shortest plan establishing $on(b,c)$ (or $on(a,b)$) and, thereafter, to establish the other subgoal $on(a,b)$ (or $on(b,c)$).

3.2.1 A Fluent Calculus Implementation

The simple fluent calculus is a first order calculus, where conjunctive planning problems can be represented and solved [HS90]. States as well as conditions and effects are represented by fluent terms. Actions are represented using a ternary relation symbol *action*, *action* where the arguments encode the conditions, the name, and the effects of the action. For example, the actions of the simple blocks world are represented by the set of clauses

$$\begin{array}{lcl} \mathcal{K}_A & = & \{\ action(clear(V) \circ ontable(V) \circ empty,\ pickup(V),\ holding(V)), \\ & & \quad action(clear(V) \circ on(V,W) \circ empty,\ unstack(V,W),\ holding(V) \circ clear(W)), \\ & & \quad action(holding(V),\ putdown(V),\ clear(V) \circ ontable(V) \circ empty), \\ & & \quad action(holding(V) \circ clear(W),\ stack(V,W),\ on(V,W) \circ clear(V) \circ empty)\ \}. \end{array}$$

causes With the help of a ternary relation symbol *causes*, we can express that a state is transformed into another one by applying sequences of actions.

$$\begin{array}{lcl} \mathcal{K}_C & = & \{\ causes(X,[\,],Y) \leftarrow X \approx Y \circ Z, \\ & & \quad causes(X,[V|W],Y) \leftarrow action(P,V,Q) \wedge P \circ Z \approx X \wedge causes(Z \circ Q,W,Y), \\ & & \quad X \approx X \qquad \}. \end{array}$$

The first clause in $\mathcal{K}_C$ states that there is nothing to do ([]), if the goal state Y is contained in the current state X. The second clause is read declaratively as

> the execution of the plan $[V|W]$ transforms state X into state Y if there is an action with condition P, name V, effect Q and there is a Z with $P \circ Z \approx_{AC1} X$ and the plan W transforms $Z \circ Q$ into Y

or procedurally as

> to solve the problem of whether there exists a plan $[V|W]$ such that its execution transforms the state X into Y, find an action with condition P, name V, and effect Q, find a Z with $P \circ Z \approx_{AC1} X$ and solve the problem of whether there exists a plan W such that its execution transforms the state $Z \circ Q$ into Y.

The third clause is the axiom of reflexivity needed to solve the equations occurring in the conditions of the first two clauses.

The question of whether there exists a plan P solving a conjunctive planning problem with initial state $\mathcal{I}$, goal state $\mathcal{G}$, and a given set of actions is represented by the question of whether

$$(\exists P)\ \mathit{causes}(\mathcal{I}^{-I}, P, \mathcal{G}^{-I})$$

is a logical consequence of $\mathcal{K}_A \cup \mathcal{K}_C \cup \mathcal{E}_{AC1} \cup \mathcal{E}_{\approx}$, where $\cdot^{-I}$ is the mapping from multisets to fluent terms and $\mathcal{E}_{AC1}$ is the equational system for fluent terms, both introduced in the previous Section 2.4.

Having fixed the alphabet and the language of the fluent calculus, we proceed by introducing its set of axioms and its set of inference rules. Because the calculus is a negative calculus, the set of axioms contains the empty clause as single element. The set of inference rules also contains only a single element: SLDE-resolution, i.e., SLD-resolution, where the equational system is built into the unification computation.

3.2.2 SLDE-Resolution

The inference rule SLDE-resolution can be used to compute the logical consequences of a set of definite clauses, which can be split into an equational system $\mathcal{E}$ and a set of definite clauses $\mathcal{K}$ which does not contain the equality symbol in the conclusion of a clause except within the axiom of reflexivity [GR86, Höl89a]. This condition is satisfied for the simple fluent calculus with $\mathcal{E} = \mathcal{E}_{AC1}$ and $\mathcal{K} = \mathcal{K}_A \cup \mathcal{K}_C$. The axioms $\mathcal{E}_{\approx}$ of equality are not explicitely needed in SLDE-resolution; they are built into the unification computation. The axiom of reflexivity must be kept, however, if $\mathcal{K}$ contains an equation $s \approx t$ in the body of some clause. This equation can only be resolved against the $X \approx X$.

Let $\mathsf{UP}_{\mathcal{E}}$ be an $\mathcal{E}$-unification procedure, C a new variant $H \leftarrow A_1 \wedge \ldots \wedge A_m$ of a clause in $\mathcal{K}$ and G the goal clause $\leftarrow B_1 \wedge \ldots \wedge B_n$. If H and an atom B_i, $1 \le i \le n$, are $\mathcal{E}$-unifiable with $\theta \in \mathsf{UP}_{\mathcal{E}}(H, B_i)$, then

$$\leftarrow (B_1 \wedge \ldots \wedge B_{i-1} \wedge A_1 \wedge \ldots \wedge A_m \ \wedge B_{i+1} \wedge \ldots \wedge B_n)\theta$$

is called *SLDE-resolvent* of C and G. The concepts of deduction and refutation can be *SLDE-resolvent*

defined for SLDE-resolution in the obvious way.

SLDE-resolution is sound if the used $\mathcal{E}$-unification procedure is sound. It is also complete if the used $\mathcal{E}$-unification procedure is complete. Moreover, the selection of the atom B_i in each SLDE-resolution step is don't care non-deterministic (see e.g. [Höl89b]). Table 3.1 shows an SLDE-refutation for the planning problem depicted in Figure 3.1. One should observe that all $\mathcal{E}$-unification problems which have to be solved within this refutation are either fluent matching or fluent unification problems.

3.2.3 Solving Conjunctive Planning Problems

Due to the soundness and completeness of SLDE-resolution we find that a conjunctive planning problem with initial state $\mathcal{I}$, goal state $\mathcal{G}$, and given set of actions has a solution P iff there exists an SLDE-refutation of

$$(\exists P)\ causes(\mathcal{I}^{-I}, P, \mathcal{G}^{-I})$$

with respect to the equational system $\mathcal{E}_{AC1}$ and the logic program $\mathcal{K}_A \cup \mathcal{K}_C$, where $\cdot^{-I}$ is the mapping from multisets to fluent terms introduced in the previous Section 2.4. In particular, Figure 3.2 shows the solution to Sussman's anomaly corresponding to the steps taken in Table 3.1.

3.2.4 Solving the Frame Problem

The technical frame problem is elegantly solved within the fluent calculus by mapping it onto the fluent matching and fluent unification problem. Returning to the refutation shown in Table 3.1 we observe that in the deduction from (3) to (4) the variable Z_1 is bound to $ontable(a) \circ ontable(b) \circ clear(b)$. This fluent term contains precisely those fluents which are unchanged by the action $unstack(c, a)$ applied in the initial state of Sussman's anomaly. More precisely, let

$$s = ontable(a) \circ ontable(b) \circ on(c, a) \circ clear(b) \circ clear(c) \circ empty$$

and

$$t = clear(c) \circ on(c, a) \circ empty,$$

then

$$\theta = \{Z_1 \mapsto ontable(a) \circ ontable(b) \circ clear(b)\}$$

is a most general $\mathcal{E}$-matcher for the $\mathcal{E}$-matching problem

$$\mathcal{E}_{AC1} \models (\exists Z_1)\, s \approx t \circ Z_1.$$

Consequently, $unstack(c, a)$ can be applied to s yielding

$$s_1 = ontable(a) \circ ontable(b) \circ clear(b) \circ clear(a) \circ holding(c).$$

This solution to the frame problem is ultimately linked to the fact that the fluents are represented as resources, i.e., that $\circ$ is a symbol which is associative, commutative, admits the unit element 1, but is not idempotent. One could be tempted to model situations

(1) $\leftarrow causes(ontable(a) \circ ontable(b) \circ on(c,a) \circ clear(b) \circ clear(c) \circ empty,$
$W,$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

(2) $\leftarrow action(P_1, V_1, Q_1) \wedge$
$P_1 \circ Z_1 \approx ontable(a) \circ ontable(b) \circ on(c,a) \circ clear(b) \circ clear(c) \circ empty \wedge$
$causes(Z_1 \circ Q_1, W_1, ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

(3) $\leftarrow clear(v_2) \circ on(v_2, w_2) \circ empty \circ Z_1 \approx$
$ontable(a) \circ ontable(b) \circ on(c,a) \circ clear(b) \circ clear(c) \circ empty \wedge$
$causes(Z_1 \circ holding(V_2) \circ clear(W_2),$
$W_1,$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

(4) $\leftarrow causes(ontable(a) \circ ontable(b) \circ clear(b) \circ clear(a) \circ holding(c),$
$W_1,$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) empty).$

$\vdots$

(7) $\leftarrow causes(ontable(a) \circ ontable(b) \circ clear(b) \circ clear(a) \circ clear(c) \circ$
$ontable(c) \circ empty,$
$W_4,$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

$\vdots$

(10) $\leftarrow causes(ontable(a) \circ clear(c) \circ ontable(c) \circ clear(a) \circ holding(b),$
$W_7,$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

$\vdots$

(13) $\leftarrow causes(ontable(a) \circ ontable(c) \circ clear(a) \circ on(b,c) \circ clear(b) \circ empty,$
$W_{10},$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

$\vdots$

(16) $\leftarrow causes(ontable(c) \circ on(b,c) \circ clear(b) \circ holding(a),$
$W_{13},$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

$\vdots$

(19) $\leftarrow causes(ontable(c) \circ on(b,c) \circ clear(a) \circ on(a,b) \circ empty,$
$W_{16},$
$ontable(c) \circ on(b,c) \circ on(a,b) \circ clear(a) \circ empty).$

(20) $[\,]$

Table 3.1: Solving Sussman's anomaly by SLDE-resolution. Atoms with predicate symbol *action* are given first priority in the selection process. Atoms with the equality symbol are selected next. (2) is the SLDE-resolvent of (1) and the second rule for *causes*. (3) is the SLDE-resolent of (2) and the fact representing the action *unstack* . (4) is the SLDE-resolvent of (3) and the axiom of reflexivity. Following the fourth goal clause only every third goal clause is shown. The selected actions are in this sequence: *putdown*, *pickup*, *stack*, *pickup*, *stack*. One should observe that the variable W is bound to the list (3.1) by this refutation.

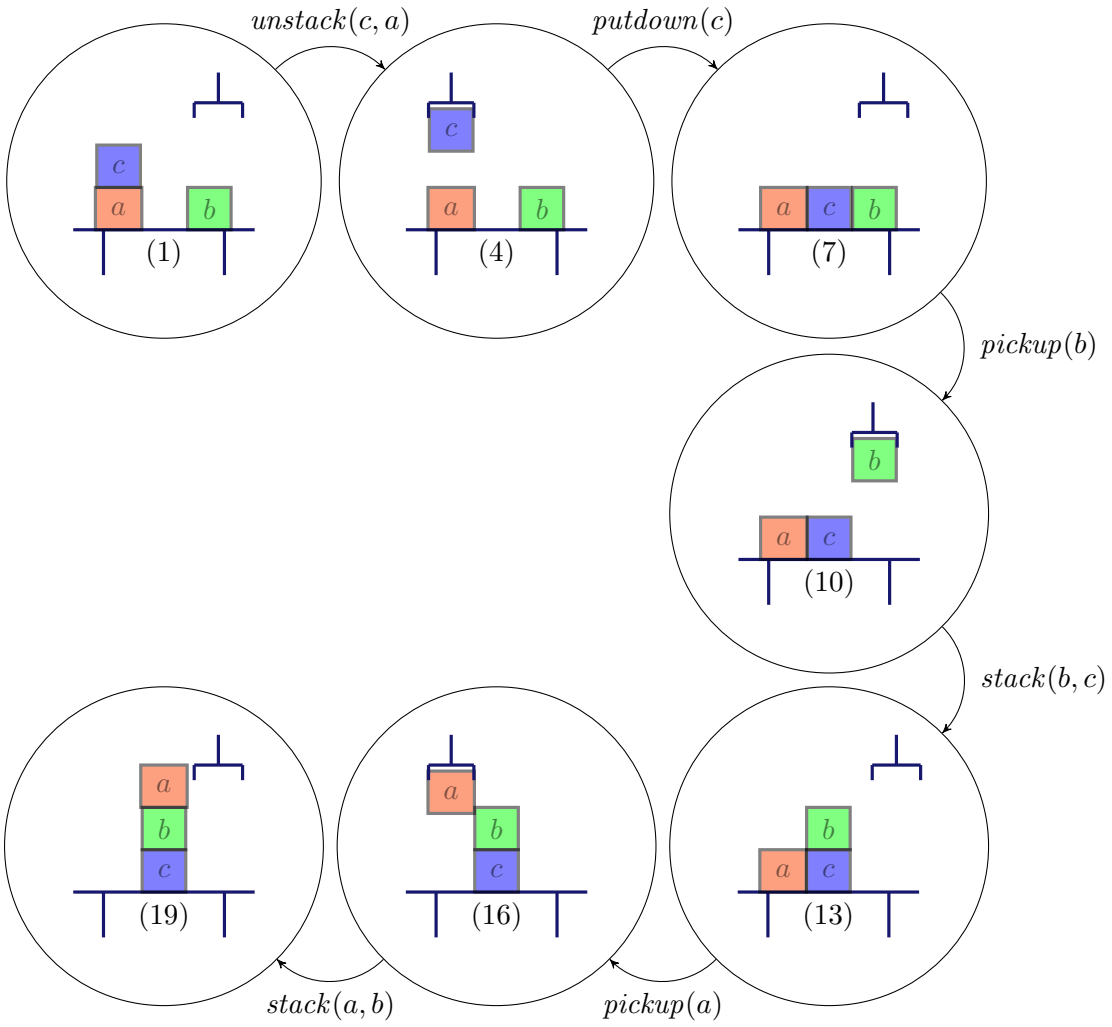


Figure 3.2: The execution of plan (3.1) to solve Sussman's anomaly. The numbers under the table indicate the correspondence between the situation shown in the circle and the respective step in the SLDE-resolution proof shown in Table 3.1.

as sets of fluents. In other words, one would not only require that $\circ$ is associative, commutative, and admits the unit element 1, but is also idempotent, i.e. satisfies the law *idempotent*

$$X \circ X \approx X. \tag{3.2}$$

Let $\mathcal{E}_{ACI1} = \mathcal{E}_{AC1} \cup \{(3.2)\}$. But now the $\mathcal{E}$-matching problem $\mathcal{E}_{ACI1}$

$$\mathcal{E}_{ACI1} \models (\exists Z_1)\, s \approx t \circ Z_1$$

has not only θ as a solution but

$$\eta = \{Z_1 \mapsto \mathit{ontable}(a) \circ \mathit{ontable}(b) \circ \mathit{clear}(b) \circ \mathit{empty}\}$$

is a solution as well. Moreover, θ and η are incomparable with respect to $\mathcal{E}_{ACI1}$. In this case the binding generated for Z_1 does not only represent those fluents which remain unchanged. Computing the successor state in this case yields

$$s_2 = \mathit{ontable}(a) \circ \mathit{ontable}(b) \circ \mathit{clear}(b) \circ \mathit{clear}(a) \circ \mathit{holding}(c) \circ \mathit{empty}$$

which is not the intended result as the arm of a robot cannot be holding a block and be empty at the same time.

3.2.5 Remarks

The technical frame problem has received much attention in the literature (see e.g. [Hay73, Bro87, Rei91]). Some people even believed that it cannot be solved within first order logic (see e.g. [HM86]). The solution presented in this chapter is discussed in detail in [Höl92]

In this section a forward planner was presented, i.e. a procedure which applies actions to the initial state until the goal state is reached. Equally well a backward planner could have been presented, i.e. a procedure which is applied to the goal state and reasons backwards until the initial state is obtained.

In the examples presented so far the initial state was always completely specified. This need not to be the case. For example, we could be interested in the question of what else is needed besides a block b lying on the table in order to build a tower as in the goal state of Sussman's anomaly, i.e. we would like to know whether

$$(\exists X, P, Y)\, \mathit{causes}(\mathit{ontable}(b) \circ Y, P, \mathit{ontable}(c) \circ \mathit{on}(b,c) \circ \mathit{on}(a,b) \circ \mathit{clear}(a) \circ \mathit{empty} \circ X)$$

is a logical consequence of $\mathcal{F}_A \cup \mathcal{F}_C \cup \mathcal{E}_{AC1}$. This problem can also be solved by using SLDE-resolution.

Actions may have indeterminate effects. For example, if we flip a coin then we do not know in advance the outcome of this action. The coin may be either heads or tails. This can be expressed with the help of an additional binary function symbol $|$ which is associative, commutative, and admits a unit element 0. Depending on the domain $|$ may be idempotent as well. Additionally some distributivity laws involving $|$ and $\circ$ have to be satisfied in such cases.

Common sense reasoning tells us that a robot arm cannot hold an object and be empty at the same instant. However, this information is not available to a computer unless we

explicitly state that it is a contradiction. In the fluent calculus, consistency constraints concerning fluent terms can be formulated and added to the clauses as conditions [HS90].

The simple fluent calculus presented in this chapter is equivalent to the multiplicative fragment of linear logic and to the linear connection method [GHS96]. It has been extended in many ways including solutions to the ramification and the qualification problem (see e.g. []), for hierarchical planning problems, for parallel planning problems, or planning problems involving specificity.

There are versions of the fluent calculus, where constraints on fluent terms allow fluents to appear at most once in a fluent. In this case, the fluent calculus becomes quite similar to modern versions of the situation calculus, which has led to a unified calculus for reasoning about actions and causality. However, in doing so the relation to linear logic and the linear connection method is lost.

Chapter 4

Deduction, Abduction, and Induction

Until now we were concerned with the logical consequences of a set of formulas. More formally, we were investigating a relation $\models$ between a set $\mathcal{K}$ of formulas and a single formula F, i.e.

$$\mathcal{K} \models F.$$

So far, $\mathcal{K}$ was given and F was either unknown or given. In the former case we were asking for the logical consequences of $\mathcal{K}$ whereas in the latter case we were testing whether the given formula F was indeed a logical consequence of $\mathcal{K}$. The process of computing or testing the logical consequences of a given set of formulas within a calculus is called *deduction*. However, there are problems which cannot be solved by deduction.

deduction

Consider the case where the knowledge base $\mathcal{K}$ of a mobile robot consists of the following rules:

- If the grass is wet then the wheels are wet ($g \rightarrow w$).
- If the sprinkler is running then the grass is wet ($s \rightarrow g$).
- If it is raining then the grass is wet ($r \rightarrow g$).

Furthermore, assume that the robot observes that its wheels are wet (w). Being curious it would like to know whether this observation follows from what it already knows about the world. However, $\mathcal{K} \not\models w$. Being unsatisfied with this finding the robot would like to explain the observed fact. What shall it do?

If the robot is rational[1] then it is aware of the fact that it does not know everything. In other words, it is aware that its knowledge base is incomplete. One attempt to explain the observed fact w is to look for a fact p such that

$$\mathcal{K} \cup \{p\} \models w$$

and $\mathcal{K} \cup \{p\}$ is consistent. There are several possibilities in the example scenario:

1. If $p \equiv w$, then this is really no new information.

[1] For a discussion of rational agents see [RN95].

2. If $p \equiv g$, then the robot knows that the grass is wet, but it does not know the reason for the grass being wet.

3. If $p \equiv s$ or $p \equiv r$ then the robot can deduce that the grass is wet.

In any case we say that p has been *abduced* and the process of finding such an abduced fact is called *abduction*. In practical applications the number of atoms that may be abduced, i.e. the so-called *abducibles*, is restricted. In our example, the number of abducibles may be the set $\{s, r\}$, in which case only the third possibility arises.

abduction
abducible

The notion of abduction was introduced by the philosopher Peirce (see [HW32]), who identified three forms of reasoning:

- *Deduction*, an analytic process based on the application of general rules to particular cases, with the inference as a result.
- *Abduction*, synthetic reasoning which infers a case (or a fact) from the rules and the result.
- *Induction*, synthetic reasoning which infers a rule from the case and the result.

deduction
abduction
induction

4.1 Deduction

So far, all reasoning processes considered in this book have all been deductions. Hence, there is not much to say at this point except for the following. In the previous chapters we have assumed that the logic is unsorted. Equivalently, all variables had only one sort, viz. terms. Likewise, function symbols were mappings from (the n-fold cross-product of) the set of terms into the set of terms and relation symbols were subsets of (the n-fold cross-product of) the set of terms. As shown in the following subsection, sorts can easily be introduced and do not raise the expressive power of a first-order language.

4.1.1 Sorts

In common sense reasoning, computer science, and many applications sorts play an important role. A statement like

every doggy is an animal

sounds natural, whereas a statement like

every object in the domain that is a doggy is also an animal

sounds somewhat awkward. Already in 1885 the philosopher Pierce has suggested to annotate quantified variables with so-called *sorts* denoting sets of objects.

As another and more formal example suppose we are computing with natural numbers and want to express that addition is commutative. This can be directly specified in first order logic by the formula

$$(\forall X, Y)\,(\mathsf{number}(X) \wedge \mathsf{number}(Y) \rightarrow \mathit{plus}(X, Y) = \mathit{plus}(Y, X)), \tag{4.1}$$

where number is a unary predicate denoting natural numbers and *plus* is a binary predicate denoting addition. For the moment we are not concerned in how number and *plus* are defined; this will be discussed in detail in Section **??**. A closer look at formula (4.1) leads to several observations:

- The formalization itself looks lengthy and clumsy.
- The sort information concerning natural numbers is encoded in a unary predicate.
- The unary predicate restricts the possible bindings for the variables X and Y.

The drawback of the first observation can be removed by writing

$$(\forall X, Y : number)\, plus(X, Y) = plus(Y, X), \tag{4.2}$$

where $X, Y : number$ specifies that the variables X and Y are of sort *number*. As will be shown in this subsection sort information can be expressed in terms of unary predicates and a formula like (4.2) may be seen as a short hand notation for formula (4.1). Moreover, building the unary predicates denoting sort information into the deductive machinery may result in more efficient computations.

Formally, a *first order language with sorts* is a first order language together with a function

$$sort : \mathcal{V} \to 2^{\mathcal{R}_S},$$

where $\mathcal{R}_S \subseteq \mathcal{R}$ is a finite set of unary (or monadic) predicate symbols called *base sorts*. A *sort* s is a set of base sorts, i.e., $s \in 2^{\mathcal{R}_S}$. $\emptyset \in 2^{\mathcal{R}_S}$ is called *top* sort. Usually, variables are annotated by their sort and we write

$\mathcal{R}_S$
sort
top sort

$$X : s$$

if $sort(X) = s$. Finally, we assume that for every sort s there are countably many variables $X : s$. According to these definitions, formula (4.2) is a well-formed formula of a first order logic with sort *number*.

To assign a meaning to sorted formulas we extend the notion of an interpretation I to sorts. Let $\mathcal{D}$ be the domain of I. I maps each sort

$$s = \{p_1, \ldots, p_n\}$$

to

$$s^I = \mathcal{D} \cap p_1^I \cap \ldots \cap p_n^I,$$

where $p_j^I \subseteq \mathcal{D}$ is the interpretation of p_j wrt I, $1 \leq j \leq n$. A variable assignment $\mathcal{Z}$ is said to be *sorted* iff for all variables $X : s$ we find that

sorted variable assignment

$$X^{\mathcal{Z}} \in s^I.$$

There is a subtlety involved with this definition. Because sorts may denote empty sets, a sorted variable assignment is only a partial mapping and it is not clear at all what is meant by an application of a sorted variable assignment to a term which contains the occurrence of a variable with empty sort. To avoid this problem we assume in the sequel that sorts are non-empty. Under these conditions sorted variable assignments are total and the application of a sorted variable assignment to a term is defined as usual.

Now let I be an interpretation and $\mathcal{Z}$ a sorted variable assignment with respect to I. The meaning of a formula F in a sorted language under I and $\mathcal{Z}$, in symbols $F^{I,\mathcal{Z}}$, is defined inductively as follows:

$$\begin{aligned}
[p(t_1,\ldots,t_n)]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad (t_1^{I,\mathcal{Z}},\ldots,t_n^{I,\mathcal{Z}})) \in p^I.\\
[\neg F]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad F^{I,\mathcal{Z}} = \bot.\\
[F_1 \wedge F_2]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad F_1^{I,\mathcal{Z}} = \top \text{ and } F_2^{I,\mathcal{Z}} = \top.\\
[F_1 \vee F_2]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad F_1^{I,\mathcal{Z}} = \top \text{ or } F_2^{I,\mathcal{Z}} = \top.\\
[F_1 \rightarrow F_2]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad F_1^{I,\mathcal{Z}} = \bot \text{ or } F_2^{I,\mathcal{Z}} = \top.\\
[F_1 \leftrightarrow F_2]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad [F_1 \rightarrow F_2]^{I,\mathcal{Z}} = \top \text{ and } [F_2 \rightarrow F_1]^{I,\mathcal{Z}} = \top.\\
[(\exists X : s)\, F]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad \text{there exists } d \in s^I \text{ such that } F^{I,\{X\mapsto d\}\mathcal{Z}} = \top.\\
[(\forall X : s)\, F]^{I,\mathcal{Z}} = \top \quad &\text{iff} \quad \text{for all } d \in s^I \text{ we find } F^{I,\{X\mapsto d\}\mathcal{Z}} = \top.
\end{aligned}$$

One should observe that each interpretation I maps the top sort to its domain $\mathcal{D}$. Hence, variables with top sort are interpreted as standard variables. In this sense the first order language with sorts seems to be a generalization of the standard first order language. However, each valid formula in a sorted first order language can be transformed to a valid formula in an unsorted first order language and vice versa with the help of a so-called *relativization function* *relativization function* rel.

$$\begin{aligned}
rel(p(t_1,\ldots,t_n)) &= p(t_1,\ldots,t_n)\\
rel(\neg F) &= \neg rel(F)\\
rel(F_1 \wedge F_2) &= rel(F_1) \wedge rel(F_2)\\
rel(F_1 \vee F_2) &= rel(F_1) \vee rel(F_2)\\
rel(F_1 \rightarrow F_2) &= rel(F_1) \rightarrow rel(F_2)\\
rel(F_1 \leftrightarrow F_2) &= rel(F_1) \leftrightarrow rel(F_2)\\
rel((\forall X : \mathtt{s})\, F) &= (\forall Y)\,(p_1(Y) \wedge \ldots \wedge p_n(Y) \rightarrow rel(F\{X \mapsto Y\}))\\
&\quad \text{if } sort(X) = s = \{p_1,\ldots,p_n\} \text{ and } Y \text{ is a new variable}\\
rel((\exists X : s)\, F) &= (\exists Y)\,(p_1(Y) \wedge \ldots \wedge p_n(Y) \wedge rel(F\{X \mapsto Y\}))\\
&\quad \text{if } sort(X) = s = \{p_1,\ldots,p_n\} \text{ and } Y \text{ is a new variable}
\end{aligned}$$

Thus, the expressive power of sorted and unsorted first order languages is identical. However, in a calculus, where the sort information has been built into the deductive machinery, computations may be considerable faster (see [Wei96]).

So far, we have shown how variables can be sorted by means of a function $sort$. In the sequel it will be shown that sorting of variables suffices to sort function and relation symbols in the presence of the axioms of equality.

The underlying idea is quite simple and will be illustrated by two examples. Suppose the knowledge base $\mathcal{K}$ contains the axioms of equality. Furthermore, suppose that $\mathcal{K}$ contains the fact

$$p(t_1,\ldots,t_n),$$

where $t_1,\ldots,t_n$ are terms. Then this fact can be equivalently replaced by

$$(\forall X_1 \ldots X_n)\,(p(X_1,\ldots,X_n) \leftarrow X_1 \approx t_1 \wedge \ldots \wedge X_n \approx t_n)$$

using the axiom of substitutivity, where $X_1,\ldots,X_n$ are new variables. Likewise, if $\mathcal{K}$ contains the atom

$$A\lceil f(t_1,\ldots,t_n)\rceil,$$

then this atom can be equivalently replaced by

$$(\forall X_1 \ldots X_n)\,(A\lceil f(t_1,\ldots,t_n)/f(X_1,\ldots,X_n)\rceil \leftarrow X_1 \approx t_1 \wedge \ldots \wedge X_n \approx t_n).$$

Using a straightforward generalization of these two replacement techniques each formula F can be transformed into an equivalent formula F', in which

- all arguments of function and relation symbols different from $\approx$ are variables and
- all equations are of the form $t_1 \approx t_2$ or $f(X_1,\ldots,X_n) \approx t$, where $X_1,\ldots,X_n$ are variables and t, t_1, and t_2 are variables or constants.

Sorting the variables occurring in F' effectively sorts the function and relation symbols.

A formula like the abovementioned F' is usually quite lengthy and cumbersome to read if compared to the original formula F. To ease the notation we will stay with F but add so-called *sort declarations* to sort variables, function and relation symbols. If $sort(X) = s$ then the sort declaration for the variable X is *sort declarations*

$$X : s$$

as before. Let s_i, $1 \leq i \leq n$, and s be sorts, f an n-ary function and p an n-ary relation symbol. Then

$$f : s_1 \times \ldots \times s_n \rightarrow s$$

and

$$p : s_1 \times \ldots \times s_n$$

are sort declarations for f and p, respectively.

4.2 Abduction

In many real situations observations are made that cannot immediately be explained. For example, if the car is not starting in the morning after the driver has turned the key then this observation cannot be explained with respect to the normal behavior of a car. A car should be built such that the engine is supposed to start as soon as the key is turned. However, if the engine is not running then this surprising behavior needs to be explained. For example, the driver checks the battery. If he finds that the battery is empty then this new fact may explain the observation that the car is not running.

Abduction consists of computing explanations for observations. It has many applications. The introductory example is taken from fault diagnosis. A specification describes a normal behavior of a system and abduction has to identify parts of the system which are not normal to explain a fault. In medical diagnosis, for example, the symptoms are the observations which have to be explained. In high level vision the camera yields a partial descriptions of objects in a scene and abduction is used to identify the objects. Sentences in natural language are often ambiguous and abductive explanations correspond to the various interpretations of such sentences. Planning problems can be viewed as abductive problems as well. The generated plan is the explanation for reaching the goal state. In knowledge assimilation the assimilation of a new datum can be performed by adding to the knowledge base an abduced fact that explaines the observed new datum.

4.2.1 Abduction in Logic

Given a set of formulas $\mathcal{K}$ and a formula G, abduction consists – to a first approximation
explanation – of finding a set of atoms $\mathcal{F}'$, called *explanation* such that

- $\mathcal{K} \cup \mathcal{K}' \models G$ and
- $\mathcal{K} \cup \mathcal{K}'$ is satisfiable.

The elements of $\mathcal{K}'$ are said to be *abduced*.

One should note that abducing only sets of atoms is no real restriction as atoms can be used to name formulas. For example, suppose we want to abduce the formula

$$(\forall X)\,(bird(X) \rightarrow fly(X))$$

then we may name this formula by means of an atom $birdsFly(X)$, add to $\mathcal{K}$ the clause

$$(\forall X\,(birdsFly(X) \rightarrow (bird(X) \rightarrow fly(X)))$$

and abduce $birdsFly(X)$ instead.

However, the characterization of abduction given so far is too weak. First of all, we need to distinguish abduction from induction. Moreover, as shown in the introductory example of this chapter, it allows us to explain the observation that the grass is wet by the fact that the grass is wet. We need to restrict $\mathcal{K}'$ such that it conveys some reason why the observation holds. We do not want to explain one effect in terms of another effect, but only in terms of some cause. For both reasons, explanations are often restricted to belong to a
abducibles special class of pre-specified and domain-dependent atoms called *abducibles*. We assume that such a set is given. For example, if $\mathcal{K}$ is a logic program, then the set of abducibles is typically the set of predicates for which there is no definition in $\mathcal{K}$, where r is *defined in* $\mathcal{K}$ iff $\mathcal{K}$ contains a definite clause with r being the relation symbol occurring in the head of the clause (i.e. the only positive literal occurring in the clause).

There may be additional criteria for restricting the number of possible candidates for explanations.

basic explanation

- An explanation should be *basic* in the sense that it cannot be explained by another explanation. Returning to the example shown in the beginning of this chapter, the explanation g (grass is wet) for the observation w (wheels are wet) is not basic because it can be explained by either s (sprinkler was running) or r (it was raining). On the other hand, both s and r are basic explanations.

minimal explanation

- An explanation should be *minimal* in that it cannot be subsumed by another explanation. For example, let

  $$\mathcal{F} = \{p \leftarrow q,\ p \leftarrow q \wedge r\}$$

  and

  $$G = p.$$

  Then the explanation

  $$\{q, r\}$$

  is not minimal because it is subsumed by the explanation

  $$\{q\}.$$

- Additional information can help to discriminate among different explanations. For example, an explanation may be rejected if some of its logical consequences are not observed. Let us return to the introductory example of this chapter. It is raining (r) and the sprinkler is running (s) are possible explanations for the observation that the wheels are wet (w). Suppose the knowledge base contains an additional clause stating that if it is raining, then there are clouds (c).

  $$r \rightarrow c.$$

  Now, if no clouds are observed, then the explanation r should be rejected.

- Domain-dependent preference criteria may be applied to (partially) order the set of possible explanations. Again, in the introductory example of this chapter we could choose to prefer explanations which we are able to change. Therefore, because we cannot change the fact that it is raining (r), but we can change the fact that the sprinkler is running (s), the explanation s would be preferred.

- So-called *integrity constraints* can be defined which have to be satisfied by the explanations.

integrity constraints

The concept of integrity constraints first arose in the field of databases. An integrity constraint is simply a formula. The basic idea is that states of a database are only acceptable iff the integrity constraints are satisfied in these states. This can be directly applied to abduction in that explanations are only acceptable iff the integrity constraints are satisfied.

abductive framework

Formally, an *abductive framework* $\langle \mathcal{K}, \mathcal{K}_A, \mathcal{K}_{IC} \rangle$ consists of a set $\mathcal{K}$ of formulas, a set $\mathcal{K}_A$ of ground atoms called *abducibles* and a set of integrity constraints $\mathcal{K}_{IC}$. Given an observation G, G is *explained* by $\mathcal{K}'$ iff

- $\mathcal{K}' \subseteq \mathcal{K}_A$,
- $\mathcal{K} \cup \mathcal{K}' \models G$ and
- $\mathcal{K} \cup \mathcal{K}'$ satisfies $\mathcal{K}_{IC}$.

There are several ways to define what it means that $\mathcal{K} \cup \mathcal{K}'$ *satisfies* $\mathcal{K}_{IC}$. The *satisfiability view* requires that

satisfiability view

$$\mathcal{K} \cup \mathcal{K}' \textit{ satisfies } \mathcal{K}_{IC} \text{ iff } \mathcal{K} \cup \mathcal{K}' \cup \mathcal{K}_{IC} \text{ are satisfiable.}$$

The stronger *theoremhood view* requires that

theoremhood view

$$\mathcal{K} \cup \mathcal{K}' \textit{ satisfies } \mathcal{K}_{IC} \text{ iff } \mathcal{K} \cup \mathcal{K}' \models \mathcal{K}_{IC}.$$

In the next two sections, several applications of abduction in knowledge assimilation and theory revision are discussed. Thereafter, abduction is related to model generation, thereby showing how abducibles can be effectively computed.

4.2.2 Knowledge Assimilation

Knowledge assimilation is the process of assimilating new knowledge into a given knowledge base. Rather than presenting an overview of knowledge assimilation we will show how abduction can be used to assimilate knowledge by an example.

Let the knowledge base be defined as the following logic program, where we assume that all clauses are universally closed.

$$\begin{array}{rcl} \mathcal{K} & = & \{sibling(X,Y) \leftarrow parents(Z,X) \wedge parents(Z,Y), \\ & & \ parents(X,Y) \leftarrow father(X,Y), \\ & & \ parents(X,Y) \leftarrow mother(X,Y), \\ & & \ father(john, \mathsf{mary}), \\ & & \ mother(jane, \mathsf{mary})\}. \end{array}$$

Viewed as a database, the predicates *father* and *mother* are extensionally defined, wheras the predicates *sibling* and *parents* are intensionally defined. Let the set of integrity constraints be defined as

$$\begin{array}{rcl} \mathcal{K}_{IC} & = & \{X \approx Y \leftarrow father(X,Z) \wedge father(Y,Z), \\ & & \ X \approx Y \leftarrow mother(X,Z) \wedge mother(Y,Z)\}, \end{array}$$

where $\approx$ is a 'built-in' binary relation symbol written infix. As usual the formulas in $\mathcal{K}_{IC}$ are assumed to be universally closed. In addition we assume that the axiom of reflexivity ($X \approx X$) holds and that $s \not\approx t$ holds for all distinct ground terms s and t. In other words, the integrity constraints state that an individual can only have one mother and one father. Furhermore, let the set of abducibles be

$$\mathcal{K}_A = \{A \mid A \text{ is a ground instance of } father(john, Y) \text{ or } mother(jane, Y)\}.$$

Suppose that we have to assimilate the observation that mary and bob are siblings, i.e.

$$sibling(\mathsf{mary}, \mathsf{bob}).$$

There are two minimal explanations, viz.

$$\{father(john, \mathsf{bob})\}$$

and

$$\{mother(jane, \mathsf{bob})\}.$$

Both explanations satisfy the integrity constraints with respect to the satisfiability view. However, if we additionally observe that

$$mother(joan, \mathsf{bob})$$

holds, then only the first explanation satisfies the integrity constraints.

The example also demonstrates that newly assimilated knowledge may lead to a revision of earlier assimilated knowledge. This is a non-monotonic form of reasoning also called *belief revision* and will be studied in Chapter 5. The following subsection contains another example of this kind.

belief revision

4.2.3 Theory Revision

In all real world situations we do not know everything. Rather we have to base our decisions on so-called rules of thumb which allow us to jump to conclusions if the world is *normal*. A typical example is the way we handle the flight schedule of an airline. If we look at the booklet containing the flight schedule of Lufthansa then we may find that there are flights from Dresden to Frankfurt at 6:30am, 11:30am, 2:30pm, 5:30pm and 9:30pm each day. Given this information almost everybody is willing to accept the conclusion that there is no flight from Dresden to Frankfurt at 8:00am. However, if we observe that there is as a matter of fact a flight at 8:00am from Dresden to Frankfurt, then we have to revise our theory.

In this section, a formalization of this kind of theory revision within an abductive framework is given. Again, the method will only be exemplified, this time by another famous example used quite frequently in the area of knowledge representation and reasoning. For a formal account of theory revision the reader is referred to [Poo88].

Let the knowledge base be the following universally closed set of formulas:

$$\begin{aligned}\mathcal{K} \;=\; \{&\mathit{penguin}(X) \rightarrow \mathit{bird}(X),\\ &\mathit{birdsFly}(X) \rightarrow (\mathit{bird}(X) \rightarrow \mathit{fly}(X)),\\ &\mathit{penguin}(X) \rightarrow \neg \mathit{fly}(X),\\ &\mathit{penguin}(\mathit{tweedy}),\\ &\mathit{bird}(\mathit{john})\}.\end{aligned}$$

Let the set of integrity constraints be empty and let the set of abducibles be

$$\mathcal{K}_A = \{A \mid A \text{ is a ground instance of } \mathit{birdsFly}(X)\}.$$

If we observe

$$\mathit{fly}(\mathit{john})$$

then this can be explained by the minimal set

$$\{\mathit{birdsFly}(\mathit{john})\}.$$

On the other hand,

$$\mathit{fly}(\mathit{tweedy})$$

cannot be explained at all, because the set

$$\mathcal{K} \cup \{\mathit{birdsFly}(\mathit{tweedy})\}$$

is unsatisfiable. Similarly, if we additionally learn that *john* is a penguin, i.e. if we add the fact $\mathit{penguin}(\mathit{john})$ to $\mathcal{K}$, then $\mathit{fly}(\mathit{john})$ cannot be explained and we have to revise our theory.

In this line of reasoning $\mathit{birdsFly}(X)$ can be seen as a kind of so-called *default* and $\mathit{fly}(\mathit{john})$ is explained by *default reasoning*. We are willing to accept such a default if it does not contradict with any other information that we have gained so far. *default reasoning*

Default reasoning is another important method within the area of knowledge representation and reasoning and will be studied in Chapter 5.

4.2.4 Abduction and Model Generation

As pointed out in [Kow91] there is a strong link between deduction and abduction. In fact, explanations for abductive problems can be computed by deduction. Consider the following knowledge base

$$\begin{aligned}\mathcal{K} \;=\; \{&\mathit{wobblyWheel} \leftrightarrow \mathit{brokenSpokes} \vee \mathit{flatTyre},\\ &\mathit{flatTyre} \leftrightarrow \mathit{picturedTube} \vee \mathit{leakyValve}\}\end{aligned}$$

which can be split into an if-part

$$\begin{aligned}\mathcal{K}_{\leftarrow} \;=\; \{&\mathit{wobblyWheel} \leftarrow \mathit{brokenSpokes},\\ &\mathit{wobblyWheel} \leftarrow \mathit{flatTyre},\\ &\mathit{flatTyre} \leftarrow \mathit{picturedTube},\\ &\mathit{flatTyre} \leftarrow \mathit{leakyValve}\}\end{aligned}$$

and an only-if-part

$$\begin{aligned}\mathcal{K}_{\rightarrow} \;=\; \{&\mathit{wobblyWheel} \rightarrow \mathit{brokenSpokes} \vee \mathit{flatTyre},\\ &\mathit{flatTyre} \rightarrow \mathit{picturedTube} \vee \mathit{leakyValve}\}.\end{aligned}$$

Let $\mathcal{K}_{IC}$ be the empty set and

$$\mathcal{K}_A = \{\mathit{brokenSpokes},\ \mathit{picturedTube},\ \mathit{leakyValve}\}$$

be the set of abducibles.

One should note that $\mathcal{K}_{\leftarrow}$ is a logic program and, hence, SLD-resolution can be used to derive answers for questions posed to $\mathcal{K}_{\leftarrow}$. Furthermore, all abducibles are not defined within $\mathcal{K}_{\leftarrow}$. This ensures that all abductions wrt the abductive framework $\langle \mathcal{K}_{\leftarrow}, \mathcal{K}_A, \mathcal{K}_{IC} \rangle$ will be basic.

Now consider the case that the observation *wobblyWheel* has been made and consider the abductive framework $\langle \mathcal{K}, \mathcal{K}_A, \mathcal{K}_{IC} \rangle$. There are three minimal and basic explanation, viz.

$$\begin{aligned}&\{\mathit{brokenSpokes}\},\\ &\{\mathit{picturedTube}\},\\ &\{\mathit{leakyValve}\}.\end{aligned}$$

These explanations can be obtained in two different ways, one using SLD-resolution and the other one using model generation.

- Turning to the first method, consider the abductive framework $\langle \mathcal{K}_{\leftarrow}, \mathcal{K}_A, \mathcal{K}_{IC} \rangle$. As soon as an observation like *wobblyWheel* has been made, the obvious way to proceed is to try to show whether the observation is already a logical consequence of the knowledge base. In case of logic programs like $\mathcal{K}_{\leftarrow}$ this is the case if an SLD-refutation of the query

  $$\leftarrow \mathit{wobblyWheel}$$

  wrt to $\mathcal{K}_{\leftarrow}$ can be found. Figure 4.1 shows the complete search space generated by SLD-resolution for this query. The search space is finite. At each branch there is a failing goal. The negation of each goal is a possible explanation of the observation *wobblyWheel* wrt $\langle \mathcal{K}_{\leftarrow}, \mathcal{K}_A, \emptyset \rangle$.

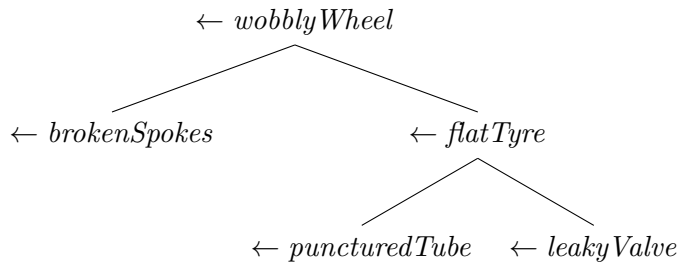


Figure 4.1: The search space generated by SLD-resolution for $\mathcal{K}_{\leftarrow} \cup \{\leftarrow wobblyWheel\}$.

- Turning to the second method and having observed *wobblyWheel*, we may add *wobblyWheel* to our knowledge base, which in this case is $\mathcal{K}_{\rightarrow}$. The minimal models of the extended knowledge base are

$$\{wobblyWheel,\ flatTyre,\ puncturedTube\},$$
$$\{wobblyWheel,\ flatTyre,\ leakyValve\}$$

and

$$\{wobblyWheel,\ brokenSpokes\}.$$

Restricting these models to the abducible predicates we obtain precisely the three explanations as in the first method.

In fact this duality between abduction and model generation can be exploited even in the case of non-propositional abducibles as shown in [CDT91].

4.2.5 Remarks

In the article [KKT93] an ehm.xcellent overview of abductive logic programming is given. It is shown that there is a close relation between various non-monotonic reasoning techniques used within knowledge representation and reasoning (see Chapter 5).

Abduction does not only apply to toy examples. In the autumn of 1997 Mercedes Benz experienced heavy losses when it was demonstrated by example that

$$\{babyBenz\} \not\models elchTest,$$

where the atom *babyBenz* denotes the specification of a car nicknamed Baby-Benz – todays *A class*) – and the atom *elchTest* denotes the specification of a certain driving maneuver, viz. driving around an elch which unexpectedly steps on the road. In these tests, the car overturned. After a lengthy abductive process Meredes-Benz demonstrated that after adding an electronic stability program *ESP* to the car, the Baby-Benz passed the driving maneuver, i.e.

$$\{babyBenz, esp\} \models elchTest.$$

4.3 Induction

As an introductory example for inductive reasoning consider the sorted equational system

$$\begin{aligned}\mathcal{K}_{plus} \quad = \quad & \{(\forall Y : number)\, plus(0, Y) \approx Y, \\ & (\forall X, Y : number)\, plus(s(X), Y) \approx s(plus(X, Y))\}\end{aligned}$$

which can be used to define addition (*plus*) on the natural numbers. Informally, each natural number is represented by either the constant 0 or by an application of the unary function symbol s (representing the successor function) to the representation of another natural number; a precise specification will be given in Section **??**. Given $\mathcal{K}_{plus}$ we would like to prove some properties of addition like the commutativity of *plus* , i.e.

$$(\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X).$$

Is this law a logical consequence of $\mathcal{K}_{plus}$? Unfortunately, it is not. This can be seen if we consider the following interpretation: Let

$$\mathcal{D} = \mathbb{N} \cup \{\diamondsuit\}$$

be the domain consisting of the natural numbers $\mathbb{N} = \{0, f(0), f(f(0)), \ldots\}$ extended by the additional object $\diamondsuit$. Let the interpretation I be such that

I	0	s	$plus$,
	0	f	$\otimes$,

where

$$f(d) \quad = \quad \begin{cases} f(0) & \text{if } d = \diamondsuit, \\ d + f(0) & \text{if } d \in \mathbb{N}, \end{cases}$$

$$d \oplus e \quad = \quad \begin{cases} 0 & \text{if } d = e = \diamondsuit, \\ \diamondsuit & \text{if } d = 0 \text{ and } e = \diamondsuit, \\ d & \text{if } d \in \mathbb{N}^+ \text{ and } e = \diamondsuit, \\ e & \text{if } d = \diamondsuit \text{ and } e \in \mathbb{N}, \\ d + e & \text{if } d, e \in \mathbb{N}, \end{cases}$$

$+ : \mathbb{N} \to \mathbb{N}$ is the usual addition on $\mathbb{N}$, and $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. It is easy to verify that

$$I \models \mathcal{K}_{plus}.$$

However,

$$I \not\models (\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X)$$

because

$$\diamondsuit \oplus 0 = 0 \neq \diamondsuit = 0 \oplus \diamondsuit.$$

Almost every student knows that addition is commutative from a freshman mathematics course. The student probably also still remembers how this can be formally proved: It can be shown by induction on either the first or the second argument of the definition of addition. The induction principle applied in this case is *Peanos induction principle*

induction

Peanos induction principle

$$(P(0) \wedge (\forall M : number)\, (P(M) \to P(s(M)))) \to (\forall M : number)\, P(M). \tag{4.3}$$

In other words, if a certain property P holds for 0 (the so-called *base case*) and we find that for all natural numbers M the property P holds for $s(M)$ given that it holds for

M (the so-called *step case*), then we may conclude that P for all natural numbers M. In our example, it is applied to the so-called *induction variable* X with

induction variable

$$P(X) \equiv (\forall Y : number)\, plus(X, Y) \approx plus(Y, X). \tag{4.4}$$

To prove the induction base, Peanos induction principle has to be applied recursively (see Table 4.1).

Thus, if we add to the knowledge base $\mathcal{K}_{plus}$ the two instances $\mathcal{K}_I$ of the induction principle (4.3) obtained by choosing P as in (4.4) and in (4.7), then we are able to show that addition is commutative, i.e.

$$\mathcal{K}_{plus} \cup \mathcal{K}_I \models (\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X).$$

To summarize, $\mathcal{K}_{plus}$ admits some interpretations which are *non-standard* in the sense that the domains and the functions over these domains do not correspond to the set of natural numbers and the functions usually defined on this set, respectively. By adding appropriate induction axioms to $\mathcal{K}_{plus}$ these non-standard interpretations are excluded. This process will be analyzed in more detail in this section.

non-standard interpretation

Mathematical induction is an essential proof technique used to verify statements about recursively defined objects like natural numbers, lists, trees, stacks, logic formulas etc. As another example consider propositional logic formulas. The use of structural induction to prove properties of such formulas is sanctioned by a corresponding induction theorem. Similar theorems can be proven for other recursively defined objects. Because recursively defined data structures appear almost everywhere, induction plays a central role in the fields of mathematics, algebra, logic, computer science, formal language theory, to mention just a few.

The example presented in the introduction of this section already illustrates the main questions that have to be answered if a property shall be proved by induction:

1. First of all, should induction be really used to prove a statement? There are other proof techniques like proof by contradiction or contraposition or by resolution, which are often simpler than induction. Very often only experience can tell which proof technique should be used.
2. Should the statement be generalized before an attempt is made to prove it by induction? Sometimes it is simply easier to prove a more general statement or property.
3. Which variable is to be the induction variable? This decision is often combined with the following two questions.
4. What induction principle is to be used?
5. What is the property used within the induction principle?
6. Should nested induction be taken into account? If we prove the base case and the induction step then the very same questions may again have to be answered.

In this section I will show how properties of recursively defined programs are verified by induction. Such programs are typically defined as functions operating on top of recursive data structures. Therefore, we start out to have a closer look at these structures.

To show that

$$(\forall Y : number)\, plus(0, Y) \approx plus(Y, 0) \tag{4.5}$$

holds, we observe that the first equation of $\mathcal{K}_{plus}$ can be applied to reduce the left-hand-side of (4.5) and we obtain the reduced problem of showing that

$$(\forall Y : number)\, Y \approx plus(Y, 0)$$

holds. By the law of symmetry this is equivalent to showing that

$$(\forall Y : number)\, plus(Y, 0) \approx Y \tag{4.6}$$

holds. The proof of (4.6) is by induction on Y with

$$P(Y) \equiv plus(Y, 0) \approx Y. \tag{4.7}$$

In the base case $P(0)$ we find that

$$\underline{plus(0, 0)} \to 0$$

using again the first equation in $\mathcal{K}_{plus}$ with matching substitution $\{Y \mapsto 0\}$. Hence,

$$P(0) \tag{4.8}$$

holds trivially. Turning to the induction step we assume that $P(n)$ holds, i.e.

$$plus(n, 0) \approx n, \tag{4.9}$$

where n is is the representation of an arbitrary but fixed natural number. Now consider the case $P(s(n))$: Here we find that

$$\underline{plus(s(n), 0)} \to s(\underline{plus(n, 0)}) \to s(n) \tag{4.10}$$

using the second equation occurring in $\mathcal{K}_{plus}$ with matching substitution $\{X \mapsto n,\ Y \mapsto 0\}$ in the first rewriting step and the induction hypothesis (4.9) in the second rewriting step. Thus, we conclude that

$$plus(s(n), 0) \approx s(plus(n, 0)) \approx s(n).$$

This shows that

$$(\forall X : number)\, (P(X) \to P(s(X))) \tag{4.11}$$

holds. Finally, applying modus ponens to the induction principle (4.3) using (4.8) and (4.11) yields the desired result.

Table 4.1: A mathematical proof by induction of $(\forall Y : number)\, plus(0, Y) \approx plus(Y, 0)$.

4.3.1 Data Structures

The functions used within a program are usually defined over some data structure. As already mentioned, commonly used data structures are natural numbers, lists, trees or logic formulas. Because we intend to model these data structures within a logical language, we have to designate certain terms to denote the elements of the data structures. Given an alphabet $\mathcal{A}$, let $\mathcal{A}_C \subseteq \mathcal{A}_F$ be a set of function symbols called *constructors* and $\mathcal{A}_D \subseteq \mathcal{A}_F$ be the set of *defined* function symbols, where we assume that $\mathcal{A}_C \cap \mathcal{A}_D = \emptyset$ and $\mathcal{A}_C \cup \mathcal{A}_D = \mathcal{A}_F$. Let $T(\mathcal{A})$ denote the set of terms that can be built from the symbols occurring in $\mathcal{A}$. The set $T(\mathcal{A}_C)$ is the set of *constructor ground terms*.

constructor
defined function symbols
constructor ground term

As examples consider the following three data structures:

- The data structure *number* can be defined by the nullary constructors

  number

  $$0 : number$$

  and the unary constructor

  $$s : number \to number.$$

  Informally, 0 represents the natural number $\emptyset$ and $s/1$ represents the successor function on natural numbers.

  $$T(\{0, s\}) = \{0,\ s(0),\ s(s(0)),\ \ldots\}$$

  is a set of constructor ground terms which is called the *sort* *number*.

- Similarly, the data structure *bool* can be defined by the two nullary constructors

  bool

  $$\langle\,\rangle : bool$$

  and

  $$[\,] : bool.$$

  The sort *bool* is

  $$T(\{\langle\,\rangle,\ [\,]\}) = \{\langle\,\rangle,\ [\,]\}.$$

- The data structure *list*(*number*) (list of natural numbers) can be defined by the nullary constructor

  list(*number*)

  $$[\,] : list(number)$$

  and the binary constructor

  $$:: number \times list(number) \to list(number).$$[2]

  The sort *list*(*number*) is

  $$\{[\,],\ [0],\ [0,0],\ [s(0)],\ \ldots\},$$

  where $[b_1,\ b_2,\ \ldots,\ b_n]$ is an abbreviation for $b_1 : (b_2 \ldots (b_n : []) \ldots)$.

[2] The symbol : is overloaded. Its first occurrence denotes a function symbol $:/2$, for which a sort declaration is given. Its second occurrence separates the function symbol from its sort declaration. Likewise, [] is used to denote the empty list in this item, whereas it was an element of *bool* in the previous item. Its intended denotion should always be obvious from the context.

It is enlightening to specify the data structure of propositional logic formulas and a function f from this set to *number* which counts the number of symbols occurring in a propositional logic formula.

As discussed in Subsection 4.1.1, sort information can be added to a logic without changing its expressive power. In this section I assume that all variables and function symbols are sorted. For example, the sort declaration

$$X : number$$

represents a variable of sort *number* and the sort declaration

$$p : number \to number$$

represents a unary function from *number* to *number*, which will later be used to denote the predecessor function on *number*. As shown in Sections **??** and **??** sort information can be expressed with the help of unary predicate symbols so that whenever a clause C contains a term t of sort q then the literal $\neg q(t)$ is added to C as an additional constraint. These constraints can be used to decide whether a term is well-sorted, where
well-sortedness well-sortedness is defined as follows: A term t is said to be *well-sorted* wrt to a set of sort declarations $\mathcal{S}$ iff

- t is a constant or a variable of some sort or
- t is of the form $f(t_1, \ldots, t_n)$, $\mathcal{S}$ contains a sort declaration $f : sort_1 \times \ldots \times sort_n \to sort$ and for all $1 \leq i \leq n$ we find that t_i is of sort $sort_i$. In this case $ft_1, \ldots, t_n)$ is of sort $sort$.

For example, the term

$$[0,\ s(0),\ s(s(0))]$$

is well-sorted with respect to the sorts $list(number)$ and *number*, whereas the term

$$s([\,])$$

is not well-sorted. One should also observe that the sort $list(number)$ just contains all
! well-sorted lists of natural numbers. In this section I will always assume that terms are well-sorted.

Returning to data structures we are now in the position to define structures like *number* or $list(number)$ but we are not yet able to access the elements of a data structure. Therefore, we additionally assume that for each constructor c/n, $n > 0$, there are n defined
selector function symbols $s_i/1$ called *selectors*, which applied to $c(t_1, \ldots, t_n)$ yield t_i. For example, the predecessor function $p/1$ is the selector for the only argument of $s/1$ in the sort *number*, i.e. $p/1$ is defined by the equation

$$p(s(n)) \approx n.$$

Formally, we require that the following conditions are satisfied by a data structures:

1. Different constructors denote different objects.
2. Constructors are injective.

3. Each object can be denoted as an application of some constructor to its selectors (if any exists).

4. Each selector is 'inverse' to the constructor it belongs to.

5. Each selector returns a so-called *witness term* if applied to a constructor it does not belong to (see below).

Because we intend to prove properties about data structures, each sort $sort$ is translated into a set of first order formulas $\mathcal{F}_S$ which satisfies the conditions mentioned above. For the data structure $number$ these conditions are satisfied by the following clauses:

$$\begin{aligned}\mathcal{F}_{\mathsf{number}} \;=\; \{&(\forall N : number)\, 0 \not\approx s(N),\\ &(\forall N, M : number)\, (s(N) \approx s(M) \rightarrow N \approx M),\\ &(\forall N : number)\, (N \approx 0 \vee N \approx s(p(N))],\\ &(\forall N : number)\, p(s(N)) \approx N,\\ &p(0) \approx 0\}.\end{aligned} \tag{4.12}$$

The first four clauses correspond directly to the first four conditions. Taking the fifth condition into consideration we observe that p is only a partial function with respect to the data structure $number$. For reasons given in the next subsection I like to deal with total functions. Any ground constructor term can be assigned to $p(0)$. One usually assigns constants to such terms, which are called *witness terms*. In the last clause of $\mathcal{F}_{\mathsf{number}}$ 0 has been assigned to $p(0)$ as witness term. *witness term*

This example concludes the presentation of data structures. Clauses similar to the one mentioned in (4.12) must be specified for each data structure or sort. I am now in a position to formally define functions over data structures.

4.3.2 Admissible Programs

Functions are defined over recursively specified data structures by means of structural induction. As an example consider again propositional logic formulas. A function over propositional logic formulas can be defined according to Theorem **??** which states the principle of structural recursion. Similar theorems can be proven for other data structures like $number$ or $list(number)$.

In this subsection functions are specified with the help of a set of conditional equations, i.e. universally closed equations of the form

$$l \approx r \leftarrow C$$

such that

$$var(C) \cup var(r) \subseteq var(l)$$

and C denotes a conjunction of equations and negated equations. I will use the notation shown in the following example, which defines the function $plus/2 \in \mathcal{A}_D$. $plus/2$ takes two numbers X and Y as arguments and yields a number:

$$\begin{aligned}\mathcal{F}_{plus} \;=\; \{&(\forall X, Y : number)\, (plus(X, Y) \approx Y \;\leftarrow\; X \approx 0),\\ &(\forall X, Y : number)\, (plus(X, Y) \approx s(plus(p(X), Y)) \;\leftarrow\; X \not\approx 0)\}.\end{aligned}$$

One should observe that the two conditions $X \approx 0$ and $X \not\approx 0$ are mutually exclusive. Similarly, we can define a less-than order ($\mathsf{lt}/2$) on *number* as a function which takes two numbers as arguments and returns a boolean:

$$\begin{array}{rcll} \mathcal{F}_{\mathsf{lt}} & = & \{(\forall X, Y : \mathit{number})\,(\mathsf{lt}(X,Y) \approx [\,] & \leftarrow \quad Y \approx 0), \\ & & (\forall X, Y : \mathit{number})\,(\mathsf{lt}(X,Y) \approx \langle\,\rangle & \leftarrow \quad X \approx 0 \wedge Y \not\approx 0), \\ & & (\forall X, Y : \mathit{number})\,(\mathsf{lt}(X,Y) \approx \mathsf{lt}(p(X), p(Y)) & \leftarrow \quad X \not\approx 0 \wedge Y \not\approx 0)\}. \end{array}$$

program One should observe that the conditions are again mutually exclusive. We will call a set of clauses consisting of data structure declarations and function definitions a *program*. For example, $\mathcal{F}_{\mathsf{number}} \cup \mathcal{F}_{plus}$ is a program.

Such a program $\mathcal{F}$ is said to be

well-formedness
- *well-formed* iff it can be ordered such that each function symbol occurring in the definition of a function g in $\mathcal{F}$ either is introduced before by a data structure declaration, or by another function definition, or it is g itself, in which case the function is said to be *recursive*;

well-sortedness
- *well-sorted* iff each term occurring in $\mathcal{F}$ is well-sorted;

determinism
- *deterministic* iff for each function definition occurring in $\mathcal{F}$ the defining cases are mutually exclusive;

condition-completeness
- *condition-complete* iff for each function definition of a function g/n occurring in $\mathcal{F}$ and each well-sorted n-tuple of constructor ground terms given as input to g/n there is at least one condition which is satisfied.

For example, the program $\mathcal{F} = \mathcal{F}_{\mathsf{number}} \cup \mathcal{F}_{plus}$ is well-formed, well-sorted, deterministic and condition-complete. The alert reader might have noted that the definition of $p/1$ in $\mathcal{F}_{\mathsf{number}}$ does not contain an explicit condition. The condition is implicitly contained in the left-hand-side of the equations because in the first equation the argument of $p/1$ must be of the form $s(X)$ and in the second equation it must be of the form 0. In fact, the final two elements of (4.12) can be equivalently replaced by the universally closed clauses

$$p(X) \approx N \leftarrow X \approx s(N)$$

and

$$p(X) \approx 0 \leftarrow X \approx 0$$

respectively. Because a well-sorted argument of $p/1$ can be either 0 (exclusively) or $s(X)$, $p/1$ is condition-complete.

rewriting Such a well-formed, well-sorted, deterministic and condition-complete program $\mathcal{F}$ is called by a well-sorted ground term t. t is *rewritten* (or *evaluated*) as follows. If t contains a subterm of the form $g(t_1, \ldots, t_n)$ such that each t_i, $1 \leq i \leq n$, is a constructor ground term, then find the rule

$$g(X_1, \ldots, X_n) \approx r \leftarrow C \in \mathcal{F}$$

such that $g(t_1, \ldots, t_n)$ and $g(X_1, \ldots, X_n)$ are unifiable with most general unifier θ and $\mathcal{F} \models C\theta$. In this case, replace $g(t_1, \ldots, t_n)$ by $r\theta$. One should observe that there

is exactly one such rule because $\mathcal{F}$ is condition-complete. Consequently, this rewrite relation is confluent. An example can be found in the following subsection.

A program is *terminating* iff there is no infinite rewriting sequence for any well-sorted ground term. Finally, a program is *admissible* iff it is well-formed, well-sorted, deterministic, condition-complete and terminating. Because $\mathcal{F}_{\mathsf{number}} \cup \mathcal{F}_{plus}$ is also terminating it is an admissible program.

terminating

admissible

In the sequel I will consider admissible programs. Given an admissible program $\mathcal{F}$ and ground term t as input to $\mathcal{F}$, we can now evaluate t.

4.3.3 Evaluation

For admissible programs $\mathcal{F}$ the rewrite relation defines a unique evaluator

$$eval_{\mathcal{F}} : T(\mathcal{A}_F) \to T(\mathcal{A}_C),$$

which maps all well-sorted ground terms to constructor ground terms. $eval_{\mathcal{F}}(t)$ is the normal form of t with respect to the rewrite relation defined in the previous subsection and is called the *value* of t. For example, the term

value

$$\underline{plus(s(0), s(0))}$$

is subsequently rewritten to

$$s(plus(\underline{p(s(0))}, s(0)))$$

and to

$$s(\underline{plus(0, s(0))})$$

and to

$$s(s(0)),$$

where I have underlined the subterm that was replaced. Hence, its value is $s(s(0))$. One should observe that $eval_{\mathcal{F}}$ would not be total for well-sorted ground terms if the function symbols defined in the program were not total. For example, if the clause

$$p(X) \approx 0 \leftarrow X \approx 0$$

is eliminated from $\mathcal{F}_{\mathsf{number}}$ then the well-sorted term $p(0)$ cannot be rewritten into a constructor ground term.

$eval_{\mathcal{F}}$ can also be viewed as an interpretation whose domain is the set of well-formed constructor ground terms. $eval_{\mathcal{F}}$ behaves as a Herbrand interpretation if applied to a well-sorted constructor ground term t, i.e.

$$eval_{\mathcal{F}}(t) = t,$$

and if applied to a well-sorted term s containing occurrences of defined function symbols, then it maps s to its unique value. $eval_{\mathcal{F}}$ is called the *standard interpretation* of the program $\mathcal{F}$.

standard interpretation

Let $\mathcal{F} = \mathcal{F}_{\mathsf{number}} \cup \mathcal{F}_{plus}$. It is easy to verify that the following relations hold:

$$\begin{aligned}
&eval_{\mathcal{F}} \models \mathcal{F},\\
&eval_{\mathcal{F}} \models (\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X),\\
&eval_{\mathcal{F}} \models (\forall X : number)\, X \not\approx s(X).
\end{aligned}$$

In other words, under the standard interpretation the addition over natural numbers is commutative and each number is different from its successor. We say that a formula F is *true* with respect to an admissible program $\mathcal{F}$ iff

$$eval_{\mathcal{F}} \models F.$$

The set

$$\{F \mid eval_{\mathcal{F}} \models F\}$$

theory of true statements is called the *theory* of the admissible program $\mathcal{F}$. Of course, we are interested in whether a given formula F belongs to the theory of a program. Because in general the theory of an admissible program is neither decidable nor semi-decidable,[3] the best we can hope for is to find sufficient condititions such that under these conditions F can be shown to belong to the theory.

Returning to the previous example we note that neither

$$\mathcal{F} \models (\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X)$$

nor

$$\mathcal{F} \models (\forall X : number)\, X \not\approx s(X)$$

because there are non-standard interpretations which are models for $\mathcal{F}$ but not for

$$(\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X)$$

or

$$(\forall X : number)\, X \not\approx s(X).$$

This can be demonstrated using a domain with an additional symbol, say $\diamondsuit$, as shown in the introduction of this section.

But we want to model natural numbers and the usual operations on natural numbers in a correct way. In particular, we want that the theorems about natural numbers can be obtained as logical consequences of the program. The approach taken here is to add additional clauses to the program $\mathcal{F}$ such that those non-standard interpretations, which caused the problems, are no longer models of $\mathcal{F}$. The additional clauses are induction axioms.

4.3.4 Induction Axioms

Let us assume that each admissible program $\mathcal{F}$ is associated with a decidable set $\mathcal{F}_I$ *induction axioms* of first order formulas called the *induction axioms* of $\mathcal{F}$. For the moment we shall only require that the standard interpretation models $\mathcal{F}_I$, i.e.

$$eval_{\mathcal{F}} \models \mathcal{F}_I.$$

For example, let $\mathcal{F} = \mathcal{F}_{\mathsf{number}} \cup \mathcal{F}_{plus}$ and $\mathcal{F}_I$ be the set of all formulas of the form

$$(P(0) \wedge (\forall X : number)\, (P(X) \rightarrow P(s(X)))) \rightarrow (\forall X : number)\, P(X), \tag{4.13}$$

[3] This follows from Gödel's incompleteness result (see Chapter **??**).

where $P(X)$ is any first order formula with X as the only free variable. Expression (4.13) is a scheme for an infinite set of induction axioms which are obtained by instantiating $P(X)$. For example, if $P(X)$ is replaced by

$$X \not\approx s(X)$$

then (4.13) becomes

$$(0 \not\approx s(0) \wedge (\forall X : number)\,(X \not\approx s(X) \rightarrow s(X) \not\approx s(s(X)))) \rightarrow (\forall X : number)\, X \not\approx s(X). \tag{4.14}$$

One should note that

$$eval_{\mathcal{F}} \models (4.14).$$

It can now be shown by any sound and complete calculus for first order logic that

$$\mathcal{F} \cup \{(4.14)\} \models (\forall X : number)\, X \not\approx s(X) \tag{4.15}$$

holds. One should observe that (4.15) holds if we can show that the condition of (4.14) holds. This condition is a conjunction. The first conjunct

$$0 \neq s(0)$$

is an immediate consequence of the first element of $\mathcal{F}_{\mathsf{number}}$ obtained by replacing N by 0. The second conjunct

$$(\forall X : number)\,(X \not\approx s(X) \rightarrow s(X) \not\approx s(s(X))]$$

follows from the second element in $\mathcal{F}_{\mathsf{number}}$). In a similar manner it can be shown that

$$(\forall X, Y : number)\, plus(X, Y) \approx plus(Y, X)$$

is a logical consequence of $\mathcal{F}$ and appropriate instances of (4.14).

4.3.5 Remarks

In order to show semantically that

$$eval_{\mathcal{F}} \models (\forall X : number)\, X \not\approx s(X)$$

we have to replace X by each element d from the sort *number* and show that

$$eval_{\mathcal{F}} \models_{\{X/d\}} X \not\approx s(X).$$

Because the sort *number* is infinite, the number of proofs to be given is infinite. Using the induction axiom (4.14) instead, the proof is finite.

We cannot expect to find inductive axioms such that all formulas in the theory of admissible programs can be proved. This is an immediate consequence of Gödel's first incompleteness result [Göd31]. Theorem proving by induction is incomplete, i.e. there are true statements about an admissible program which cannot be deduced.

Because the data structures used in programs are often inductively defined, the computation of induction axioms may be based on the definition of the data structures and

the functions. Heuristics may be applied to guide the selection of the induction variable, the induction schema and the induction axiom within an attempt to show that a certain formula in the theory of an admissible program.

Mathematical induction has been investigated within computer science for almost 30 years [Bur69]. Several automated theorem provers based on this principle have been developed over the years, among which the systems NQTHM [BM88], OYSTER-CLAM [BvHHS90] and INKA [HS96] are the most advanced. An excellent overview can be found in [Wal94].

In some cases it is unnessary to explicitely use induction axioms to prove inductive statements. Rather a generalization of the Knuth-Bendix completion procedure presented in Section 2.3.3 suffices. This technique is known as *inductionless induction* or *proof by consistency* (see [KM87]).

Chapter 5

Non-Monotonic Reasoning

Common sense reasoning is non-monotonic in general. But what precisely is a non-monotonic logic? What is a non-monotonic reasoning system? What is our intuition about non-monotonic reasoning? These and other questions are discussed in this section. Various non-monotonic reasoning systems are presented. It will turn out that there is no general agreement on how to model common sense reasoning; instead there is a whole family of systems.

In Section 5.1 an introduction to non-monotonic reasoning is given by discussing the so-called qualification problem, which arises in reasoning about situations, actions and causality. The closed world assumption is discussed in Section 5.2. In Section 5.3 the completion semantics is presented together with its application in logic programming. In particular, it is shown that the completion semantics is captured by the negation as failure inference rule. Thereafter, circumscription and default logic are introduced in Sections 5.4 and 5.5 respectively. Finally, answer set computing is presented in Section 5.6.

5.1 Introduction

Propositional, first order and equational logic are monotonic, i.e., the addition of new knowledge to a knowledge base does not invalidate previously drawn logical consequences. In However, many common sense reasoning scenarios are non-monotonic. Adding new tuples to a data base or making a new observation may invalidate previously drawn logical consequences.

monotonicity

A striking example demonstrating the need for non-monotonic behavior was presented by John McCarthy in [McC90], where he discussed the *missionaries and cannibals* puzzle.

missionaries and cannibals

> *Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten. How shall they cross the river?*

The alert reader can easily solve this problem. For example, considering states as triples comprising the number of missionaries, cannibals and boats on the starting bank of the

river, the sequence

$$(331,\ 220,\ 321,\ 300,\ 311,\ 110,\ 221,\ 020,\ 031,\ 010,\ 021,\ 000)$$

presents one solution (see e.g. [Ama71]). But can this solution be derived as a logical consequence of a first order formalization of the puzzle? This is apparently not the case for two reasons:

- First, many properties of boats, missionaries or cannibals, or the fact that rowing across the river does not change the number of missionaries or cannibals have not been stated. These properties and facts follow from common sense knowledge. Although there is the problem of specifying the relevant aspects of common sense knowledge we assume for the moment that the common sense properties and facts relevant for the missionaries and cannibals puzzle are given as first order sentences.

- The second reason is much deeper. This is best illustrated by quoting [McC90]:

  Imagine giving someone the problem, and after he puzzles for a while, he suggests going upstream half a mile and crossing on a bridge. "What bridge," you say. "No bridge is mentioned in the statement of the problem." And this dunce replies, "Well, you don't say there isn't a bridge." You look at the English and even at the translation of the English into first order logic, and you must admit that "they don't say" there is no bridge. So you modify the problem to exclude bridges and pose it again, and the dunce proposes a helicopter, and after you exclude that, he proposes a winged horse or that the others hang onto the outside of the boat while two row.

  You know see that while a dunce, he is an inventive dunce. Despairing of getting him to accept the problem in the proper puzzler's spirit, you tell him the solution. To your further annoyance, he attacks your solution on the grounds that the boat might have a leak or lack oars. After you rectify that omission from the statement of the problem, he suggests that a sea monster may swim up the river and may swallow the boat. Again you are frustated, and you look for a mode of reasoning that will settle his hash once and for all.

  But how shall this form of reasoning look like? We cannot simply state that there is no other way to cross the river than by boat and that nothing can go wrong with the boat. There are infinitely many such facts. Moreover, a human does not need such an ad hoc narrowing of the problem.

The second problem can be solved if we allow statements like

> *unless it can be deduced that an object is present, we conjecture that it is not present*

and

> *unless there is something wrong with the boat or something else prevents the boat from using it, it can be used to cross the river*.

Whereas the first statement allows us to exclude bridges and helicopters, the second allows us to conclude that the boat can in fact be used for crossing the river. Informally, these statements may be regarded as "rules of thumb".

One should observe that if we alter the puzzle by adding a sentence about a nearby bridge, then the first statement can no longer be used to infer that no bridge is present. Likewise, if we add a sentence about missing oars, then the second statement (in conjunction with the relevant facts of the encoded common sense knowledge) can no longer be used to infer that the boat can be used to cross the river. In other words, previously drawn logical consequences become invalid after new knowledge has been added to the knowledge base.

non-monotonic logics

Formally, a logic $\langle \mathcal{A}, \mathcal{L}, \models \rangle$ is said to be *non-monotonic* iff there exist $\mathcal{F}$, $\mathcal{F}'$ and G such that

$$\mathcal{F} \models G \quad \text{and} \quad \mathcal{F} \cup \mathcal{F}' \not\models G,$$

where $\mathcal{F}$ and $\mathcal{F}'$ are sets of formulas in $\mathcal{L}$ and G is a formula in $\mathcal{L}$.

In the sequel I will define various non-monotonic logics, show how statements like *unless it can be deduced* or *unless there is something wrong* can be encoded in these logics and discuss their main properties, strengths and weaknesses. I start out with logics based on the closed world assumption.

5.2 Closed World Assumption

open world assumption

closed world assumption

The closed world assumption (CWA) has been proposed by Reiter in [Rei77] in an attempt to model databases in a formal logic. Queries to databases can be answered in two ways. Under the so-called *open world assumption*, the only answers given to a query are those that can be obtained from proofs of the query, given the database, i.e., the answers are logical consequences of the database. Whereas under the so-called *closed world assumption* certain additional answers are admitted as a result of a failure to prove a result, i.e., a failure to prove that the answers are logical consequences.

5.2.1 An Example

Reconsider the database with the relation lectures presented in Section **??**. From a logical point of view, this relation is simply a set of atoms, viz.

$$\begin{aligned}\mathcal{F} \;=\; \{&\mathit{lectures}(\mathit{steffen}, \mathit{cl001}),\ \mathit{lectures}(\mathit{steffen}, \mathit{cl005}),\ \mathit{lectures}(\mathit{michael}, \mathit{cl002}),\\ &\mathit{lectures}(\mathit{heiko}, \mathit{cl004}),\ \mathit{lectures}(\mathit{horst}, \mathit{cl003}),\ \mathit{lectures}(\mathit{michael}, \mathit{cl005})\}.\end{aligned}$$

Under the open world assumption, queries are evaluated in the usual way for a first order logic. Hence, queries like

$$(\exists X)\, \mathit{lectures}(\mathit{steffen}, X) \tag{5.1}$$

are answered positively with X bound to *cl001* or *cl005*. On the other hand, queries like

$$\neg \mathit{lectures}(\mathit{michael}, \mathit{cl006}) \tag{5.2}$$

cannot be answered at all, because some models of $\mathcal{F}$ satisfy (5.2), whereas others do not.

Under the closed world assumption the evaluation of the query (5.1) leads to the same answers as under the open world assumption. However, the query (5.2) is answered positively. The positive answer is obtained as a result of attempting to show that

$$\mathit{lectures}(\mathit{michael}, \mathit{cl006}) \tag{5.3}$$

is a logical consequence of $\mathcal{F}$. This, however, is not the case. Moreover, the search space is finite. Because (5.3) is answered negatively, the closed world assumption allows the conclusion that its negation (5.2) is answered positively.

Evaluating a database under the closed world assumption is a quite natural thing to do. Students typically evaluate the course program of a semester under the closed world assumption. If a lecture is not shown in the program then most students are willing to conclude that this lecture is not given. The closed world assumption leads to a non-monotonic behavior of the reasoning system, because the announcement of an additional course may invalidate some of the conclusions previously drawn. For example, if the fact (5.3) is added to $\mathcal{F}$ then the query (5.2) will be answered negatively.

5.2.2 The Formal Theory

Let $\langle \mathcal{A}, \mathcal{L}, \models \rangle$ be a first order logic. First recall that the theory of a satisfiable set $\mathcal{F}$ of formulas is defined as

$$\mathcal{T}(\mathcal{F}) = \{G \mid \mathcal{F} \models G\}.$$

In other words, the theory of $\mathcal{F}$ contains $\mathcal{F}$ and all logical consequences of $\mathcal{F}$. Now let

$$\overline{\mathcal{F}} = \{\neg A \mid A \text{ is a ground atom in } \mathcal{L} \text{ and } \mathcal{F} \not\models A\}$$

$\mathcal{T}_{\mathrm{CWA}}(\mathcal{F})$ The *theory of* $\mathcal{F}$ *under the closed world assumption*, $\mathcal{T}_{CWA}(\mathcal{F})$, is defined as

$$\mathcal{T}_{CWA}(\mathcal{F}) = \mathcal{T}(\mathcal{F} \cup \overline{\mathcal{F}}).$$

Returning to our example we recall that

$$\mathcal{F} \not\models \mathit{lectures}(\mathit{michael}, \mathit{cl006})$$

and hence

$$\neg \mathit{lectures}(\mathit{michael}, \mathit{cl006}) \in \overline{\mathcal{F}}.$$

I have mentioned on several occasions that the definition of the logical consequence relation for first order theories is the standard one but that for certain applications, other logical consequence relations may better serve our purposes. The theory of a set of formulas under the closed world assumption can alternatively be defined by a new logical
$\models_{\mathrm{CWA}}/2$ consequence relation $\models_{CWA}/2$. Formally, for a first order logic $\langle \mathcal{A}, \mathcal{L}, \models \rangle$ we define $\langle \mathcal{A}, \mathcal{L}, \models_{CWA} \rangle$ as follows. Let

- $M_0 = \mathcal{T}(\mathcal{F}) \cup \overline{\mathcal{F}}$,
- $M_{i+1} = \{H \mid \text{there exists } G \in M_i \text{ such that } \mathcal{F} \cup \{G\} \models H\}$ for all $i \geq 0$ and
- $M = \bigcup_{i \geq 0} M_i$.

$\mathcal{F} \models_{CWA} G$ iff $G \in M$. It is an easy exercise to show that the following theorem holds.

Theorem 5.1 $\mathcal{T}_{\mathrm{CWA}}(\mathcal{F}) = \{G \mid \mathcal{F} \models_{\mathrm{CWA}} G\}$.

There is also a straightforward way of building the closed world assumption into a first order calculus $\langle \mathcal{A}, \mathcal{L}, \mathcal{F}_A, \vdash \rangle$, where $\mathcal{A}$ denotes the alphabet, $\mathcal{L}$ the language, $\mathcal{F}_A$ the set of axioms and $\vdash/2$ the inference relation. All we have to do is to extend the set of inference rules by adding the rule

$$\text{if } \nvdash A \text{ then conclude } \neg A,$$

where A is a ground atom in $\mathcal{L}$.[1]

5.2.3 Satisfiability

Whenever we extend a satisfiable set of formulas, we have to ensure that the new extended set is also satisfiable because otherwise any formula would be a logical consequence of this set.[2] We checked for this condition in the abductive framework presented in Section **??** and it is necessary to check it here also.

An example may help to clarify the situation in the case of reasoning under the closed world assumption. Let

$$\mathcal{F} = \{leakyValve \vee puncturedTube\}$$

Then,

$$\mathcal{F} \not\models leakyValve$$

and

$$\mathcal{F} \not\models puncturedTube.$$

Recall that $\overline{\mathcal{F}}$ contains all ground literals $\neg A$, where A is not a logical consequence of $\mathcal{F}$. Hence, we find that

$$\{\neg leakyValve, \neg puncturedTube\} \subseteq \overline{\mathcal{F}}.$$

As a result we find that

$$\mathcal{F} \cup \overline{\mathcal{F}} \supseteq \{leakyValve \vee puncturedTube,\ \neg leakyValve, \neg puncturedTube\}$$

is unsatisfiable. In other words, the theory of a satisfiable set of formulas under the closed world assumption may be unsatisfiable. However, there is a large class of formulas for which this theory is satisfiable:

Theorem 5.2 *Let $\mathcal{F}$ be a satisfiable set of formulas. $\mathcal{T}_{\mathrm{CWA}}(\mathcal{F})$ is satisfiable iff $\mathcal{F}$ admits a least Herbrand model.*

The proof is left to the reader as an exercise.

---

[1] This rule is in fact a meta-rule since it considers $\vdash/2$ as argument.

[2] If $\mathcal{F}$ is unsatisfiable, then $\mathcal{F} \cup \{\neg G\}$ is also unsatisfiable, indifferent to what G is. Therefore, in this case we know that $\mathcal{F} \models G$ for any formula.

5.2.4 Models and the Closed World Assumption

To gain a better understanding of the closed world assumption we will have a closer look at what happens to the set of models of a set $\mathcal{F}$ of formulas while reasoning under the the closed world assumption.

Let $\mathcal{F}$ be a set of first order formulas and $M = (D, I)$ and $M' = (D', I')$ be two models of $\mathcal{F}$, where D as well as D' are non-empty domains and I as well as I' are *submodel* interpretations. M is said to be a *submodel* of M' with respect to a set P of predicate symbols, in symbols $M \preceq_P M'$, iff the following conditions hold:

- $D = D'$ and
- I and I' are identical except that for all $q \in P$ we find $q_I \subseteq q_{I'}$.

If $P = \mathcal{A}_R$ then we write $M \preceq M'$ instead of $M \preceq_P M'$.

minimal model A model M of $\mathcal{F}$ is said to be *minimal* iff for all models M' of $\mathcal{F}$ we find that

$$M' \preceq M \text{ implies } M = M'$$

least model holds. Finally, a model M of $\mathcal{F}$ is said to be the *least model* of $\mathcal{F}$ iff for all models M' of $\mathcal{F}$ we find that

$$M \neq M' \text{ implies } M \prec M'$$

holds, where $M \prec M'$ iff $M \preceq M'$ and $M \neq M'$.

To exemplify these definitions we consider Herbrand interpretations. I.e., let the domain of an interpretation be the Herbrand universe and the assignment to predicate symbols be subsets of the Herbrand base. For example, let $\mathcal{A}_F = \{tweedy/0,\ john/0\}$, $\mathcal{A}_R = \{penguin/1,\ bird/1\}$ and

$$\mathcal{F} = \{penguin(tweedy),\ (\forall X)\,(penguin(X) \rightarrow bird(X))\}.$$

$\mathcal{F}$ has three Herbrand models, viz.

$$\begin{array}{lcl} M_1 & = & \{penguin(tweedy),\ bird(tweedy)\}, \\ M_2 & = & \{penguin(tweedy),\ bird(tweedy),\ bird(john)\} \text{ and} \\ M_3 & = & \{penguin(tweedy),\ bird(tweedy),\ bird(john),\ penguin(john)\}, \end{array}$$

with $M_1 \prec M_2 \prec M_3$. We conclude that

$$\mathcal{F} \not\models bird(john)$$

and

$$\mathcal{F} \not\models penguin(john).$$

Consequently,

$$\overline{\mathcal{F}} = \{\neg bird(john), \neg penguin(john)\}.$$

It is easy to check that M_2 and M_3 are not models of $\mathcal{F} \cup \overline{\mathcal{F}}$, whereas M_1 is a model of $\mathcal{F} \cup \overline{\mathcal{F}}$. In fact, it is the only Herbrand model of $\mathcal{F} \cup \overline{\mathcal{F}}$. In other words, the closed world assumption eliminates non-least models.

5.2.5 Remarks

Because first order logic is undecidable, the relation $\mathcal{F} \not\models A$ used in the definition of $\overline{\mathcal{F}}$ cannot be decided. This indicates that there are considerable difficulties in computing the theory of a set of formulas under the closed world assumption.

Renaming of formulas affects the theory of a set of formulas under the closed world assumption. For example, if we rename a predicate p occurring in a set $\mathcal{F}$ of formulas by $\neg q$ to obtain $\mathcal{F}'$, then $\mathcal{T}_{CWA}(\mathcal{F}) \neq \mathcal{T}_{CWA}(\mathcal{F}')$.

Consider the following set of formulas:

$$\begin{array}{rcll} \mathcal{F} & = & \{ & bird(tweedy), \\ & & & (\forall X)\,(bird(X) \wedge \neg ab(X) \rightarrow fly(X)) \quad \}. \end{array} \tag{5.4}$$

The second formula in $\mathcal{F}$ expresses that unless there is something wrong with a bird we are willing to conclude that the bird flies. In other words, this formula states the rule of thumb that birds normally fly. $\mathcal{F}$ is not a Horn set and does not admit a least Herbrand model. Hence, $\mathcal{T}_{CWA}(\mathcal{F})$ is unsatisfiable. Recall that the closed world assumption minimizes the sets p_I assigned to the predicate symbols p occurring in $\mathcal{F}$ by interpretations I. In the example, we do not really want to minimize birds or flying objects; instead we just like to minimize abnormalities. With this idea in mind we could apply the closed word assumption only to ground atoms of the form $ab(t)$. This idea works out for the example, but does not work in general.

There are several extensions to the closed world assumption which have been developed to overcome some of its limitations. Examples are the so-called *generalized closed world assumption* [Min82] or *extended closed world assumption* [GPP89].

Implementations, Unique name assumption as special case, further extensions

The closed world assumption is the basis for several further developments of non-monotonic reasoning such as predicate completion and negation as failure, which are presented in detail in the following Section 5.3. Virtually all proposals for non-monotonic reasoning are concerned with minimizing models. As a rule of thumb, we may state that non-monotonic reasoning is reasoning with respect to the minimal and/or least models of a set of formulas.

!

5.3 Completion

In the previous section, we have seen that a non-monotonic behavior can be achieved if certain assumptions are added to the knowledge base. In the case of the closed world assumption, only negative ground atoms are added to the knowledge base. In this section I present a method which allows to add more complex formulas.

5.3.1 An Example

As an introductory example, let $\mathcal{A}_F = \{tweedy/0, john/0\}$, $\mathcal{A}_R = \{penguin/1\}$ and

$$\mathcal{F} = \{penguin(tweedy)\}.$$

As discussed at the end of Section 5.2, non-monotonic reasoning can be regarded as reasoning in the minimal models of $\mathcal{F}$. In our example there are two models, viz.

$$M_1 = \{penguin(tweedy)\}$$

and

$$M_2 = \{penguin(tweedy),\ penguin(john)\}$$

with $M_1 \prec M_2$. The minimal model M_1 can be computed as follows: The formula

$$penguin(tweedy)$$

is replaced by the equivalent formula

$$(\forall X)\,(X \approx tweedy \rightarrow penguin(X)). \tag{5.5}$$

This formula is regarded as the "if" half of a definition of $penguin/1$. One way to exclude models which satisfy $penguin(john)$ is to extend this formula by adding the "only-if" half:

$$(\forall X)\,(X \approx tweedy \leftarrow penguin(X)). \tag{5.6}$$

completion This extension is called (*predicate*) *completion* and the formula (5.6) is called the *completion formula* of (5.5).

Let

$$\mathcal{F} = \{penguin(tweedy),\ penguin(john)\}.$$

The "if" half of the definition of $penguin$ is now of the form

$$(\forall X)\,(X \approx tweedy \vee X \approx john \rightarrow penguin(X))$$

and is completed by its completion formula

$$(\forall X)\,(X \approx tweedy \vee X \approx john \leftarrow penguin(X)).$$

In general, if a predicate is defined by a set of atoms, then the completion is identical to the closed world assumption. However, the two approaches differ as soon as more complex formulas are considered.

As an example consider the formula

$$(\forall X)\,(\neg fly(X) \rightarrow fly(X)) \tag{5.7}$$

which is equivalent to $(\forall X)\,fly(X)$. Extending (5.7) by its completion formula

$$(\forall X)\,(\neg fly(X) \leftarrow fly(X))$$

we obtain the unsatisfiable formula

$$(\forall X)\,(\neg fly(X) \leftrightarrow fly(X))$$

This example demonstrates that, as with the closed world assumption, we must expect satisfiability problems when computing the completion of a predicate. Hence, the question arises of whether there exists a class of formulas for which completion is guaranteed to yield satisfiable sets of formulas.

`Input:` A set $\mathcal{F}$ of clauses and a predicate symbol p/m.
`Output:` The completion formula $C_{\mathcal{F},p}$ of $\mathcal{F}$ with respect to p.

1. Replace each clause of the form $\{\neg L_1, \ldots, \neg L_n, p(t_1, \ldots, t_m)\}$ occurring in $\mathcal{F}$ by
$$L_1 \wedge \ldots \wedge L_n \rightarrow p(t_1, \ldots, t_m). \tag{5.8}$$
2. Replace each clause of the form (5.8) occurring in $\mathcal{F}$ by
$$(\forall \overline{X})(\exists \overline{Y})\,(X_1 \approx t_1 \wedge \ldots \wedge X_m \approx t_m \wedge L_1 \wedge \ldots \wedge L_n \rightarrow p(\overline{X})], \tag{5.9}$$
where $\overline{X} = X_1, \ldots, X_m$ is a sequence of 'new' variables and $\overline{Y}$ is a sequence of those variables which occur in (5.8).
3. Let
$$\{(\forall \overline{X})\,(C_i \rightarrow p(\overline{X})) \mid 1 \leq i \leq k\}$$
be the set of clauses having the form (5.9). Return the completion formula
$$C_{\mathcal{F},p} = (\forall \overline{X})\,(C_1 \vee \ldots \vee C_k \leftarrow p(\overline{X})).$$

Table 5.1: The completion algorithm computing the completion formula with respect to the predicate symbol p for a given set $\mathcal{F}$ of clauses.

5.3.2 The Completion

I turn now to the specification of a completion algorithm which computes the completion formula for a given set of clauses with respect to a given predicate. Before doing so however, I characterize certain sets of clauses as being solitary in a certain predicate symbol. It will turn out that the completion of a set of clauses with respect to a predicate symbol is satisfiable if the set is solitary with respect to the predicate symbol.

An occurrence of a predicate symbol p/n in a clause C is said to be

- *positive* iff we find terms t_i, $1 \leq i \leq n$, such that $p(t_1, \ldots, t_n) \in C$ and — *positive occurrence*
- *negative* iff we find terms t_i, $1 \leq i \leq n$, such that $\neg p(t_1, \ldots, t_n) \in C$. — *negative occurrence*

A set $\mathcal{F}$ of clauses is said to be *solitary* with respect to the predicate symbol p/n iff for each clause $C \in \mathcal{F}$ we find that if C contains a positive occurrence of p/n then C does not contain another occurrence of p/n. For example, the clause — *solitary set*

$$\{\neg \mathit{fly}(\mathit{tweedy}),\ \neg \mathit{fly}(\mathit{john}),\ \mathit{penguin}(\mathit{tweedy}),\ \neg \mathit{penguin}(\mathit{john})\}$$

is solitary in $\mathit{fly}/1$, but not solitary in $\mathit{penguin}/1$.

Table 5.1 defines a completion algorithm. Initialized with a set $\mathcal{F}$ of clauses and a predicate symbol p/m, it returns the completion formula $C_{\mathcal{F},p}$. One should observe that the clauses considered in the first step of this algorithm may contain several positive occurrences of p/m, in which case the algorithm is assumed to choose one of these occurrences arbitrarily unless otherwise specified.

As an example consider the following set $\mathcal{F}$ of clauses:

$$\begin{array}{rcll} \mathcal{F} & = & \{ & \neg penguin(Y) \vee bird(Y), \\ & & & bird(tweedy), \\ & & & \neg penguin(john) \qquad \}. \end{array} \tag{5.10}$$

Suppose we are calling the completion algorithm with $\mathcal{F}$ and $bird/1$. Then, after the first step $\mathcal{F}$ is replaced by

$$\begin{array}{rcll} \mathcal{F}_1 & = & \{ & penguin(Y) \rightarrow bird(Y), \\ & & & bird(tweedy), \\ & & & \neg penguin(john) \qquad \}. \end{array}$$

After the second step we obtain

$$\begin{array}{rcll} \mathcal{F}_2 & = & \{ & (\forall X)(\exists Y)\,(X \approx Y \wedge penguin(Y) \rightarrow bird(X)), \\ & & & (\forall X)\,(X \approx tweedy \rightarrow bird(X)), \\ & & & \neg penguin(john) \qquad \}. \end{array}$$

One should observe that the two clauses in $\mathcal{F}_2$ which define $bird/1$ are equivalent to

$$(\forall X)\,((\exists Y)\,(X \approx Y \wedge penguin(Y)) \vee X \approx tweedy \rightarrow bird(X)).$$

Finally, in the third step the completion algorithm returns

$$C_{\mathcal{F},bird} \quad = \quad (\forall X)\,((\exists Y)\,(X \approx Y \wedge penguin(Y) \vee X \approx tweedy \leftarrow bird(X)). \tag{5.11}$$

Because the completion formula contains occurrences of the equality symbol, we have to specify the equational theory under which these symbols are to be interpreted. Recall that one reason for computing with the completion is that it should be possible to derive negative facts. Hence, inequalities must be stated, along with equalities. The equational theory $\mathcal{F}_C$ shown in Table 5.2 was introduced by Clark in [Cla78] and serves this purpose. It consists of five schemes:

- The first scheme tells us that different function symbols (including constants) denote different data constructors.
- The second scheme corresponds to the occurs check in unification under the empty equational theory.
- The third scheme tells us that two complex terms are diffirent as soon as one pair of corresponding arguments is different.
- The fourth formula is the axiom of reflexivity and tells us that objects are equal if they are syntactically equal.
- The fifth scheme tells us that constructed objects are equal if they are constructed from equal components by applying the same constructor.
- The sixth scheme tells us that predicates applied to equal components have the same truth value.

$$
\begin{array}{rcl}
\mathcal{F}_C & = & \{(\forall \overline{X}, \overline{Y})\, f(\overline{X}) \not\approx g(\overline{Y}) \mid \\
 & & \quad \text{for each pair } f/n,\ g/m \text{ of different function symbols occurring in } \mathcal{A}_F\} \\
 & \cup & \\
 & & \{(\forall X)\, t[X] \not\approx X \mid \text{for each term } t \text{ which is different from } X \\
 & & \quad \text{but contains an occurrence of } X\} \\
 & \cup & \\
 & & \{(\forall \overline{X}, \overline{Y})\, (\bigvee_{i=1}^{n} X_i \not\approx Y_i \rightarrow f(\overline{X}) \not\approx f(\overline{Y})) \mid \\
 & & \quad \text{for each function symbol } f/n \text{ occurring in } \mathcal{A}_F\} \\
 & \cup & \\
 & & \{(\forall X)\, X \approx X\} \\
 & \cup & \\
 & & \{(\forall \overline{X}, \overline{Y})\, (\bigwedge_{x=1}^{n}\ X_i \approx Y_i \rightarrow f(\overline{X}) \approx f(\overline{Y})) \mid \\
 & & \quad \text{for each function symbol } f/n \text{ occurring in } \mathcal{A}_F,\} \\
 & \cup & \\
 & & \{\forall\, (\bigwedge_{x=1}^{n}\ X_i \approx Y_i \wedge p(\overline{X}) \rightarrow p(\overline{Y})) \mid \\
 & & \quad \text{for each predicate symbol } p/n \text{ occurring in } \mathcal{A}_F\}
\end{array}
$$

Table 5.2: The equational system $\mathcal{F}_C$ for predicate completion consisting of five schemes, where $\overline{X}$ and $\overline{Y}$ denote the sequences $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ of variables respectively and $t[X]$ denotes a term t which contains an occurrence of the variable X.

We can now formally define predicate completion: Let $\mathcal{F}$ be a set of formulas, which is solitary in p/n. The *predicate completion* $\mathcal{T}_C(\mathcal{F}, p)$ of p is defined as $\mathcal{T}_C(\mathcal{F}, p)$

$$\mathcal{T}_C(\mathcal{F}, p) = \{G \mid \mathcal{F} \cup C_{\mathcal{F},p} \cup \mathcal{F}_C \models G\}.$$

Theorem 5.3 *Let $\mathcal{F}$ be a set of formulas which is solitary in p/n. If $\mathcal{F}$ is satisfiable, then so is $\mathcal{T}_C(\mathcal{F}, p)$.*

Returning to the knowledge base $\mathcal{F}$ specified in (5.10) and the completion of $bird/1$ as computed in (5.10) we now find that

$$\neg bird(john) \in \mathcal{T}_C(\mathcal{F}, bird). \tag{5.12}$$

Predicate completion is non-monotonic which can be demonstrated by adding the fact

$$bird(john)$$

to $\mathcal{F}$. Now, (5.12) no longer holds.

Reasoning with the completion of p/n with respect to a knowledge base $\mathcal{F}$ amounts to nothing more than computing in the minimal models of $\mathcal{F}$ with respect to p. In this respect, predicate completion is similar to the closed world assumption. However, reasoning with the completion may lead to different results from those obtained when reasoning under the closed world assumption.

5.3.3 Parallel Completion

As in Section 5.2 we are not just interested in minimizing the extension of a single predicate symbol, but in minimizing the extension of several predicate symbols in parallel. This,

however, may lead to some complications as the following example demonstrates. Let

$$\begin{aligned} \mathcal{F} \;=\; \{ \quad & \mathit{bird}(\mathit{tweedy}), \\ & (\forall X)\,(\mathit{bird}(X) \wedge \neg \mathit{ab}(X) \rightarrow \mathit{fly}(X)) \quad \}. \end{aligned} \tag{5.13}$$

Informally, the second formula in $\mathcal{F}$ states that birds normally fly. Intuitively, we would like to minimize the models of $\mathcal{F}$ with respect to abnormalities and flying objects. However, completing $\mathit{ab}/1$ and $\mathit{fly}/1$ in parallel leads to a cyclic definition between the two relations. We simply cannot use the second formula occurring in $\mathcal{F}$ to define both $\mathit{ab}/1$ and $\mathit{fly}/1$.

Who is going to decide in cases like $\mathcal{F}$ above which relation is defined and which one is not? This question cannot be answered easily if $\mathcal{F}$ is an arbitrary knowedge base. However, there is an easy answer if $\mathcal{F}$ is a logic program. In this case the user has made the decision.

5.3.4 Parallel Completion and Logic Programming

normal logic programs

A *normal logic program* is a set $\mathcal{F}$ of clauses of the form

$$p(\bar{t}) \leftarrow A_1 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge \ldots \wedge \neg A_n \tag{5.14}$$

where p/m is a predicate symbol, $\bar{t}$ is a sequence $t_1, \ldots, t_m$ of terms and $A_i,\ 1 \leq i \leq n$, are atoms over some first order alphabet $\mathcal{A}$. Likewise, a *normal query* is a clause of the form

$$\leftarrow A_1 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge \ldots \wedge \neg A_n. \tag{5.15}$$

defined predicate symbol $\mathcal{A}_D$

Given a normal logic program $\mathcal{F}$, a predicate symbol p/m is said to be *defined* in $\mathcal{F}$ iff $\mathcal{F}$ contains a clause of the form shown in (5.14). Let $\mathcal{A}_D$ denote the set of defined predicate symbols in a logic program $\mathcal{F}$.

For example, reconsider the set $\mathcal{F}$ of formulas specified in (5.13). $\mathcal{F}$ is a normal logic program and contains definitions for the predicate symbols $\mathit{bird}/1$ and $\mathit{fly}/1$. Applying the completion algorithm shown in Table 5.1 to $\mathcal{F}$ and completing both defined function symbols yields the two completion formulas

$$(\forall X)\,(\mathit{bird}(X) \rightarrow X \approx \mathit{tweedy}) \tag{5.16}$$

and

$$(\forall X)\,(\mathit{fly}(X) \rightarrow \neg \mathit{ab}(X) \wedge \mathit{bird}(X)). \tag{5.17}$$

$\mathcal{T}_C(\mathcal{F})$

The *completion* $\mathcal{T}_C(\mathcal{F})$ of a normal logic program $\mathcal{F}$ with defined predicate symbols $\mathcal{A}_D$ is defined as

$$\mathcal{T}_C(\mathcal{F}) = \{G \mid \mathcal{F} \cup \{C_{\mathcal{F},p} \mid p \in \mathcal{A}_D\} \cup \mathcal{F}_C \cup \{(\forall \overline{X})\,\neg p(\overline{X}) \mid p \in \mathcal{A}_P \setminus \mathcal{A}_D\} \models G\}.$$

One should observe that for all non-defined predicate symbols $p/n \in \mathcal{A}_P \setminus \mathcal{A}_D$ it has been assumed that $(\forall \overline{X})\,\neg p(\overline{X})$ holds. In other words, it is assumed that the extension of these predicate symbols is empty.

Returning to the example we find that

$$\mathcal{T}_C(\mathcal{F}) = \{G \mid \mathcal{F} \cup \{(5.16), (5.17)\} \cup \mathcal{F}_C \cup \{(\forall X)\,\neg \mathit{ab}(X)\} \models G\}$$

and consequently

$$\{\neg ab(tweedy),\ fly(tweedy)\} \subseteq \mathcal{T}_C(\mathcal{F}, \{bird, fly\}).$$

Thus, the completion encodes the statement that unless there is something wrong with a bird we are willing to conclude that the bird flies. There is nothing wrong with *tweedy* and hence *tweedy* flies.

We have already observed that adding the completion formula to a satisfiable set of formulas may lead to unsatisfiable knowledge bases. Such cases must be excluded and hence we are interested in finding sufficient conditions such that the completion of a normal logic program is guaranteed to be satisfiable. An example for such a condition is given in the remainder of this section. Much more refined conditions can be found in the literature.

Given an alphabet $\mathcal{A}$, a *level mapping* is total mapping from $\mathcal{A}_P$ to $\mathbb{N}$ assigning a so-called *level* to each predicate symbol occurring in $\mathcal{A}$. For example, the mapping which assigns level 1 to $bird/1$, 2 to $ab/1$ and 3 to $fly/1$ is such a level mapping. *level mapping*

A normal logic program $\mathcal{F}$ is said to be *stratified* iff in each clause of the form *stratified programs*

$$p(\bar{t}) \leftarrow p_1(\overline{s_1}) \wedge \ldots \wedge p_m(\overline{s_m}) \wedge \neg p_{m+1}(\overline{s_{m+1}}) \wedge \ldots \wedge \neg p_n(\overline{s_n})$$

of $\mathcal{F}$ the level of p is greater or equal than the level of each $p_i,\ 1 \leq i \leq m$, and strictly greater than the level of each $p_j,\ m < j \leq n$.

Theorem 5.4 *Let $\mathcal{F}$ be a stratified normal logic programs. Then $\mathcal{T}_C(\mathcal{F})$ is satisfiable.*

A proof of this result can be found e.g. in [Llo93].

5.3.5 Negation as Failure

We have defined the completion of a normal logic program purely semantically. Informally, a normal logic program consists of the "if" halves of the definitions of relations. The completion is obtained by adding to the program the "only-if" parts of these definitions, the equational system $\mathcal{F}_C$, the negative facts for each undefined relation symbols and considering the logical consequences of the extended program. But how can we compute with the completion?

For practical purposes, we do not want to include either $\mathcal{F}_C$ or the "only-if" halves of the definitions of relations or the negative facts of the undefined relation symbols to the program. Instead we would like to compute with the "if" halves of the definitions of relations, i.e., with the program only. In doing so, however, we realize almost immediately that a calculus based on the SLD-resolution rule is incomplete. In the context of normal logic programs goal clauses may contain negative literals. SLD-resolution cannot be applied to negative literals occurring in a goal clause. On the other hand, we do not want to give up the merits of SLD-resolution which allows for an efficient implementation of logic programming.

It is straightforward to verify that

$$\{\neg A \mid \neg A \in \mathcal{T}(\mathcal{F})\} \neq \{\neg A \mid \neg A \in \mathcal{T}_C(\mathcal{F})\}.$$

In other words, the negation occurring in normal logic programs evalutated under the completion semantics is not the usual negation in logic. To make this distinction explicit, we replace the negation sign $\neg/1$ occurring in normal logic programs by $\sim/1$, i.e., (5.14) and (5.15) become

$$p(\bar{t}) \leftarrow A_1 \wedge \ldots \wedge A_m \wedge \sim A_{m+1} \wedge \ldots \wedge \sim A_n$$

and

$$\leftarrow A_1 \wedge \ldots \wedge A_m \wedge \sim A_{m+1} \wedge \ldots \wedge \sim A_n,$$

negation as failure

respectively. $\sim$ is called *negation as failure* for reasons which are explained below. As a concrete example, the logic program shown in (5.13) becomes

$$\begin{array}{rcll} \mathcal{F} & = & \{ & bird(tweedy), \\ & & & fly(X) \leftarrow bird(X) \wedge \sim ab(X) \quad \}, \end{array} \tag{5.18}$$

where I have omitted the universal quantifier and have written the second clause using the reverse implication.

Before turning to the definition of the calculus for computing with the "if" halves only, we need an auxiliary definition. The derivations of a linear resolution calculus can be represented as a so-called *search tree* in a straightforward manner. Such a search tree is said to be *finitely failed* iff the search tree is finite and each leaf is labelled as a failure. As an example consider the program

finitely failed search tree

$$\begin{array}{rcll} \mathcal{F}' & = & \{ & ab(X) \leftarrow brokenWing(X), \\ & & & ab(X) \leftarrow ratite(X),^{3} \\ & & & ratite(X) \leftarrow ostrich(X), \\ & & & ratite(X) \leftarrow emu(X), \\ & & & ratite(X) \leftarrow kiwi(X) \qquad \} \end{array} \tag{5.19}$$

and the query

$$\leftarrow ab(tweedy). \tag{5.20}$$

Its search tree is shown in Figure 5.1. It has only finitely many nodes and no leaf can be evaluated further.

To compute with the "if" halves only, we define a new rule of inference called *SLDNF-resolution* (for SLD-resolution with negation as failure) as follows: Let G be a goal clause consisting of positive and negative literals, $\mathcal{F}$ a normal logic program, L be the selected literal in G and A be a ground atom.

SLDNF-resolution

- If L is positive, then each SLD-resolvent of G using L and some new variant of a clause in $\mathcal{F}$ is also an *SLDNF-resolvent*.
- If L is a ground negative literal, i.e., $L = \sim A$ and the query $\leftarrow A$ finitely fails with respect to $\mathcal{F}$ and SLDNF-resolution, then the *SLDNF-resolvent* of G is obtained from G by deleting L.
- If L is a ground negative literal, i.e., $L = \sim A$ and the query $\leftarrow A$ suceeds with respect to $\mathcal{F}$ and SLDNF-resolution, then the *SLDNF-derivation* of G fails.

[3] A *ratite* is a bird with a flat unkeeled breastbone, unable to fly.

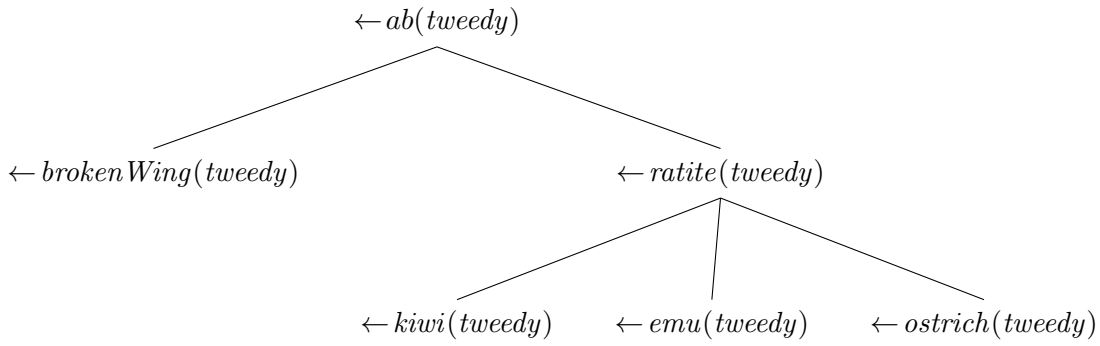


Figure 5.1: A finitely failed search tree.

- If L is negative and non-ground, then without loss of generality we may assume that each literal in G is negative and non-ground.[4] In this case G is said to be *blocked*. *blocked goal clause*

As before, the notions of derivation and refutation can be extended to hold for SLDNF-resolution. A normal logic program $\mathcal{F}$ and a goal clause G are said to *flounder* if some SLDNF-derivation of G with respect to $\mathcal{F}$ is blocked. *floundering*

It should be obvious from this definition why $\sim$ is called *negation as failure*: Let G be the goal clause $\leftarrow \sim A$. If the query $\leftarrow A$ finitely fails, then SLDNF-resolution concludes that $\leftarrow \sim A$ is successful. In other words, the failure to prove $\leftarrow A$ leads to the success of $\leftarrow \sim A$. Conversely, if the query $\leftarrow A$ is successful, then $\leftarrow \sim A$ fails.

Returning to our example, let $\mathcal{F}$ be the union of the clauses shown in (5.18) and (5.19) and let G be the goal clause

$$\leftarrow fly(tweedy). \tag{5.21}$$

Applying SLDNF-resolution using the clause defining *fly* in $\mathcal{F}$ yields

$$\leftarrow \sim ab(tweedy) \wedge bird(tweedy) \tag{5.22}$$

If the selection function selects the first literal in (5.22) then we have to consider the goal clause (5.20). As shown in Figure 5.1 this query finitely fails and, consequently, (5.22) reduces to

$$\leftarrow bird(tweedy),$$

which can be solved using the clause defining *bird* in $\mathcal{F}$. Hence, the initial goal clause is answered positively. As another example consider the query

$$\leftarrow \sim ab(X),$$

which flounders immediately.

[4] Like SLD-resolution, the selection function applied within SLDNF resolution selects literals in a don't-care non-deterministic manner. Hence, if the first choice of the selection function is a negative and non-ground literal, then the selection function may choose another literal.

Computing with negation as failure is non-monotonic. Suppose we learn that *tweedy* is in fact an ostrich and add the fact

$$ostrich(tweedy)$$

to the union of the clauses shown in (5.18) and (5.19). Reconsidering query 5.21 we again obtain (5.22) in one step. But now the query

$$\leftarrow ab(tweedy)$$

can be successively reduced to

$$\leftarrow ratite(tweedy)$$

and

$$\leftarrow ostrich(tweedy),$$

which in turn succeeds. Hence, the initial goal fails in this case.

Theorem 5.5 *Let $\mathcal{F}$ be a logic program. SLDNF-resolution is sound with respect to the completion of $\mathcal{F}$.*

This result was shown in [Cla78]. On the other hand, SLDNF-resolution is generally incomplete, but complete for restricted classes of programs. For a detailed discussion see [AB94].

One should observe, that sometimes negation as failure in logic programs leads to undesirable results. The following example is due to McCarthy and can be found in [GL90]: A school bus may cross railway tracks under the condition that there is no approaching train. The naive solution

$$cross \leftarrow \sim train$$

allows the bus to cross tracks when there is no information about either the presence or the absense of a train – for instance, when the driver's vision is blocked. In this case the use of classical negation

$$cross \leftarrow \neg train$$

leads to the desired result: Crossing tracks is only allowed if the negative fact $\neg train$ is established. Whenever we cannot assume that the available positive information about a predicate is complete, then the closed world assumption cannot be applied. We will come back to this and related examples in Section 5.6.

5.4 Circumscription

Using the closed world assumption or the completion does not lead to the intended result if we have to deal with formulas like

$$p(a) \vee p(b) \tag{5.23}$$

or

$$(\exists X)\, green(X).$$

We are interested in the minimal models of these formulas. Intuitively, the minimal models for (5.23) are the models of

$$(\forall X)\,(p(X) \leftrightarrow X \approx a) \vee (\forall X)\,(p(X) \leftrightarrow X \approx b).$$

In other words, either a is the only element in the extension of p/m or so is b but not both. More generally, we want to conjecture that the tuples $(X_1, \ldots, X_m)$ which can be shown to satisfy a relation p/m are all the tuples satisfying this relation. Speaking with McCarthy [McC90], we want to *circumscribe* the set of relevant tuples.

!

Formally, we consider a formula F. Let $p(\overline{X})$ denote the atom $p(X_1, \ldots, X_m)$ and

$F\{p/p^*\}$

$$F\{p/p^*\}$$

the formula obtained from F by replacing each occurrence of p/m by p^*/m. The *circumscription* of p in F is the second order scheme

circumscription
$Circ(F, p)$

$$Circ(F,p) = (F\{p/p^*\} \wedge (\forall \overline{X})\,(p^*(\overline{X}) \rightarrow p(\overline{X}))) \rightarrow (\forall \overline{X})(p(\overline{X}) \rightarrow p^*(\overline{X}))$$

It is a scheme because p^* is a predicate parameter which may be substituted by an arbitrary first order formula. $F\{p/p^*\}$ states that any condition imposed on p/m is imposed on p^*/m as well. $(\forall \overline{X})\,(p^*(\overline{X}) \rightarrow p(\overline{X}))$ states that any tuple in the extension of p^*/m is also in the extension of p/m. Likewise, $(\forall \overline{X})(p(\overline{X}) \rightarrow p^*(\overline{X}))$ states that any tuple in the extension of p/m is also in the extension of p^*/m.

As a first example taken from the blocks world consider the formula

$$F = \mathit{isblock}(a) \wedge \mathit{isblock}(b) \wedge \mathit{isblock}(c)$$

In this example, only the objects a, b and c must be any extension of the predicate symbol $\mathit{isblock}/1$, but there may be other objects. We want to make sure that a, b and c are all the objects in any extension of p/m. Circumscribing $\mathit{isblock}$ in F yields

$$(p^*(a) \wedge p^*(b) \wedge p^*(c) \wedge (\forall X)\,(p^*(X) \rightarrow \mathit{isblock}(X))) \rightarrow (\forall X)\,(\mathit{isblock}(X) \rightarrow p^*(X)). \quad (5.24)$$

If we substitute

$$p^*(X) \leftrightarrow (X \approx a \vee X \approx b \vee X \approx c)$$

in (5.24) and use F, we find the the condition of the implication (5.24) is satisfied and, consequently, its conclusion

$$(\forall X)\,(\mathit{isblock}(X) \rightarrow (X \approx a \vee X \approx b \vee X \approx c))$$

holds. In other words, there are just the three blocks a, b and c in this rather simple scenario.

As a second example reconsider the disjunction (5.23). Circumscribing p in this formula yields

$$((p^*(a) \vee p^*(b)) \wedge (\forall X)\,(p^*(X) \rightarrow p(X))) \rightarrow (\forall X)\,(p(X) \rightarrow p^*(X)). \quad (5.25)$$

We may now substitute

$$p^*(X) \leftrightarrow X \approx a$$

in (5.25) to obtain

$$((a \approx a \vee b \approx a) \wedge (\forall X)\,(X \approx a \rightarrow p(X))) \rightarrow (\forall X)\,(p(X) \rightarrow X \approx a)$$

which simplifies to

$$p(a) \rightarrow (\forall X)\,(p(X) \rightarrow X \approx a). \tag{5.26}$$

Similarly, we may substitute

$$p^*(X) \leftrightarrow X \approx b$$

in (5.25) to obtain

$$((a \approx b \vee b \approx b) \wedge (\forall X)\,(X \approx b \rightarrow p(X))) \rightarrow (\forall X)\,(p(X) \rightarrow X \approx b)$$

which simplifies to

$$p(b) \rightarrow (\forall X)\,(p(X) \rightarrow X \approx b). \tag{5.27}$$

Finally, (5.26), (5.27) combined with (5.23) leads to

$$(\forall X)\,(p(X) \rightarrow X \approx a) \vee (\forall X)\,(p(X) \rightarrow X \approx b)$$

which is the intended result. More examples can be found in [McC90].

In order to characterize the circumscription of a predicate p/m in a formula F semantically, we consider the minimal models of F with respect to $\{p/m\}$. *G follows minimally from F with respect to p/m*, written $F \models_{\{p\}} G$, iff G holds in all models of F which are minimal in $\{p/m\}$.

follows minimally $\models_{\{p\}}$

Theorem 5.6 *$Circ(F, p)$ holds in all models of F which are minimal in $\{p/m\}$.*

A proof of this theorem can be found in [McC90]. Moreover, as an immediate consequence of this result we find that:

Corollary 5.1 *If $F \wedge Circ(F, p) \models G$ then $F \models_{\{p\}} G$.*

Some remarks are helpful at this point:

- It is easy to show that computing with circumscription is a non-monotonic form of reasoning.
- The circumscription of a predicate may again lead to an unsatisfiable theory. As in the case of the closed world assumption and the completion there are known sufficient conditions, which guarantee satisfiability (see e.g. [Lif86]).
- Although the circumscription of a predicate involves a second order scheme there are cases in which circumscription can be reduced to first order reasoning (see e.g. [Lif85]). But this is not always possible as can be demonstrated by the following formula.

  $$(\forall V, W)\,(q(V, W) \rightarrow p(V, W)) \wedge (\forall X, Y, Z)\,(p(X, Y) \wedge p(Y, Z) \rightarrow p(X, Z)) \tag{5.28}$$

  This formula specifies that the set of tuples satisfying $p/2$ contains the transitive closure of the set of tuples satisfying $q/2$. The circumscription of $p/2$ in (5.28) specifies that the set of tuples satisfying $p/2$ is exactly the set of tuples satisfying $q/2$. Because the transitive closure of a binary relation cannot be defined in first order logic, we cannot reduce the circumscription of $p/2$ in (5.28) to first order logic.

- Many extensions of circumsciption are known. We may circumscribe more than one predicate in parallel, we may allow to enlarge the extension of some predicate symbols while circumscribing others, we may circumscribe predicates using priorities or we may circumscribe a predicate only in one point (see e.g. [Lif87]).

5.5 Default Logic

The reasoning patterns considered so far in this chapter are of the form "*unless any information to the contrary is known assume that ... holds*." Under the closed world assumption, in programming with completed predicates as well as circumscribing predicates, this line of reasoning was modelled by extending the knowledge base. We have already seen that a similar effect can be achieved by altering the logical consequence relation. The most prominent approach in this respect is the so-called *default logic*, which was introduced by Reiter in [Rei80].

5.5.1 Some Examples

Many examples in common sense reasoning are of the following form: "*Most objects of sort s have property p. Object o is of sort s. Does object o have property p?*" For example, most birds fly. Given a particular bird, say *tweedy*, what do we know about its capabilities to fly? Well, most of us are willing to conclude that *tweedy* flies unless we happen to know that it belongs to one of the known exceptions like being an ostrich or a penguin.

How can we represent our knowledge about birds and their capabilities to fly? In first order logic this can naturally be done by explicitely stating the exceptions:

$$(\forall X)\,(bird(X) \wedge \neg penguin(X) \wedge \neg ostrich(X) \wedge \ldots \rightarrow fly(X)) \tag{5.29}$$

There are at least two difficulties with this approach.

- In common sense reasoning we usually do not know all exceptions. In other words, we usually do not know what is really meant by "..." in (5.29). For example, a yet unknown species of non-flying birds may live in the rain forest.
- Suppose that we happen to know all exceptions, then (5.29) does not allow us to conclude that *tweedy* flies if we just happen to know that it is a bird, because we cannot conclude that *tweedy* is not a penguin and not an ostrich etc.

In other words, using first order logic in a straightforward manner blocks us from concluding that *tweedy* flies, although we intuitively would like to do so.

Just knowing that *tweedy* is a bird, we somehow would like to conclude that *tweedy* flies *by default*. How is the default to be interpreted? We may take it as saying that "unless any information to the contrary is known we conclude that *tweedy* flies." But, then, what is the precise meaning of this phrase? Does it mean that the exceptions are not logical conseqences of our knowledge gathered so far, or does it mean that we finitely failed to prove the exceptions? *default*

In default logic we interpret the phrase "unless any information to the contrary is known we assume that *tweedy* flies" by "it is consistent to assume that *tweedy* can fly." More formally, this interpretation is represented by a new kind of inference rule called *default rule*

$$bird(X) \ : \ fly(X) \ / \ fly(X) \ .$$

Informally, this rule is read as "if X is a bird and it is consistent to assume that X flies, then conclude that X flies." The exceptions to flight are then given by standard first order sentences:

$$\begin{array}{ll} \{ & (\forall X)\,(penguin(X) \rightarrow \neg fly(X)), \\ & (\forall X)\,(ostrich(X) \rightarrow \neg fly(X)), \\ & \ldots \qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

One should observe that a conclusion like $fly(tweedy)$ drawn with the help of a default rule has the status of a belief. It may change if additional information like *tweedy* being a penguin is discovered.

There still remains the problem of how to interprete the phrase "*it is consistent to assume that tweedy flies*". This is probably the most difficult issue in default logic. Informally, consistency is defined with respect to all first order formulas in the knowledge base and all other beliefs sanctioned by all other default rules in force. A formal definition will be given in Section 5.5.2.

Default rules can also be used to represent phrases like "*Few objects of sort s have property p*". For example, the statement few men have been on the moon, is represented by

$$man(X) \ : \ \neg moon(X) \ / \ \neg moon(X) \ .$$

5.5.2 Default Knowledge Bases

default rule Let $\langle \mathcal{A}, \mathcal{L}, \models \rangle$ be a first order logic. A *default rule* is any expression of the form

$$F : G_1, \ldots, G_n \ / \ H \ .$$

F is called *prerequisite*, $G_1, \ldots, G_n$ are called *justifications* and H is called *consequent* of the default rule. A default rule is said to be *closed* iff all formulas occurring in it are closed, and it is said to be *open* iff it is not closed. An open default rule is a scheme and represents the set of its ground instances.

There are several special cases of default rules.

- If F is missing, then this is interpreted as $F \equiv \langle\,\rangle$. In other words, the prerequisite does always hold in this case.
- If $n = 0$, then the default rule is a rule in the underlying first order logic. This case is not of interest in this chapter as it is subsumed by the first order logic.
- If $n = 1$ and $G_1 = H$, then the default rule is said to be *normal*.
- If $n = 1$ and $G_1 = H \wedge H'$, then the default rule is said to be *semi-normal*.

Most of the examples considered here and in the literature are either normal or semi-normal.

A *default knowledge base*[5] is a pair $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$, where $\mathcal{F}_D$ is a set of at most countably many default rules and $\mathcal{F}_W$ is a set of at most countably many closed first order formulas over $\mathcal{A}$. A default knowledge base is said to be *closed* iff all default rules occurring in it are closed, and it is said to be *open* iff it is not closed.

default knowledge base

As an example consider the following simple scenario: *Jane and John are married. John lives in Munich. Jane works at the Computer Science Department of the TU Dresden. Most people's hometown is the hometown of his/her spouse. Most people's hometown is where his/her employer is located*. This scenario can straightforwardly be represented by a default knowledge base.

$$\begin{array}{lcll} \mathcal{F}_D & = & \{ & spouse(X,Y) \ \wedge \ htown(Y) = Z : htown(X) = Z \ / \ htown!(X) \approx Z \ , \\ & & & employer(X,Y) \ \wedge \ location(Y) \approx Z : htown(X) \approx Z \ \setminus \ htown(X) \approx Z \quad \} \\ \mathcal{F}_W & = & \{ & spouse(jane, john), \\ & & & htown(john) \approx munich, \\ & & & employer(jane, tud), \\ & & & location(tud) \approx dresden, \\ & & & (\forall X, Y, Z)\,(htown(X) \approx Y \wedge htown(X) \approx Z \rightarrow Y \approx Z) \qquad \} \end{array}$$

The last formula occurring in $\mathcal{F}_W$ states that a person can have only one hometown.

If we now apply the substitution

$$\theta_1 = \{X \mapsto jane, \ Y \mapsto john, \ Z \mapsto munich\}$$

to the first default, then we find that

$$\mathcal{F}_W \models spouse(jane, john) \wedge htown(john) \approx munich.$$

Because it is consistent to assume that *jane*'s hometown is *munich* the default rule is applicable and we conclude that *jane*'s hometown is *munich*. Similarily, we may apply the substitution

$$\theta_2 = \{X \mapsto jane, \ Y \mapsto tud, \ Z \mapsto dresden\}$$

to the second default to find that

$$\mathcal{F}_W \models employer(jane, tud) \wedge htown(jane) \approx dresden.$$

However, having concluded that *jane*'s hometown is *munich*, this is no longer consistent with respect to $\mathcal{F}_W$ and the previously drawn default conclusions to assume that *jane*'s hometown is *dresden*. Consequently, the second default rule is not enforced. One should observe that if we would have considered the second default rule first, then we had concluded that *jane*'s hometown is Dresden and consequently had rejected the first default rule.

This seems to be a surprising behavior at first sight because it demonstrates that there is not a unique and well-defined relation between a default knowledge base and the theory

[5] In the literature, default knowledge bases are often called default theories. In this book, theories denote sets of logical consequences of sets of formulas. In many logics like propositional and first order logic there is a unique and well-defined relation between a set of formulas and its logical consequences and, hence, it is acceptable to call a set of formulas a theory. As we will see later such a unique and well-defined relation does not exist for default knowledge bases.

defined by this knowledge base. We may believe that *jane* lives in *munich* or that *jane* lives in *dresden* but not both. For this reason, I will not be talking about theories with respect to a default knowledge base but about extensions in the following section.[6]

5.5.3 Extensions of Default Knowledge Bases

Any formalism for non-monotonic reasoning is based on the observation that a knowledge base is usually incomplete. Nevertheless, in many situations we would like or even need to draw conclusions despite of the fact that our knowledge base is incomplete. The default rules sanction additional pieces of information which are added to the knowledge base as long as this addition does not lead to inconsistencies. We have to keep this in mind when we formally define the extensions of a default knowledge base.

Let $\mathcal{F}$ be a set of closed first-order formulas. Intuitively, an extension $\mathcal{F}_E$ of $\mathcal{F}$ should have the following properties:

- $\mathcal{F} \subseteq \mathcal{F}_E$, i.e., $\mathcal{F}$ should be contained in its extension.
- $\mathcal{T}(\mathcal{F}_E) = \mathcal{F}_E$, i.e., the extension should be deductively closed.
- For each default rule, if the prerequisite is contained in $\mathcal{F}_E$ and the negation of each justification is not in $\mathcal{F}_E$, then the conclusion should occur in $\mathcal{F}_E$. In other words, $\mathcal{F}_E$ should be closed under the application of default rules.

This motivates the following definition.

Let $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ be a default knowledge base. For any set $\mathcal{F}$ of closed first order formulas let $\Gamma(\mathcal{F})$ be the smallest set satisfying the following three properties:

1. $\mathcal{F}_W \subseteq \Gamma(\mathcal{F})$.
2. $\mathcal{T}(\Gamma(\mathcal{F})) = \Gamma(\mathcal{F})$.
3. If $F : G_1, \ldots, G_n \,/\, H \in \mathcal{F}_D$, $F \in \Gamma(\mathcal{F})$ and for all $1 \leq j \leq n$ we find that $\neg G_i \notin \mathcal{F}$ then $H \in \Gamma(\mathcal{F})$.

extension $\mathcal{F}$ is said to be an *extension* of $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ iff $\Gamma(\mathcal{F}) = \mathcal{F}$.

From this definition we conclude immediately that the set of extensions of a default knowledge base $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ is a subset of the set of models for $\mathcal{F}_W$. A more intuitive characterizaton of extensions is given in the following theorem, whose proof can be found in [Rei80].

Theorem 5.7 *Let $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ be a default knowledge base and $\mathcal{F}$ be a set of sentences. Define*

$$\mathcal{F}_0 = \mathcal{F}_W$$

and for $i \geq 1$:

$$\mathcal{F}_{i+1} = \mathcal{T}(\mathcal{F}_i) \cup \{H \mid F : G_1, \ldots, G_n \,/\, H \in \mathcal{F}_D,\ F \in \mathcal{F}_i \text{ and for all } 1 \leq j \leq n\ \neg G_j \notin \mathcal{F}\}.$$

Then, $\mathcal{F}$ is an extension of $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ iff $\mathcal{F} = \bigcup_{i=0}^{\infty} \mathcal{F}_i$.

[6] The extensions of a default knowledge base should not be confused with the notion of an extension of a predicate symbol under an interpretation defined in Section **??**.

One should observe the occurrence of $\mathcal{F}$ in the definition of $\mathcal{F}_{i+1}$. This forces us to guess an extension; thereafter, Theorem 5.7 can be applied to verify that our guess is correct. To illustrate the notion of an extension we consider the default knowledge base $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$, where

$$\begin{array}{lcl} \mathcal{F}_D & = & \{ \quad bird(X) : fly(X) \ / \ fly(X) \quad \}, \\ \mathcal{F}_W & = & \{ \quad bird(tweedy) \qquad\qquad\quad \}, \end{array}$$

and let

$$\mathcal{F} = \mathcal{T}(\{bird(tweedy),\ fly(tweedy)\}).$$

Theorem 5.7 can now be applied to verify that $\mathcal{F}$ is an extension. Let

$$\mathcal{F}_0 = \mathcal{F}_W = \{bird(tweedy)\}.$$

Then,

$$\mathcal{F}_1 = \mathcal{T}(\{bird(tweedy)\}) \cup \{fly(tweedy)\}$$

and

$$\mathcal{F}_i = \mathcal{T}(\{bird(tweedy), fly(tweedy)\}),$$

for all $i \geq 2$. Consequently,

$$\bigcup_{i=0}^{\infty} \mathcal{F}_i = \mathcal{T}(\{bird(tweedy), fly(tweedy)\}) = \mathcal{F}.$$

Extensions are not unique. The interested reader may verify that the example scenario about the couple John and Mary discussed in Subsection 5.5.2 admits the two extensions

$$\begin{array}{l} \mathcal{T}(\{spouse(jane, john),\ htown(john) \approx munich, employer(jane, tud), \\ \quad location(tud) \approx dresden,\ htown(jane) \approx munich\}) \end{array}$$

and

$$\begin{array}{l} \mathcal{T}(\{spouse(jane, john),\ htown(john) \approx munich, employer(jane, tud), \\ \quad location(tud) \approx dresden,\ htown(jane) \approx tud\}). \end{array}$$

Reasoning in a default logic is reasoning with respect to the extensions of the default knowledge bases. We distinguish two kinds of reasoning. Let $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ be a default knowledge base.

- G *follows credolously* from $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ (in symbols $\langle \mathcal{F}_D, \mathcal{F}_W \rangle \models_b G$) iff there exists an extension $\mathcal{F}$ of $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ such that $G \in \mathcal{F}$. *credolous conclusion*
- G *follows sceptically* from $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ (in symbols $\langle \mathcal{F}_D, \mathcal{F}_W \rangle \models_s G$) iff for all extensions $\mathcal{F}$ of $\langle \mathcal{F}_D, \mathcal{F}_W \rangle$ we find $G \in \mathcal{F}$. *sceptical conclusion*

In the scenario involving John and Jane we find that

$$\begin{array}{lll} \langle \mathcal{F}_D, \mathcal{F}_W \rangle & \models_s & spouse(jane, john) \wedge htown(john) \approx munich \\ & & \wedge\ employer(jane, tud) \wedge location(tud) \approx dresden \end{array}$$

but concerning Jane's hometown only credolous conclusions are possible:

$$\langle \mathcal{F}_D, \mathcal{F}_W \rangle \models_b htown(jane) \approx munich$$

and

$$\langle \mathcal{F}_D, \mathcal{F}_W \rangle \models_b htown(jane) \approx dresden.$$

One should observe that

$$\langle \mathcal{F}_D, \mathcal{F}_W \rangle \models_s htown(jane) \approx munich \vee htown(jane) \approx dresden.$$

We conclude this section by adding some remarks:

- It is again easy to see that default logic is non-monotonic and we leave it to the interested reader to design an example demonstrating this property.
- Extensions of default knowledge bases are always satisfiable.
- There are examples where an extension of a default knowledge base contains counter-intuitive facts. For example, let

  $$\mathcal{F}_W = \{broken(left - arm) \vee broken(right - arm)\}$$

  and

  $$\mathcal{F}_D = \{ : usable(X) \wedge \neg broken(X) \ / \ usable(X)\}.$$

  The only extension of this default knowledge base contains

  $$usable(left - arm) \wedge usable(right - arm)$$

  which is clearly counter-intuitive given $\mathcal{F}_W$.
- There are a variety or extensions of default logics most notably default logics where priorities between otherwise competing default rules are defined.

5.6 Answer Set Programming

The driving idea behind most of the research in non-monotonic reasoning was to "*jump to a conclusion*" in scenarios, where it is intrinsically impossible to formalize every aspect. Reviewing the techniques presented in this chapter so far, however, reveals that they are extremely complex. One does not "*jump to a conclusion*" but rather finds itself in rather lengthy, often intractable computations which in most cases are not even guaranteed to succeed because the problem is undecidable or the calculus is incomplete. Can we efficiently implement non-monotonic reasoning techniques capable of modelling common sense scenarios?

On the other hand, the techniques presented are often not powerful enough to handle interesting scenarios. For example, completion has considerable problems with disjunctions. The use of negation as failure in logic programming forces the programmer to replace the usual negation in logic by negation as failure. But as already shown at the end of Subsection 5.3.5 there are cases, where negation of failure leads to undesirable results. In fact, there are cases, where we would like to have both forms of negation in one program. Consider the following example taken from [GL90]: A certain college in the USA uses the following rules for awarding scholarships to students:

1. Every student with a GPA of at least 3.8 is eligible.
2. Every minority student with a GPA of at least 3.6 is eligible.
3. No student with a GPA under 3.6 is eligible.
4. The students whose eligibility is not determined by these rules are interviewed by the scholarship committee.

The rules can be encoded as follows:

$$\begin{array}{rcll} \mathcal{F}_1 & = & \{ & \mathit{eligible}(X) \leftarrow \mathit{highGPA}(X), \\ & & & \mathit{eligible}(X) \leftarrow \mathit{minority}(X) \wedge \mathit{fairGPA}(X), \\ & & & \neg\mathit{eligible}(X) \leftarrow \neg\mathit{fairGPA}(X), \\ & & & \mathit{interview}(X) \leftarrow \sim \mathit{eligible}(X) \wedge \sim\neg\mathit{eligible}(X) \quad \} \end{array}$$

The last implication specifies that $\mathit{interview}(X)$ holds if there is no evidence that $\mathit{eligible}(X)$ holds and there is also no evidence that $\neg\mathit{eligible}$ holds.

$\mathcal{F}_1$ is to be used in conjunction with a set of literals specifying the values of the predicates *minority*, *highGPA* and *fairGPA*. Assume that this set is given by

$$\begin{array}{rcll} \mathcal{F}_2 & = & \{ & \mathit{fairGPA}(\mathit{john}) \leftarrow, \\ & & & \neg\mathit{highGPA}(\mathit{john}) \leftarrow \quad \}. \end{array}$$

It does not contain any information about whether *john* belongs to a minority. He may not be a minority student, but he may as well be a minority student who, for whatever reasons, did not state this fact on his application. Now, the interesting question is, what happens with *john* ?

In this section I will present a technique that can handle this problem and has attracted much attention recently: *answer set programming*. It is a generalization of the so-called *stable model semantics* developed for logic programming by Michael Gelfond and Vladimir Lifschitz [GL88] and can handle disjunction as well as both kinds of negation. Moreover, efficient implementions are known. An excellent overview can be found in [Lif99].

stable models

5.6.1 Answer Sets

I start by defining the alphabet and the language of the programs underlying answer set programming. The alphabet is just the usual alphabet for propositional logic extended by the connective $\sim/1$, which intuitively represents negation as failure. A *rule* is an expression of the form

rule

$$L_1 \vee \ldots \vee L_k \vee \sim L_{k+1} \vee \ldots \vee \sim L_l \leftarrow L_{l+1} \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n, \quad (5.30)$$

where all L_i, $1 \leq i \leq n$, are literals and $0 \leq k \leq l \leq m \leq n$. A rule of the form shown in (5.30) is called *constraint* if $k = l = 0$. It will later become clear why the name constraint has been chosen. A *program* is a set of rules.

constraint

program

Thus $\mathcal{F}_1 \cup \mathcal{F}_2$ is a program, and so is

$$\begin{array}{rcll} \mathcal{F}_3 & = & \{ & s \vee r \leftarrow , \\ & & & \neg b \leftarrow r \quad \}. \end{array}$$

The latter program describes a little scenario where an agent knows that either the sprinkler is on (s) or it is raining (r), and if it is raining then the sky is not blue (b).

The notion of an answer set is first defined for programs which do not contain negation as failure, i.e., for which $k = l$ and $m = n$ for every rule (5.30) in the program. Let $\mathcal{F}$ be such a program and let $\mathcal{M}$ be a satisfiable set of literals. $\mathcal{M}$ is said to be *closed* under $\mathcal{F}$ if for every rule (5.30) of $\mathcal{F}$ we find that $\{L_1, \ldots, L_k\} \cap \mathcal{M} \neq \emptyset$ whenever $\{L_{l+1}, \ldots, L_m\} \subseteq \mathcal{M}$. $\mathcal{M}$ is said to be an *answer set* for $\mathcal{F}$ if M is minimal among the sets closed under $\mathcal{F}$ (relative to set inclusion).

closed

answer set

For the simple example program $\mathcal{F}_3$ we find two answer sets, viz. $\{s\}$ and $\{r, \neg b\}$. If we extend $\mathcal{F}_3$ by the constraint

$$\leftarrow \; s \tag{5.31}$$

then we obtain a new program $\mathcal{F}_4$ whose only answer set is $\{r, \neg b\}$. $\mathcal{F}_4$ illustrates a general property of constraints, which in facts sanctions the name "constraint": adding a constraint to a program affects its collection of answer sets by eliminating the answer sets that "violate" this constraint.

We now extend the notion of answer sets to programs with negation as failure. Let $\mathcal{F}$ be a program and $\mathcal{M}$ a satisfiable set of literals. The *reduct* $\mathcal{F}|_{\mathcal{M}}$ of $\mathcal{F}$ relative to $\mathcal{M}$ is the set of rules

reduct

$$L_1 \vee \ldots \vee L_k \leftarrow L_{l+1} \wedge \ldots \wedge L_m$$

for all rules (5.30) in $\mathcal{F}$ such that $\{L_{k+1}, \ldots, L_l\} \subseteq \mathcal{M}$ and $\{L_{m+1}, \ldots, L_n\} \cap \mathcal{M} = \emptyset$. One should observe that $\mathcal{F}|_{\mathcal{M}}$ is a program without negation by failure. $\mathcal{M}$ is said to be an *answer set* for $\mathcal{F}$ iff $\mathcal{M}$ is an answer set for $\mathcal{F}|_{\mathcal{M}}$.

answer set

As a simple example consider the program

$$\mathcal{F}_5 = \{p \leftarrow \sim q\}$$

The reduct of $\mathcal{F}_5$ relative to $\{p\}$ is

$$\{p \leftarrow\}.$$

Because $\{p\}$ is an answer set for this reduct, it is an answer set for $\mathcal{F}_5$. The reduct of $\mathcal{F}_5$ relative to $\{p, q\}$ is the empty set. Because $\{p, q\}$ is not an answer set for the empty set, it is not an answer set for $\mathcal{F}_5$.

As another example consider the program

$$\{\neg p \leftarrow \sim p\}.$$

It expresses the closed world assumption for the predicate p. Its only answer set is $\{\neg p\}$. The opposite assumption can also be expressed:

$$\{p \leftarrow \sim \neg p\}.$$

Its only answer set is $\{p\}$.

So far, programs and answer sets were only defined for propositional literals. It is, however, possible to extend the approach to literals built up from n-ary predicate symbols, finitely many constants and variables. In this case, rules are viewed as schemas representing all possible ground instantiations. For example, consider an alphabet with two

constants 1 and 2, predicate symbol $in/2$ and variables X and Y. In this case, the rule

$$in(X,Y) \vee \neg in(X,Y) \leftarrow$$

represent the set consisting of the ground rules

$$\begin{array}{l} in(1,1) \vee \neg in(1,1) \leftarrow \\ in(1,2) \vee \neg in(1,2) \leftarrow \\ in(2,1) \vee \neg in(2,1) \leftarrow \\ in(2,2) \vee \neg in(2,2) \leftarrow \end{array}$$

Each ground literal can be equivalently replaced by a propositional literal and, hence, the approach presented in this section can be applied.

With the help of this little trick we can now reconsider the introductory example $\mathcal{F}_1 \cup \mathcal{F}_2$. This program has only one answers set, viz.

$$\{fairGPA(john),\ \neg highGPA(john),\ interview(john)\} \tag{5.32}$$

This answer set tells us that *john* is to be invited for the interview.

Answer sets are non-monotonic in the sense that the addition of rules may lead to new answer sets. In other words, if M is an answer set for $\mathcal{F} \cup \mathcal{F}'$, then it may not be an answer set for $\mathcal{F}$. To exemplify this behavior reconsider the scholarship example once again and add *non-monotonicity*

$$minority(john) \leftarrow$$

to $\mathcal{F}_2$. In this case, (5.32) is no longer an answer set, but

$$\{fairGPA(john),\ \neg highGPA(john),\ minority(john),\ eligible(john)\}$$

is.

Whereas the program

$$\begin{array}{lllll} \{ & q & \leftarrow & p \wedge \sim q, & \\ & p & \leftarrow & , & \\ & q & \leftarrow & & \} \end{array}$$

has the unique answer set $\{p, q\}$, it has no answer sets at all if the last rule is deleted.

5.6.2 Programming with Answer Sets

To illustrate the use of answer sets I will show how answer set programming can be used to find all Hamiltonian cycles of a given finite directed graph G. A *Hamiltonian cycle* is a cyclic tour through the graph visiting each vertex exactly once. The problem of finding a Hamiltonian cycle is known to be NP-complete. Figure 5.2 shows a graph with two different Hamiltonian cycles. *Hamiltonian cycle*

Let G be a graph with vertices $0, \ldots, n$. The alphabet contains the constants $0, \ldots, n$, the predicate symbol $reachable/1$ and the predicate symbol $in/2$. Informally, $reachable(i)$ represents the fact that vertex i is reachable from vertex 0, and $in(i,j)$ represents the fact that the edge from vertex i to vertex j is in the Hamiltonian cycle. We are going

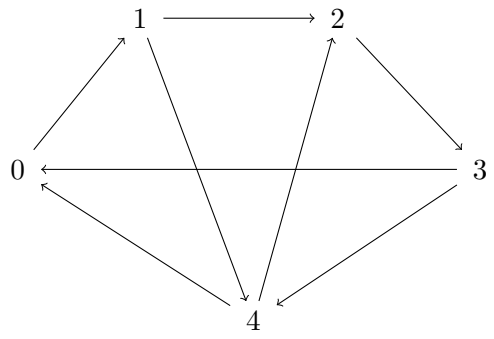


Figure 5.2: A graph with two different Hamiltonian cycles: $(0,\ 1,\ 4,\ 2,\ 3,\ 0)$ and $(0,\ 1,\ 2,\ 3,\ 4,\ 0)$.

to specify a program such that each answer set M of the program corresponds to a Hamiltonian cycle in the sense that

$$\{\langle u, v\rangle \mid in(u, v) \in M\}$$

is the set of edges in the Hamiltonian cycle.

The first group of rules in the program is

$$\{in(u, v) \vee \neg in(u, v) \leftarrow \mid \langle u, v\rangle \in G\} \tag{5.33}$$

stating that each edge in G either is in the Hamiltonian cycle or is not in the Hamiltonian cycle. We will now add constraints that eliminate all subsets of the edges in G which are not Hamiltonian cycles. This is done in two steps. Firstly, we eliminate subsets in which a vertex has more than one outgoing edge:

$$\{\leftarrow in(u, v) \wedge in(u, w) \mid \langle u, v\rangle, \langle u, w\rangle \in G \text{ and } v \not\approx w\} \tag{5.34}$$

Secondly, we eliminate subsets in which a vertex has more than one ingoing edge:

$$\{\leftarrow in(v, u) \wedge in(w, u) \mid \langle v, u\rangle, \langle w, u\rangle \in G \text{ and } v \not\approx w\} \tag{5.35}$$

It remains to ensure that by starting from the vertex 0 following the in-edges one can visit all vertices in G and return to 0. To this end, we specify

$$\begin{array}{l} \{reachable(u) \leftarrow in(0, u) \mid \langle 0, u\rangle \in G\} \\ \cup\ \{reachable(v) \leftarrow reachable(u) \wedge in(u, v) \mid \langle u, v\rangle \in G\} \\ \cup\ \{\leftarrow \sim reachable(u) \mid 0 \leq u \leq n\} \end{array} \tag{5.36}$$

We leave it to the interested reader to check that the answer sets for the program

$$\mathcal{F} = (5.33) \cup (5.34) \cup (5.35) \cup (5.36)$$

correspond to the Hamiltonian cycles of the example graph.

5.6.3 Computing Answer Sets

Various systems have been developed which compute answer sets for restricted classes of programs. These systems have taken a remarkable development over the last couple of years in terms increasing their expressiveness as well as their efficiency considerably. Notably, these are the systems like *Smodels* [NS97, Nie00], *Dlv* [ELM$^+$98] or *DeReS* [CMMT96].

5.7 Remarks

This chapter contains just some of the major approaches to model non-monotonic reasoning. There is a variety of other techniques like inheritance networks (e.g. [HTT90]), modal non-monotonic logics, auto-epistemic logics (e.g. [MT91]), conditional logics or relevance logics [Del91], for which we must refer the reader to the literature. There are also many papers on relating the various approaches to non-monotonic reasoning, although no general theory about non-monotonic reasoning has been developed so far.

We haven't mentioned complexity results for the various non-monotonic reasoning systems, althouth many such results have been established in the literature.

BELIEF REVISION, TRUTH MAINTENANCE SYSTEMS, ELABORATION TOLERANCE

Bibliography

[AB94] K.R. Apt and R. Bol. Logic programming and negation: A survey. Technical Report CS-R9402, CWI Centrum voor Wiskunde en Informatica, 1994.

[Ama71] S. Amarel. On representation of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Edinburgh University Press, 1971.

[Ave95] J. Avenhaus. *Reduktionssysteme*. Springer, Berlin, Heidelberg, New York, 1995.

[Baa11] F. Baader. What's new in description logics. *Informatik Spektrum*, 34(5):434–442, 2011.

[BCM + 03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[Bib92] W. Bibel. Intellectics. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 705–706. John Wiley, New York, 1992.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, 1988.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM Journal of Computing*, 4:412–430, 1975.

[Bra78] R. J. Brachman. Structured inheritance networks. In W. A. Woods and R. J. Brachman, editors, *Research in Natural Language Understanding, Annual Report*, Quarterly Research Reports No. 1, BBN Report No. 4274. Bolt, Beranek and Newman Inc., 1978.

[Bro87] F. M. Brown. *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*. Morgan Kaufmann Publishers, Inc., 1987.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[BS94] F. Baader and J. Siekmann. Unification theory. In J.A. Robinson D.M. Gabbay, C.J. Hogger, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2*, pages 41–125. Oxford University Press, 1994.

[BS99] F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers B.V., 1999.

[Buc87] B. Buchberger. History and basic features of the critical pair / completion procedure. *Journal of Symbolic Computation*, 3(1,2):3–38, 1987.

[Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[Bün98] R. Bündgen. *Termersetzungssysteme*. Vieweg, 1998.

[Bur69] R. M. Burstall. Proving properties of programs by structural induction. *Comput. J.*, 12(1), 1969.

[Bür86] H.-J. Bürckert. Lazy theory unification in Prolog: An extension of the Warren abstract machine. In *Proceedings of the German Workshop on Artificial Intelligence*, pages 277–288, 1986.

[BvHHS90] A. Bundy, F. van Hermelen, C. Horn, and A. Smaill. The oyster-clam system. In *Proceedings of the Tenth International Conference on Automated Deduction*, volume 449 of *LNAI*. Springer, 1990.

[CDT91] L. Console, D. Dupré, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 2(5):661–690, 1991.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum, New York, 1978.

[CMMT96] P. Cholewiński, V.W. Marek, A. Mikitiuk, and M. Truszczyński. Default reasoning system DeReS. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation anr Reasoning*, pages 518–528. Morgan Kaufmann Publishers, 1996.

[Del91] J. P. Delgrande. Incorporating nonmonotonic reasoning in horn clause theories. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, 1991.

[ELM + 98] T. Eiter, N. Leone, C. Mateis, G. Pfeier, and F. Scarnello. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[FGM + 07] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In J. Marques-Silva and K.A. Sakallah, editors, *Proc. SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354, Berlin Heidelberg, 2007. Springer.

[FH83] F. Fages and G Huet. Complete sets of unifiers and matchers in equational theories. In *Proceedings of the Colloquium on Trees in Algebra and Programming*, 1983.

[FH86] F. Fages and G Huet. Complete sets of unifiers and matchers in equational theories. *Journal of Theoretical Computer Science*, 43:189–200, 1986.

[GHS96] G. Große, S. Hölldobler, and J. Schneeberger. Linear deductive planning. *Journal of Logic and Computation*, 6(2):233–262, 1996.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the International Joint Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Proceedings of the International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

[Göd31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. english translation in [**?**].

[GPP89] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38(1):75–94, 1989.

[GR86] J. H. Gallier and S. Raatz. SLD-resolution methods for Horn clauses with equality based on E-unification. In *Proceedings of the Symposium on Logic Programming*, pages 168–179, 1986.

[Hay73] P. J. Hayes. The frame problem and related problems in artificial intelligence. In A. Elithorn and D. Jones, editors, *Artificial and Human Thinking*, pages 45–49. Jossey-Bass, San Francisco, 1973.

[Hay79] P. J. Hayes. The logic of frames. In Metzing, editor, *Frame Conceptions and Text Understanding*. de Gruyter, Berlin, 1979.

[HL78] G. Huet and D. Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, IRIA, 1978.

[HM86] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 328–333, 1986.

[Höl89a] S. Hölldobler. Combining logic programming and equation solving. Technical report, FG Intellektik, FB Informatik, TH Darmstadt, 1989.

[Höl89b] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. Springer, Berlin, 1989.

[Höl92] S. Hölldobler. On deductive planning and the frame problem. In A. Voronkov, editor, *Proceedings of the Conference on Logic Programming and Automated Reasoning*, pages 13–29. Springer, LNCS, 1992.

[HS90] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.

[HS96] D. Hutter and C. Sengler. INKA: Tth next generation. In *Proceedings of the Conference on Automated Deduction*, 1996.

[HST93] S. Hölldobler, J. Schneeberger, and M. Thielscher. AC1–unification/matching in linear logic programming. In F. Baader, J. Siekmann, and W. Snyder, editors, *Proceedings of the Sixth International Workshop on Unification*. BUCS Tech Report 93-004, Boston University, Computer Science Department, 1993.

[HTT90] J. F. Horty, R. H. Thomason, and D. S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. *Artificial Intelligence*, 42:311–348, 1990.

[HW32] C. Hartshorn and P. Weiss, editors. *Collected Papers of Charles Sanders Peirce*, volume 2. Harvard University Press, 1932.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

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

[KM87] D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.

[Kow91] R.A. Kowalski. Logic programming in artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1991.

[LB87] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93, 1987.

[Lif85] V. Lifschitz. Computing circumscription. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 121–127, 1985.

[Lif86] V. Lifschitz. On the satisfiability of circumscription. *Artificial Intelligence*, 28(1):17–27, 1986.

[Lif87] V. Lifschitz. Pointwise circumscription. In M. Ginsberg, editor, *Nonmonotonic Reasoning*, pages 179–193. Morgan Kaufmann, 1987.

[Lif90] V. Lifschitz. Frames in the space of situations. *Artificial Intelligence*, 46:365–376, 1990.

[Lif99] V. Lifschitz. Answer set planning. In *Proceedings of the International Conference on Logic Programming*, pages 23–37. MIT Press, 1999.

[Llo93] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1993.

[McC63] J. McCarthy. Situations and actions and causal laws. Stanford Artificial Intelligence Project: Memo 2, 1963.

[McC90] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1990.

[MH69] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463 – 502. Edinburgh University Press, 1969.

[Min75] M. L. Minsky. A framework for representing knowledge. In Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, 1975.

[Min82] J. Minker. On indefinite data bases and the closed world assumption. In *Proceedings of the Conference on Automated Deduction*, volume 138 of *LNCS*, pages 292–308. Springer, 1982.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[MT91] W. Marek and M. Truszczyński. Autoepistemic logic. *J. of the ACM*, 38:588–619, 1991.

[Neb90] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[New42] M. H. A. Newman. On theories with a combinatorical definition of 'equivalence'. *Annuals of Mathematics*, 43:223–243, 1942.

[Nie00] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 2000. (to appear).

[NS90] B. Nebel and G. Smolka. Representation and reasoning with attributive descriptions. In K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, pages 112–139. Springer, LNCS 418, 1990.

[NS97] I. Niemelä and P. Simons. Smodels — an implementation of the well–founded and stable model semantics. In *Proceedings of the 4th International Conderence on Logic Programming and Non–monotonic Reasoning*, pages 420–429, 1997.

[Pla93] David A. Plaisted. Equational creasoning and term rewriting system. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 5. Oxford University Press, Oxford, 1993.

[Poo88] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Qui68] R. M. Quillian. Semantic memory. In Minsky, editor, *Semantic Information Processing*, pages 216–270. MIT Press, 1968.

[Rei77] R. Reiter. On closed world data bases. In H. Gallaire and J. M. Nicolas, editors, *Workshop Logic and Databases*, CERT, Toulouse, France, 1977.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132, 1980.

[Rei91] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation — Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence*. Prentice Hall, 1995.

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

[Rob67] J. A. Robinson. A review on automatic theorem proving. In *Annual Symposia in Aplied Mathematics XIX*, pages 1–18. American Mathematical Society, 1967.

[Sch76] L. K. Schubert. Extending the expressive power of semantic networks. *Artificial Intelligence*, 7(2):163–198, 1976.

[Sus75] G. J. Sussman. *A Computer Model of Skill Aquisition*. Elsevier Publishing Company, 1975.

[Wal94] C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford Science Publications, 1994.

[Wei96] C. Weidenbach. *Computational Aspects of a First–Order Logic with Sorts*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1996.

[Woo75] W. A. Woods. What's in a link: Foundations for semantic networks. In D. G. Bobrow and A. M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, 1975.

Index