Rewriting Systems

Lecture Notes

Johan Boye
Jan Małuszyński

September 2003

Contents

Synopsis

The intention of these notes is to give a uniform perspective of some major rewriting formalisms relevant for computing. The first version of the notes has been edited by Johan Boye in the nineties, and they have been subject of substantial revision since then.

A widely adopted view of a computation as a sequence of state transitions becomes a rewriting view if the states are represented as expressions in some language, and possible state transitions are represented as rewrite rules. The concepts and the formalisms presented in these notes have played an important role in development of programming languages. While some comments on the practical relevance are scattered in the text, the focus is on a uniform presentation of various formalisms stressing the relations between them. The objective is to give a general introduction to rewriting techniques. Therefore the presentation of each of the formalisms discussed here is restricted to selected basic concepts. For in-deep studies of particular formalisms the reader is referred to the rich literature on rewriting. The bibliography provides some hints.

The material is organized as follows. The introductory chapter motivates the rewriting view on computation with several examples. This leads to the concept of an *abstract reduction system*, which is just a set equipped with a binary relation. Already at this abstract level, some basic properties of rewriting can be introduced, and some preliminary results established. The concepts presented provide a uniform framework for comparing various rewriting systems discussed in the sequel.

Chapter 2 presents *lambda calculus* and *combinatory logic* as examples of rewriting systems and discusses some relations between them. Both lambda calculus and combinatory logic play a fundamental role in the theory and implementation of programming languages.

Chapter 3 introduces *term rewriting systems* (TRS) as an instance of abstract reduction systems. The combinatory logic is an example of a term rewriting system. Term rewriting systems constitute a formal, non-deterministic model of computation, which is very appealing due to the simple syntax and semantics. The theory of term rewriting has developed very rapidly in the

last decades. Methods and concepts have proved to be very general and relevant also for related areas, like functional programming, logic programming, modern algebra, etc.

Chapter 4 surveys some methods of proving that a TRS is *terminating*, i.e. that every possible rewriting sequence is finite. Although this problem is undecidable in general, termination can still be established in many interesting cases.

Chapter 5 presents the basic theory of equational logic, and shows how TRSs can be used as decision procedures to solve equality problems in equational theories. It is shown how the Knuth-Bendix completion procedure can be used for "compiling" a set E of equational axioms into a TRS which constitutes such a decision procedure.

Chapter 6 discusses the use of TRSs for *unification*, an essential part of any existing automated reasoning process that comprises a non-trivial notion of equality. In particular, unification is a basis for implementation of logic programming languages. The basic notions of unification are presented, and it is shown how TRSs can be used to create a general unification procedure, called *narrowing*. We also give some examples of cases where the narrowing procedure is not applicable, and where radically different unification procedures have been conceived.

Chapter 7 presents the semantics of the equational specifications used for definition of abstract data types. The initial algebra semantics of such specifications is discussed.

The 2003 edition of the course has two objectives:

- to present the basic concepts of rewriting, in particular term rewriting. The concepts, discussed in these lecture notes, will be subject of the lectures and the familiarity with them will be checked by the obligatory homework assignments. The problems of the homework assignments are to be solved individually by each student and delivered by the deadlines, as the necessary examination requirement.

- to survey selected rewriting-based programming languages and systems in the form of seminar presentations prepared by the students and based on web-accessible materials linked from the course web page. (For more details see the course homepage).

Jan Małuszyński
September 2003

Chapter 1

Computation as rewriting

We are going to study computation as a *rewriting* process. The view we will adopt is that computation can be thought of as making transformations in a state space, where one step of computation corresponds to a move from one state to another. When a final state is reached, the computation halts and the result is presented. Describing this a little bit differently, computation could be viewed as successive *rewriting* of *expressions* into other expressions, determined by a set of *rewrite rules*. When no rewrite rule is applicable to an expression, the expression is said to be a *normal form*, and considered to be the result of the computation. This process is nondeterministic in general, since several rewrite rules may be applicable to a given expression, yielding different resulting expressions.

The rewriting view on computation seems particularly appropriate when dealing with problems of the type: "Simplify the expression e as much as possible". For example the polynomial expression $(0+a)(a+b)$ simplifies to a^2+ab, using rules as

$$\begin{aligned}&0+x\to x\\&0\cdot x\to 0\\&x(y+z)\to xy+xz\\&(x+y)z\to xz+yz\end{aligned}$$

Note that there are many possible reductions from the initial expression to the result, two of which are: [1]

$$(0+a)(a+b)\to a(a+b)\to a^2+ab$$

$$(0+a)(a+b)\to(0+a)a+(0+a)b\to(0\cdot a+a^2)+(0+a)b\to(0+a^2)+(0+a)b\to a^2+(0+a)b\to a^2+(0\cdot b+ab)\to a^2+(0+ab)\to a^2+ab$$

[1]We assume here that the operators have the usual priorities so that the superfluous parantheses in the expression $a^2+(ab)$ can be omitted.

For the expression considered (i) all reductions are finite, and (ii) the result will always be the same, no matter how the reduction is carried out. If these two desirable and important properties holds for any expression, the rewrite rules are called (i) *terminating* and (ii) *confluent*. However, these properties are in no way guaranteed to hold for an arbitrary set of rewrite rules. Even our example rules are not confluent, since the expression $x(0+y)$ can be simplified either to xy or to $x\cdot 0+xy$, depending on the choice of the used rules.

A major theme of this text will in fact be to study how we can define rewriting systems that are terminating and confluent, and how we can establish that a given system has these properties.

Below we give more examples from mathematics and computer science using the rewriting view on computation. The examples will be frequently used later on, and serve to illustrate some key concepts which will be discussed in subsequent chapters. The examples do not include lambda calculus and combinatory logic, which will be discussed in a separate chapter.

1.1 Simplification of algebraic expressions

We have already mentioned simplification of polynomials. In the univariate (single variable) case, every polynomial can be simplified to the well-known form

$$a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0 \tag{1.1}$$

where the a_i's are, say, rational numbers. This simplification can be carried out using rewrite rules like the ones mentioned previously.

There is a similar normal form for multivariate (multiple variable) polynomials, which can be reached with the same set of rewrite rules as in the univariate case. For simplification of *rational* expressions (i.e. algebraic expressions involving division) like $\frac{x+1}{x^2}/(x-1)$, some additional rewrite rules are needed.

The emergence of computer algebra systems has rendered it interesting to find algorithmic methods to simplify more complicated algebraic expressions. *Unnested radical* expressions like $(x+3)^{-2/3}/(x^2-x-1)^{5/8}-(3-x^8)^{-1/2}$ can be simplified using an algorithm by Caviness and Fateman [7]. *Transcendental* expressions (involving functions like sin and log) pose the hardest problems, and for some classes of such expressions the simplification problem is algorithmically unsolvable. See Buchberger and Loos [6] for a nice survey on algebraic and transcendental simplification techniques.

1.2 Turing machines

The Turing machine (TM), conceived by Alan Turing in 1936 [68], is a simple but powerful mathematical model of a computational device. A TM has a finite set Q of states, and a storage device consisting of a one-dimensional array (a *tape*) of cells, each of which can contain a symbol out of a finite set Σ, that includes a distinguished *blank symbol*. Associated with the tape is a read/write head that can move right or left on the tape and that can read and write a single symbol on each move. The tape is infinitely long in both directions.

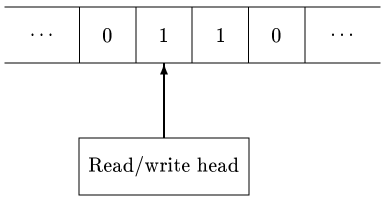


Figure 1.1: A snapshot of a Turing machine at work

When the computation starts, the TM is in its *initial state*, the tape contains a finite string of non-blank symbols placed in consequtive cells, while all other cells contain blank symbols, and the head is scanning the leftmost symbol of the string. The machine's actions are determined by its program, which is given as a partial function δ from $Q \times S$ to $Q \times S \times \{\mathit{left}, \mathit{right}\}$. Thus if the machine is in state q, and the read/write head is scanning the symbol s, then the program specifies the next state, the symbol to repalce s on the tape, and the direction in which to move the read/write head. If $\delta(q, s)$ is undefined, then the machine halts in an error state. If a *final state* is reached, the computation terminates and the contents of the tape is the result of the computation. Notice that at every step of a computation only a finite number of cells contains non-blank symbols; the contents of the tape may thus be described by a finite string.

At any given moment, the TM's configuration is described by the contents on the tape and the current state. Suppose that the contents of the tape is $s_1 \ldots s_{i-1} s_i \ldots s_n$, the read/write head is scanning the symbol s_i, and the TM is in state q. Then we let the string

$$s_1 \ldots s_{i-1}\; q\; s_i \ldots s_n$$

represent the configuration of the machine. Now if the program contains the instruction $\delta(q, s_i) = (q', t, \mathit{left})$, then the machine may change its configuration according to the rewrite rule

$$s_1 \ldots s_{i-1}\; q\; s_i \ldots s_n \quad\to\quad s_1 \ldots s_{i-2}\; q'\; s_{i-1}\; t \ldots s_n$$

If the program contains the instruction $\delta(q, s_i) = (q', t, \mathit{right})$, then the corresponding rewrite rule is

$$s_1 \ldots s_{i-1}\; q\; s_i \ldots s_n \quad\to\quad s_1 \ldots s_{i-1}\; t\; q'\; s_{i+1} \ldots s_n$$

A well-known result is that the class of (partial) functions which can be computed with Turing machines (the *partial recursive* functions) exactly coincides with the class of functions which can be defined by the lambda expressions discussed in Chapter 2.

1.3 Context-free grammars

Context-free grammars play a fundamental role in string-processing applications, notably in the definition of programming languages, parsing of programming languages and natural languages, etc. The origin of the formalism can be found in Chomsky [9].

A context-free grammar is a specification of a (context-free) language (over some alphabet Σ) by means of *production rules* of the form:

$$A \to \alpha$$

where A belongs to a set V of *non-terminals*, and α is a string in $(\Sigma \cup V)^*$. We assume that V and Σ are disjoint.

Suppose β_1 and β_2 are (possibly empty) strings in $(\Sigma \cup V)^*$. Then $\beta_1 A \beta_2$ *derives* $\beta_1 \alpha \beta_2$, using the production rule above. We write this as

$$\beta_1 A \beta_2 \quad\Rightarrow\quad \beta_1 \alpha \beta_2$$

If it is possible to reach a string $\gamma \in \Sigma^*$, starting from the special *start non-terminal* S, and by using a sequence of derivation steps, then γ belongs to the language generated by the grammar. As an example, we give a grammar that generates all strings over $\{a, b\}^*$ with an equal number of a's and b's. Here S is the only non-terminal, and ϵ denotes the empty string.

$$\begin{aligned} S &\to SS \\ S &\to \epsilon \\ S &\to aSb \\ S &\to bSa \end{aligned}$$

For instance, the string *abba* can be derived as follows:

$$S \;\Rightarrow\; SS \;\Rightarrow\; aSbS \;\Rightarrow\; abS \;\Rightarrow\; abbSa \;\Rightarrow\; abba$$

We have used the first, third, second, fourth and second rules of the grammar in the different steps.

1.4 Logic programs

A definite logic program is a set of *Horn clauses* of the form

$$H \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$$

where H, B_1, ..., B_m are atomic formulae. All variables of each program clause are universally quantified.

A computation of definite program P starts with *query* q of the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

and attempts to find a *substitution* σ i.e. a mapping from variables to terms; see section 3.3), such that $\sigma(q)$[2] is a logical consequence of P. This is based on the *SLD-resolution* (see e.g. [55]) where the clauses of the program can be seen as rewrite rules for queries. A query

$$A_1 \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_n$$

may be rewritten to the query

$$\sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge B_1 \wedge \ldots \wedge B_m \wedge A_{i+1} \wedge \ldots \wedge A_n)$$

if $H \leftarrow B_1 \wedge \ldots \wedge B_m$ is a clause of the program with variables renamed to "fresh variables" that do not appear in the previous queries in the computation, and σ is a *most general unifier* of A_i and H (see definition 3.9).
For example, the clause $grandparent(X,Y) \leftarrow parent(X,Z), parent(Z,Y)$ can be used to obtain:
$grandparent(P, mary) \rightarrow parent(P,Z), parent(Z, mary)$

1.5 Abstract reduction systems

An abstraction based on the examples of the preceding sections gives the following definition:

[2] $\sigma(q)$ is sometimes denoted $q\sigma$

Definition 1.1 An *abstract reduction system (*ARS*)* is a pair $(A, \{\rightarrow_i\}_{i \in I})$, where A is a set, and each $\rightarrow_i$ is a binary relation on A, called a *rewrite relation.* □

Often an ARS has only one rewrite relation (in which case we denote the ARS by $(A, \rightarrow)$).

Notation: If $a, b \in A$ and $(a, b) \in \rightarrow$, then we write this as $a \rightarrow b$ and call b a *reduct* of a. The reflexive and transitive closure of $\rightarrow$ is written as $\twoheadrightarrow$ (a usual notation R^* for the transitive and reflexive closure of the relation R may also be used). $\rightarrow^i$ denotes reduction in i steps. The inverse relation of $\rightarrow$ ($\twoheadrightarrow$) is denoted by $\leftarrow$ or $\rightarrow^{-1}$ ($\twoheadleftarrow$ or $\twoheadrightarrow^{-1}$). If a and b are identical (i.e. they denote the same element in A), we write $a \equiv b$. a and b are *convertible* ($a = b$) if a can be rewritten into b using the $\rightarrow$ relation forwards or backwards a finite number of times. Thus, we let $=$ be the relation $(\twoheadrightarrow \cup \twoheadleftarrow)^*$. □

Figure 1.2 shows an example rewrite relation, in which a and b are convertible.

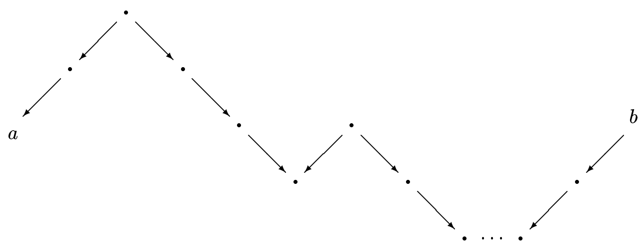


Figure 1.2: $a = b$

Definition 1.2 Given an ARS $(A, \rightarrow)$, $a \in A$ *is* a *normal form* if there is no b such that $a \rightarrow b$. We write $NF(A)$ to denote the set of normal forms in A (w.r.t. $\rightarrow$). a *has* a normal form if there is a $b \in NF(A)$ such that $a \twoheadrightarrow b$ (we say that b is a normal form of a). □

Note that an element may have zero, one, or many normal forms.

The operational intuition behind the relation(s) $\rightarrow$ is that if $a \rightarrow b$, then a can be *reduced to* b. As shown by the examples in the previous chapter, our aim is often to reduce (in a number of steps) a given element to (one of) its normal forms, using the rewrite relation. This naturally raises questions about existence and uniqueness of normal forms; questions we will address in the next section.

1.6 Properties of ARSs

Throughout this section, we assume that an ARS $(A, \to)$ is given.

Definition 1.3

(1) $\to$ is *weakly normalizing (WN)* if every $a \in A$ has a normal form.

(2) $\to$ is *strongly normalizing (SN)* (or *terminating*, or *Noetherian*) if there is no infinite sequence $a_0 \to a_1 \to a_2 \to \dots$

□

We will use the terms SN, strongly normalizing and terminating without preference.

If $\to$ is weakly normalizing, then we can reduce every element to a normal form. Practically however, the computation might not terminate if the wrong reduction strategy is used. In the example below, the reduction sequence $a_0 \to a_1 \to \dots$ is infinite.

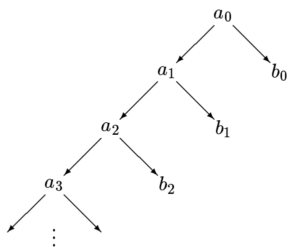


Figure 1.3: A rewrite relation which is WN but not SN

If $\to$ is terminating, then every reduction sequence will eventually end in a normal form. If one is interested in finding *all* normal forms to a given element, it will be necessary to backtrack (or to use a breadth-first search strategy) in the general case. However, if $\to$ is *confluent*, this will not be needed. We next discuss the confluence property.

Definition 1.4

(1) $\to$ is *Church-Rosser (CR)* (or *confluent*) if for all $a, b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

(2) $\to$ is *weakly Church-Rosser (WCR)* (or *weakly confluent*) if for all $a, b, c \in A$ such that $a \to b$ and $a \to c$, there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

□

We will use the terms CR, Church-Rosser and confluent without preference (the same goes for WCR, etc.). The above definition is probably easiest understood in diagram format (see Fig. 1.4).

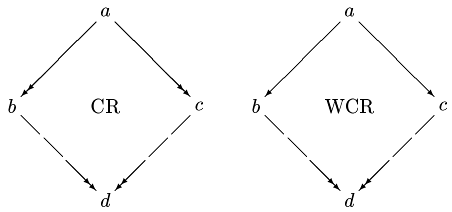


Figure 1.4: Schematic description of the CR and WCR properties (dashed arrows are existentially quantified)

Theorem 1.5 If $\to$ is confluent, then the normal form of every element (if it exists) is unique (i.e. if $a \twoheadrightarrow b$, $a \twoheadrightarrow c$, and $b, c \in NF(A)$, then $b \equiv c$).

Proof : See exercise 2. □

The distinction between confluence and weak confluence is subtle but important. The following example shows the difference.

Example 1.6 Consider the rewrite relation defined by the following diagram:
By a simple case analysis we see that $\to$ is weakly confluent. For instance, b can be reduced to both a and c, but $a \twoheadrightarrow a$ and $c \twoheadrightarrow a$. However, $\to$ is not confluent, since $b \twoheadrightarrow a$ and $b \twoheadrightarrow d$, and a and d have no common reduct. □

For terminating rewrite relations, confluence and weak confluence are equivalent. This is a consequence of the following fundamental theorem, first stated by Newman [54]. Later Huet significantly simplified the proof [32]. Below we give Huet's proof.

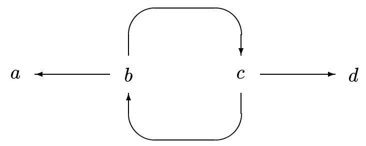


Figure 1.5: A rewrite relation which is WCR but not CR

Theorem 1.7 **[Newman's lemma]** If $\rightarrow$ is weakly confluent and terminating, then $\rightarrow$ is confluent.

Proof : Choose an arbitrary element a. We want to prove

$$P(a)\ :\ \forall b,c : a \twoheadrightarrow b \ \text{ and } \ a \twoheadrightarrow c \ \Rightarrow\ \exists d(b \twoheadrightarrow d \ \text{ and } \ c \twoheadrightarrow d)$$

Since $\rightarrow$ is terminating, we may proceed by induction [3]. Let $Red(a)$ be the set of reducts of a, i.e. $\{b \mid a \rightarrow b\}$, and suppose that $P(b)$ holds for every $b \in Red(a)$. Let $a \rightarrow^i y$ and $a \rightarrow^j z$. We show that $\exists t : y \twoheadrightarrow t$ and $z \twoheadrightarrow t$.

(i) If $i = 0$, choose $t = z$; if $j = 0$, choose $t = y$.

(ii) Otherwise, let $a \rightarrow y_1 \twoheadrightarrow y$ and $a \rightarrow z_1 \twoheadrightarrow z$.

By weak confluence, $\exists u : y_1 \twoheadrightarrow u$ and $z_1 \twoheadrightarrow u$. By the induction hypothesis $P(y_1)$, $\exists v : y \twoheadrightarrow v$ and $u \twoheadrightarrow v$. By the induction hypothesis $P(z_1)$, $\exists t : v \twoheadrightarrow t$ and $z \twoheadrightarrow t$, thereby proving $P(a)$.

□

If $\rightarrow$ is confluent, then normal forms are unique (according to theorem 1.5). This means that $\rightarrow$ defines a partial function f on A, namely:

$$f(a) = \begin{cases} b, & \text{if } b \in NF(A) \text{ and } a \twoheadrightarrow b \\ \text{undefined else} & \end{cases}$$

If $\rightarrow$ is also weakly normalizing, then f is a total function. The most interesting case is when $\rightarrow$ is both confluent and terminating, in which case $f(a)$ can be computed with any reduction strategy.

Definition 1.8 $\rightarrow$ is *complete* if $\rightarrow$ is confluent and terminating. □

Complete rewrite relations are also called *canonical*, or *convergent* rewrite relations in the literature. We will study them extensively in the following.

[3] The induction principle used here is known as *well-founded induction* (called *Noetherian induction* by Huet [32] and Cohn [12], and *R-induction* by van Dalen *et al.* [15]).

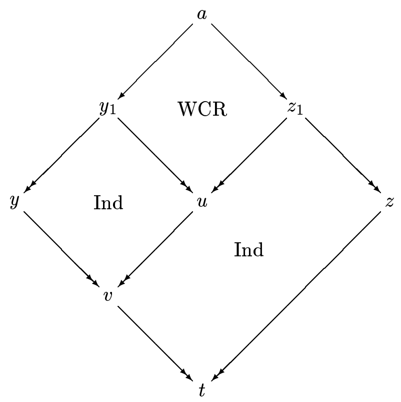


Figure 1.6: The induction step in the proof of Newman's lemma

Exercises

1. Formulate the examples in sections 1.1 through 1.4 as abstract reduction systems $(A_1, \rightarrow_1), \ldots, (A_5, \rightarrow_5)$. Identify the normal forms in $A_1, \ldots A_5$. Determine which rewrite relations:
   (a) are terminating.
   (b) are confluent.
   (c) have unique normal forms.
   (d) are deterministic.
2. Prove theorem 1.5.

Chapter 2

Lambda calculus and combinatory logic

This chapter discusses a rewriting system called *lambda calculus*. It was introduced in the 1930's by Church [10]. The system is of fundamental importance for computation theory since it gives a natural formalisation of the notion of a computable function. This notion turns out to be equivalent in expressive power to the concept of a Turing machine but appears to be much closer to real programming languages. For example, various mechanisms of procedure calling can be discussed in this framework.

2.1 Lambda terms

As any rewrite system the lambda calculus defines a domain of expressions and a rewrite relation on this domain. The expressions are called *lambda terms*. Intuitively, lambda terms represent functions on lambda terms. This may seem somewhat confusing, since one might have expected first a domain of elementary values on which an hierarchy of first- and higher-order functions could be defined. However, as will be discussed later, the elementary values such as natural numbers can easily be coded as functions.

The functional intuition is reflected by the syntax of lambda terms. The basic lambda term is a variable. The role of variables in lambda calculus can be compared to the role of the variables used in procedures of programming languages. Certain kinds of expressions in programming languges, e.g. arithmetic expressions like $x+y$ can be transformed into procedures by identifying some of the variables as formal parameters.

For example, in the context where y is a known integer the declaration

```
int sumy(x)
int x;
{ x+y }
```

transforms the arithmetic expression $x+y$ into the unary arithmetic procedure `sumy` where the variable x is used as formal parameter, while the variable y is global. A similar kind of syntactic construct is defined for lambda terms and it is called *functional abstraction*.

In a program the procedure `sumy` is *called* with an argument, represented typically by an arithmetic expression, e.g `sumy(2)`. A similar kind of syntactic construct is defined for lambda terms and it is called *functional application*.

Formally, the lambda terms are defined as follows:

(i) a variable $x, y, z, \ldots$ is a lambda term;

(ii) if M and N are lambda terms, then (MN) is a lambda term, (called *functional application*);

(iii) if M is a lambda term, then $(\lambda x.M)$ is a lambda term (called *functional abstraction*).

A term of the form (ii) represents the application of a function M to N. A term of the form (iii) makes the term M into a unary function; $(\lambda x.M)$ represents the function of one variable x.

A commonly used notational convention allows for suppression of some parantheses. Application is understood to associate to the left, so we can write xyz instead of $((xy)z)$. The notation $\lambda x.PQ$ is used instead of $(\lambda x.(PQ))$ and $\lambda x_1x_2...x_n.M$ instead of $(\lambda x_1.(\lambda x_2.(...(\lambda x_n.M)...))$.

2.2 The rewrite rules

In a lambda term of the form $\lambda x.M$, every occurrence of x in M is said to be *bound*. If an occurrence of x is not bound in M, then it is said to be *free* in M.

The rewrite rules of lambda calculus make an essential distinction between free and bound variable occurrences. Intuitively the bound occurrences play a role similar to that of the formal parameters of procedures in programming languages, while the free occurrences can be compared to global variables in the procedure. The essential mechanism used in rewriting is the operation of simultaneous replacement of all free occurrences of a variable x in a term M by a term N. The operation should be defined with care since it should not affect the status of the occurrences of the remaining variables. In particular, free occurrences should not be changed to bound occurrences. Such a danger exists if M and N share some variables. Problem can be avoided by renaming some of the bound variables in the case of conflicts.

The commonly adopted solution can be presented as the following recursive definition of the replacement operation. We define $[x \mapsto N]M$ to be the lambda term obtained from M as follows:

(i) $[x \mapsto N]x$ is N,

(ii) $[x \mapsto N]y$ is y for any variable y different from x,

(iii) $[x \mapsto N](PQ)$ is $([x \mapsto N]P[x \mapsto N]Q$,

(iv) $[x \mapsto N]\lambda x.P$ is $\lambda x.P$,

(v) $[x \mapsto N]\lambda y.P$ is $\lambda y.[x \mapsto N]P$ if y is different from x and has no free occurrence in N, or if x has no free occurrence in P,

(vi) $[x \mapsto N]\lambda y.P$ is $\lambda z.[x \mapsto N([x \mapsto z]P)$ if y is different from x and has a free occurrence in N, and x x has a free occurrence in P,

The cases (i) and (ii) above are basic ones, when no conflict is possible. Replacement in the term which is a functional application is defined by case (iii). In that case the replacement of x is to be made independently in both component terms. The difficulties may only concern functional abstraction, which is handled by the cases (iv), (v) and (vi). Intuitively, the replacement concerns only the free occurrences of x in M. If M is of the form $\lambda x.P$ (case (iv)), there is no free occurrence of x in M hence the operation leaves M unchanged. Case (v) identifies the conflict-free situations. Replacing x by N in $\lambda y.P$ would have changed the status of free occurrences of y in N. This cannot happen if y does not appear free in N. On the other hand, if x does not appear free in P the replacement will not change P, hence will not change the status of any variable occurrence.

The problem may only arise if y occurs free in N and x occurs free in P. For example take the term y as N and the term xy as P. In that case M is of the form $\lambda y.xy$. A direct replacement of x by N in M would have resulted in the term $\lambda y.yy$, thus changing the status of y in N. To avoid this, case (vi) requires that the bound variable in M is renamed, e.g. to z. The effect of the operation is thus the term $\lambda z.yz$, and the status of y in N remains unchanged after the replacement.

The rewrite relations of lambda calculus are defined by the α- and β-*rules*:

(α) If y is not free in M, then $\lambda x.M \ \rightarrow_\alpha \ \lambda y.[x \mapsto y]M$.

(β) $(\lambda x.M)N \ \rightarrow_\beta \ [x \mapsto N]M$

The intuition of the α-rule is that the name of a bound variable is irrelevant. We have seen already that renaming is used in the definition of the replacement operation. The restriction in the α-rule requires that "fresh" variables are used for renaming to avoid confusion with the free variables already present in the term, which in that case would become bound by this transformation. For example, the variable x in the term $\lambda x.xy$ cannot be renamed to y. It should be noticed that the rewrite relation $\twoheadrightarrow_\alpha$ is symmetric. This means that one need not to distinguish between "forward" and "backward" application of the α-rule. Any "backward" rewriting of a given term can also be achieved by a "forward" rewriting. For this reason the α rule is called *α-conversion rule* in accordance with the terminology of Section 1.5. The lambda terms X and Y such that $X \twoheadrightarrow_\alpha Y$ are called *α-congruent* terms.

The β-rule captures the intuition of computing by reduction. A redex for this rule have the form of functional application, where the first component is a functional abstraction, representing the function applied and the other is the actual argument. Intuitively, such redexes resemble procedural calls in programming languages. The β-reduction can be compared with invocation of a unary procedure. The computation step consists in removing the procedure header and replacing every occurrence of the (only) formal parameter by the actual parameter of the call. As a result a new lambda term is obtained which may also include redexes.

This process may or may not terminate. The normal form obtained upon termination can be seen as the value of the function for the actual argument. Generally, a lambda term can have more than one redex, and can thus be subject to more than one reduction strategy. For the above intuition to be valid, a terminating computation in any strategy should give the same result. Otherwise, the term having different normal forms could not be considered as a function.

Note that the formal definition of β-reduction uses the replacement operation, which in certain cases may cause renaming. Therefore the result of the β-reduction is only unique up to renaming of the bound variables.

2.3 Rewrite strategies and confluence

The two most important strategies are *normal-order* reduction, where the leftmost redex is chosen, and *applicative-order* reduction, where the leftmost innermost redex is chosen. Consider a lambda term which is a redex. As discussed above, such a term resembles a call of a procedure in a programming language. The normal-order reduction step will thus replace every occurrence of the formal parameter in the body by the (unchanged) actual parameter.

But the actual parameter may be/include another redex. Thus, under this strategy the computation of the value of the actual parameter is postponed. This resembles the mechanism of procedure invocation known as "call-by-name" or "lazy evaluation", since the computation of the argument is deferred as long as possible. The situation is different in the applicative-order strategy. Here the normal forms of the nested redexes are computed starting from the innermost level. This resembles the "call-by-value" mechanism of procedure invocation where the values of the actual parameters are computed first.

Consider for example the term $(\lambda x.xx)((\lambda y.y)(\lambda y.y))$. Its normal-order reduction goes as follows:

$$
\begin{array}{l}
(\lambda x.xx)((\lambda y.y)(\lambda y.y)) \to \\
((\lambda y.y)(\lambda y.y))((\lambda y.y)(\lambda y.y)) \to \\
(\lambda y.y)((\lambda y.y)(\lambda y.y)) \to \\
((\lambda y.y)(\lambda y.y)) \to \\
\lambda y.y
\end{array}
$$

On the other hand, the applicative-order reduction requires fewer steps to obtain the same normal form :

$$
\begin{array}{l}
(\lambda x.xx)((\lambda y.y)(\lambda y.y)) \to \\
(\lambda x.xx)(\lambda y.y) \to \\
((\lambda y.y)(\lambda y.y)) \to \\
\lambda y.y
\end{array}
$$

The famous *Church-Rosser theorem* states confluence of lambda calculus. More precisely, if $P \twoheadrightarrow_\beta M$ and $P \twoheadrightarrow_\beta N$ then there exist α-congruent terms T_1 and T_2 such that $M \twoheadrightarrow_\beta T_1$ and $N \twoheadrightarrow_\beta T_2$. Consequently, every lambda term has a unique (up to α-conversion) normal form, or no normal form at all. This justifies the intuition of viewing lambda terms as partial functions. Applying a lambda term to an argument results in at most one (up to renaming) normal form. The function is undefined on the arguments for which no normal form exists.

Termination may depend on the reduction strategy. Consider for example the term Y of the form $(\lambda y.yy)(\lambda y.yy)$. It has only one β-redex and the reduction step rewrites the term into itself. Hence Y has no normal form. Consider also the term Z of the form $\lambda x.\lambda z.z$. Application of Z to any term T can be β-reduced in one step to the term $\lambda z.z$ regardless of the form of T since x does not appear in $\lambda z.z$. Consider now the term ZY. it has two redexes: ZY and Y. The normal-order reduction will in one step produce the normal form $\lambda z.z$. On the other hand, the applicative-order reduction will keep reducing Y to Y in ZY, hence it will not terminate.

An important result is that if the normal form exists, it is reachable by a finite normal-order reduction sequence.

2.4 The power of lambda calculus

This section shows that both natural numbers and functions on natural numbers can be represented as lambda terms.

Church proposed to represent a natural number n by the term $\lambda xy.x^n y$, where $x^n y$ denotes n applications of x to y, i.e. the term $x(x...(xy)$ including n ocurrences of x. Such terms are called *Church numerals*. In the sequel we will abbreviate $\lambda xy.y$ as $\mathbf{0}$, $\lambda xy.xy$ as $\mathbf{1}$, etc. Notice that a Church numeral is in normal form. It can also be seen as a binary function on lambda terms: for given terms M and N the term $\mathbf{n}MN$ reduces to $M^n N$.

An m-ary function ϕ on natural numbers is said to be *lambda-defined* by a lambda term F iff for arbitrary natural numbers $n_1, ..., n_m$

$$F\mathbf{n_1} \ ... \ \mathbf{n_m} \twoheadrightarrow \mathbf{n} \text{ iff } \phi(n_1, ..., n_m) = n.$$

A function is *lambda-definable* if it is lambda-defined by some lambda term.

A classical result is that the class of lambda-definable functions is exactly the class of partial recursive functions. Thus, due to a known result, lambda-definable functions coincide also with the functions defined by Turing machines. Thus, lambda calculus can be used as a foundation of computation theory.

On the other hand, some concepts of lambda calculus turn out to be useful in the context of programming languages. As already pointed out, the β-reduction mechanism resembles parameter passing in programming languages. Another interesting observation is that all functions represented by lambda terms have fixed points. More precisely, for every term M there exists a term P such that $MP =_\beta P$. Moreover, there exists a lambda term Y such that YM is a fixpoint of M for every M. In particular consider Y defined as $\lambda x.((\lambda y.x(yy))(\lambda y.x(yy)))$. Then for any M

$$
\begin{array}{l}
YM = \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))M \to \\
(\lambda y.M(yy))(\lambda y.M(yy)) \to \\
M((\lambda y.M(yy))(\lambda y.M(yy))) = M(YM)
\end{array}
$$

Hence $M(YM) =_\beta YM$. Thus YM is a fixpoint of M. This has some relevance for the theory of programming languages, where semantics is often defined in terms of fixpoints.

In the pure version of lambda calculus discussed here, the functions are defined from very few primitive concepts. For this reason, even relatively simple functions can only be coded by rather complex lambda terms, and computation of their values requires a large number of reduction steps. In this sense, the expressive power of pure lambda calculus is rather low. We illustrate this statement by an example.

The first step towards definition of the arithmetic operations is to define the successor function s by a lambda term $\mathbf{s}$ such that $\mathbf{sn} \twoheadrightarrow \mathbf{n+1}$. A solution proposed in the literature is to define $\mathbf{s}$ as $\lambda uxy.x(uxy)$. We get:

$$\begin{aligned}&\mathbf{sn} = \lambda uxy.x(uxy)(\lambda zv.z^n v) \to\\&\lambda xy.x((\lambda zv.z^n v)xy) \to\\&\lambda xy.x((\lambda v.x^n v)y) \to\\&\lambda xy.x(x^n y) = \mathbf{n+1}\end{aligned}$$

Thus, computation of the successor is not too complicated. Now the addition can be expressed in terms of the successor. Knowing that $\mathbf{n}xy \twoheadrightarrow x^n y$, and that $x+y = s^x s^y(0)$ we can define addition by the lambda term $\lambda xy.x\mathbf{s}(y\mathbf{s0})$.

In the extended form it is:

$$\lambda xy.x(\lambda uxy.x(uxy)(y(\lambda uxy.x(uxy))(\lambda xy.y)).$$

This term is indeed not that easy to grasp and following the reduction steps needed to compute 2+2 would definitely be beyond one's patience.

Another way of defining addition is by the following recursive definition

$$\begin{aligned}&plus(0,y) = y\\&plus(s(x),y) = s(plus(x,y))\end{aligned}$$

This is a special case of the use of the *primitive recursion* scheme

$$\begin{aligned}&\phi(0,y) = \psi(y)\\&\phi(s(x),y) = \theta(x,\phi(x,y))\end{aligned}$$

defining a function ϕ on natural numbers in terms of given functions ψ and θ. It turns out that such a scheme can be represented by a lambda term, which is however quite complex. We refer for details e.g. to [28].

2.5 Combinatory logic

This section presents yet another rewriting system, called the *combinatory logic*. A motivation behind it is to represent functions without the technical difficulties related to the use of bound variables in lambda calculus. The combinatory logic has only free variables, so that it does not suffer from these difficulties. Still it has the computational power equivalent to that of lambda calculus. However, it does not seem to have as clear intuition as the lambda calculus. The combinatory logic plays an important role in the implementation of functional languages. This topic is, however, outside of the scope of these notes. Our direct motivation for presenting combinatory logic is that it is an interesting example of a general class of rewriting systems, to be discussed in Chapter 3.

Combinatory terms

We define first the domain of the rewriting system. Its elements will be called *combinatory logic terms*, or briefly *combinatory terms*.

They are constructed by *application* from the basic elements which are the variables and the two *basic combinators* K and S. Thus, formally the combinatory terms are defined as follows:

(i) A variable is a combinatory term,

(ii) The basic combinators K and S are combinatory terms,

(iii) If X and Y are combinatory terms, then (XY) is a combinatory term.

For example, $((SK)K)$ and $(((SK)K)x)$ are combinatory terms. A commonly used notation is to omit parantheses following the convention of association to the left. For example, the term $((SS)(K((SK)K)))$ will be represented as $SS(K(SKK))$

A combinatory term not including variables is called a *combinator*. The intuition of combinators is to represent operators on functions, e.g. composition of functions, identity operator, etc. Formally the transformation corresponding to such an operator is achieved by reduction of combinator terms using the rewrite rules of the system. According to the definition, any combinator is constructed from the basic combinators K and S. The examples in the next section show that even operators with simple intuition may require quite complicated combinators for their representation.

The rewrite relation

We now define the rewrite relation on combinatory terms.

Any combinatory term of the form KXY or $SXYZ$, where X, Y and Z are arbitrary terms, is called a *redex*. The *contractum* of a redex is defined as:

- X for a redex of the form KXY , and
- $XZ(YZ)$ for a redex of the form $SXYZ$.

The rewrite relation $\to$ on combinatory terms is defined as follows: $P \to Q$ iff Q can be obtained by replacing one occurrence of a redex in P by its contractum. For example, $SKKy \to Ky(Ky) \to y$.

This example shows that the combinator SKK behaves as identity operator on combinatory terms. Its application to an arbitrary term Y reduces to Y. Thus SKK is an important combinator, with a very simple intuition. It is often denoted by I. Still its definition is relatively complicated.

The intuition of the basic combinators K and S is not as simple as that of I. K forms constant functions: application of K to a term P is a function returning P for every argument, since $(KP)x \to P$. S is a function composition operator. Given the arguments X, Y, Z it applies the function XZ to the function YZ. This kind of composition does not seem very natural. On the other hand, the combinator corresponding to the usual composition of functions is more complicated. It has the form $S(KS)K$ and it is denoted B. Indeed, for every X, Y, Z the application $BXYZ$ reduces to $X(YZ)$:

$$
\begin{aligned}
&BXYZ = S(KS)KXYZ \to \\
&(KS)X(KX)YZ \to \\
&S(KX)YZ \to \\
&(KX)Z(YZ) \to \\
&X(YZ)
\end{aligned}
$$

Thus the result is the application of X to the (result of) application of Y to Z. This corresponds to the usual intuition of composition of functions.

An important result about the rewrite system of combinatory logic is the Church-Rosser theorem, analogous to that for lambda calculus, stating confluence of this rewrite system. In contrast to the lambda calculus case, the proof of confluence of the combinatory logic is relatively simple. In particular, the confluence of the combinatory logic follows directly from more general results discussed in Chapter 3.

Another important result analogous to that for lambda calculus states that if a combinatory term has a normal form, then the reduction under the normal strategy terminates. Due to the confluence of the system, such a reduction results in this normal form, which is unique.

Relating combinatory logic to lambda calculus

As mentioned above, both lambda calculus and combinatory logic are intended to represent and compute functions. The question is then how the formalisms are related to each other. A part of the answer can be obtained by studying the "compilation " of combinatory terms into lambda terms, called in the literature the transformation λ. As discussed below, this transformation preserves the rewrite relation.

The transformation λ associates to each combinatory term X a lambda term X_λ defined as follows, by induction on the structure of X:

- $y_\lambda = y$, for every variable y,
- $K_\lambda = \lambda xy.x$, $S_\lambda = \lambda xyz.xz(yz)$,
- $(YZ)_\lambda = Y_\lambda Z_\lambda$ for arbitrary combinatory terms Y and Z.

Thus, for example,

$$
\begin{aligned}
&(SKK)_\lambda = (SK)_\lambda K_\lambda = S_\lambda K_\lambda K_\lambda = \\
&\lambda xyz.xz(yz)(\lambda xy.x)(\lambda xy.x) \to \lambda yz.(\lambda xy.x)z(yz)\lambda xy.x \to \\
&\lambda yz.(\lambda y.z)(yz)\lambda xy.x \to \lambda yz.z\lambda xy.x \to \lambda z.z
\end{aligned}
$$

As illustrated by the above example, the transformation preserves the "nature" of the function. Both SKK and $\lambda z.z$ represent the identity function. For arbitrary combinatory term X the application $SKKX$ reduces to X in combinatory logic. On the other hand, for arbitrary lambda term t the application $(\lambda z.z)t$ β-reduces to t. Notice that the transformation preserves the free variables of the term transformed. Thus, the combinators are transformed to lambda terms without free variables. Such lambda terms are sometimes also called combinators (of lambda calculus).

Generally, the transformation λ preserves the rewrite relation. Formally, if Y can be obtained from a combinatory term X by a number of reduction steps in the combinatory logic then the lambda term Y_λ can be obtained by a number of β-reduction steps from X_λ.

A related question, how to compile lambda terms into combinatory terms, has also been discussed in the literature. One of the proposed solutions, a transformation denoted H will be described now.

The first step concerns simulation of functional abstraction in combinatory terms. For every variable x and for every combinatory term M we construct a combinatory term denoted $\lambda^c x.M$. The intention is that combinatory rewriting of $(\lambda^c x.MN)$ would replace the occurrences of x in M by N. In this way β-reduction would be simulated by rewriting of combinatory terms.

One can check that the goal can be achieved with the following definition:

- $\lambda^c x.x = SKK$
- $\lambda^c x.P = KP$ if x does not appear in P,
- $\lambda^c x.PQ = S(\lambda^c x.P)(\lambda^c x.Q)$ if the previous case do not apply.

For example,

$$\lambda^c x.Kx = S(\lambda^c x.K)(\lambda^c x.x) = S(KK)(SKK).$$

Now, for any term P,

$$S(KK)(SKK)P \to (KKP)(SKKP) \to KP$$

The same term can be otained by replacement of x by P in Kx.

The transformation H is now defined as follows:

- $x_H = x$ for every variable x,

- $(PQ)_H = P_H Q_H$ for any lambda terms P and Q,
- $(\lambda x.P)_H = \lambda^c x.(P)_H$

For example:

$$(\lambda xy.y)_H = \lambda^c x.(\lambda y.y)_H = \lambda^c x.(\lambda^c y.y) = \lambda^c x.SKK = K(SKK)$$

The transformation λ discussed above is a kind of inverse of H, since for every lambda term M the term $M_{H\lambda}$ is convertible to M (in lambda calculus). For example:

$$(\lambda x.x)_{H\lambda} = (SKK)_\lambda = \lambda xyz.xz(yz)(\lambda xy.x)(\lambda xy.x) \twoheadrightarrow_\beta \lambda z.z =_\alpha \lambda x.x$$

One can compile a lambda term M to the combinatory term M_H, reduce it, and transform the result R of the reduction to the lambda term R_λ. As mentioned above, the transformation λ preserves the rewrite relation. Hence $M_{H,\lambda} \twoheadrightarrow_\beta R_\lambda$. This may motivate the use of combinatory logic as the implementation language for lambda calculus.

It should be stressed that the results outlined above do not give a full picture of the relation between lambda calculus and combinatory logic. A more comprehensive discussion of the issue can be found in [28].

Exercises

1. Show that the rewrite relation $\twoheadrightarrow_\alpha$ defined by the conversion rule is symmetric, but the reduction relation $\rightarrow_\alpha$ is not. (That is, it is not always possible to undo a single α-conversion step in one step).

2. A binary "diagonalization" operator W applies its first argument to the second one and the result again to the second argument;

   a. Represent W by a lambda term.

   b. Show the relation $\rightarrow_\beta$ for the term WWW

   c. Represent W by a combinator term.

3. Give an example of a lambda term which is not in a normal form and:

   a. has no normal form,

   b. has a normal form which cannot be obtained by the applicative-order reduction,

   c. has a normal form which can be obtained by the applicative-order reduction in less reduction steps than by the normal-order reduction,

   d. has a normal form which can be obtained by the normal-order reduction in less reduction steps than by the applicative-order reduction,

4. Find the normal form of the term $(SS(K(SKK)))_\lambda$.

5. Prove confluence of the combinatory logic.

6. A common construct in many programming languages is the **if** b **then**x **else** y expression, which has the value of x if b evaluates to $true$ and the value of y if b evaluates to $false$. Show how such a construction could be represented by a lambda term.

7. Show that there is no combinator A such that

   $AX = S$ if X is a variable or a basic combinator,
   $AX = K$ if $X = UV$ for some combinatory terms U, V.

Chapter 3

Term rewriting systems

We will now consider a class of rewriting systems — *term rewriting systems (*TRS*)* — which have been extensively studied during the last decades. TRSs constitute a completely general paradigm for describing computational processes; yet both syntax and semantics are very simple, which facilitates a nice mathematical analysis. The direct applications of TRSs range from implementation (especially of functional languages) to high-level specification of datatypes, etc.

A TRS is a reduction system over expressions called *terms*, defined formally in the next section. It is given as a set of *rewrite rules*. As an example, consider:

$$0 + x \to x$$
$$s(x) + y \to s(x + y)$$

This TRS can be thought of as defining addition on natural numbers, where the number n is represented by the numeral $s^n(0)$. The first rule says: "The result of adding zero to a number is the same number". It would be inconvenient to represent this statement with an infinite set of rules:

$$0 + 0 \to 0$$
$$0 + s(0) \to s(0)$$
$$0 + s(s(0)) \to s(s(0))$$
$$\vdots$$

Instead we use variables. For instance, the term $0 + s(s(0))$ *matches* $0 + x$ (the left-hand-side of the first rule), since the two terms become equal if x takes the value $s(s(0))$. Then the first rule is applicable, and we can replace the term $0 + s(s(0))$ with the right-hand-side of the rule, where every occurrence of x

is replaced by $s(s(0))$. Thus the result of applying the first rule to $0 + s(s(0))$ is $s(s(0))$.

Note that the rules are one-directional: we cannot apply the first rule to $s(s(0))$ to get $0 + s(s(0))$. Note also that even though we have an intended interpretation for our symbols, all manipulations are purely syntactical: $x + 0$, for instance, cannot be reduced to x with the TRS above.

Sect. 3.4 contains the formal definition of a TRS and its induced *rewrite relation*. First we need some auxiliary concepts, defined in Sect. 3.1 through 3.3.

3.1 Terms

A *term* is a sequence of symbols representing atomic objects and construction operators applied to these objects. Terms are perhaps most clearly displayed as trees, with the atomic objects at the leaves and the construction operators at internal nodes. However, a linear textual representation is the easiest to work with.

As an example, $\times + x\, 0\, y$ is a term. Its tree representation is shown in Fig. 3.1.

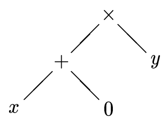


Figure 3.1: The term $\times + x\, 0\, y$

For clarity, we often use the infix notation $(x + 0) \times y$ instead. Note that in this case parentheses have to be added to disambiguate the expression, but these parentheses do not belong to the actual term. (In fact, we will use parantheses also for the prefix notation, i.e. we will write $\times(+(x, 0), y)$ instead of $\times + x\, 0\, y$.)

More formally, we assume the existence of a *signature* F, which is a finite set of construction operators (also called *function symbols*). Associated with every $f \in F$ is a natural number denoting its *arity* (number of arguments). A function symbol f with arity n is sometimes denoted f/n. Function symbols of arity 0 are called *constants*.

Definition 3.1 **[Term]** Let F be a signature, and let V be a denumerable set of variables. The set $Ter(F, V)$ of *terms* over F and V is defined:

- if $x \in V$ then $x \in Ter(F,V)$
- if $f/n \in F$ and $t_1 \ldots t_n \in Ter(F,V)$ then $f(t_1,\ldots,t_n) \in Ter(F,V)$
- nothing else is in $Ter(F,V)$

□

Notation: We write c instead of $c()$ whenever c is a constant. In the following, letters in the beginning of the alphabet $(a,b,c\ldots)$ will denote constants, and $f,g,h\ldots$ will denote function symbols with at least one argument. $x,y,z\ldots$ will denote variables. The set of variables occurring in a term t is denoted by $Var(t)$. We write $t_1 \equiv t_2$ if t_1 and t_2 are syntactically equal. □

Definition 3.2 [Ground term, Linear term] A term t is *ground* if $Var(t) = \emptyset$. t is *linear* if no variable occurs more than once in t. □

3.2 Subterms

s is a *subterm* of t if s is a term which occurs somewhere in t. For example, $+(x,0)$ is a subterm of $\times(+(x,0),y)$. However, the same subterm may occur at several places in a term (as in $f(c,c)$), and in the context of term rewriting it is necessary to be able to refer to a specific occurrence of a subterm. So the above notion of subterm in not precise enough for our requirements, but our next definition gives us a precise formalism to describe subterm occurrences.

Definition 3.3 [Position] The set $O(t)$ of *positions* in a term t is defined as follows:

$$O(t) = \begin{cases} \{\epsilon\} & \text{if } t \in V \\ \{\epsilon\} \cup \{i.p \mid 1 \le i \le n \text{ and } p \in O(t_i)\} & \text{if } t \equiv f(t_1,\ldots,t_n) \end{cases}$$

where ϵ denotes the empty sequence. □

Thus positions are sequences of natural numbers denoting subterm occurrences. We are now ready to define subterms.

Definition 3.4 [Subterm] If $p \in O(t)$ then $t_{|p}$ denotes the *subterm* of t at position p, defined as follows:

$$t_{|p} = \begin{cases} t & \text{if } p = \epsilon \\ (t_i)_{|q} & \text{if } t \equiv f(t_1,\ldots,t_n) \text{ and } p = i.q \end{cases}$$

□

For example, if t is the term $\times(+(x,0),y)$ then $t_{|1.2} \equiv 0$; that is, 0 is a subterm of $\times(+(x,0),y)$ at position 1.2. If the arity of each function symbol is less than 10 we may ommit the points in this notation (i.e. the example position above may be represented as 12).

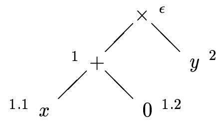


Figure 3.2: The positions of the term $\times + x\, 0\, y$

The following definitions are important in the context of rewriting.

Definition 3.5 If $p \in O(t)$ then $t[s]_p$ denotes the term obtained from t by replacing the subterm at position p with the term s, i.e. :

$$t[s]_p = \begin{cases} s & \text{if } p = \epsilon \\ f(t_1,\ldots,t_i[s]_q,\ldots t_n) & \text{if } t \equiv f(t_1,\ldots,t_n) \text{ and } p = i.q \end{cases}$$

□

For example, if t is the term $\times(+(x,0),y)$ then $t[\times(x,y)]_{1.2}$ is the term $\times(+(x,\times(x,y)),y)$.

Definition 3.6 [Non-variable position] By $\overline{O}(t)$ we denote the set of non-variable positions in t, i.e. the set $\{p \in O(t) \mid t_{|p} \notin V\}$. □

For example, if t is the term $\times(+(x,0),y)$ then $\overline{O}(t)$ is the set $\{\epsilon, 1, 1.2\}$.

Definition 3.7 [Prefix ordering] If p and q are two positions such that $q = p.r$ for some position r, we write $p \le q$ (or $q \ge p$). $\le$ is called the *prefix ordering* on positions. □

Note that if $p,q \in O(t)$ and $p \le q$, then $t_{|q}$ is a subterm of $t_{|p}$.

3.3 Substitutions

A *substitution* is an assignment of values to variables. The formal definition is as follows:

Definition 3.8 **[Substitution]** A *substitution* is a function $\sigma : \mathit{Ter}(F, V) \to \mathit{Ter}(F, V)$ such that $\sigma(f(t_1, \ldots, t_n)) \equiv f(\sigma(t_1), \ldots, \sigma(t_n))$, for every $f/n \in F$. $\square$

Thus a substitution σ is completely determined by its restriction to V, and can be seen as a function from V to $\mathit{Ter}(F, V)$.

Notation: Lowercase greek letters will denote substitutions. We will write a substitution σ as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $x_1 \ldots x_n$ are all variables x such that $x \not\equiv \sigma(x)$. If σ is the identity function, then σ is called the *empty* substitution. If $t_1, \ldots, t_n$ is a permutation of $x_1, \ldots, x_n$, then σ is a (variable) *renaming*. The composition of substitutions σ and ρ is denoted $\rho\sigma$, (i.e. the composition maps each variable x to the term $\rho(\sigma(x))$). If there exists a substitution ρ such that $\tau = \rho\sigma$, we say that σ is *more general* (less instantiating) than τ, and write this as $\sigma \leq \tau$. $\square$

For example, if $\sigma = \{x \mapsto y + z\}$ and t is the term $s(x) + s(s(x))$, then $\sigma(t) \equiv s(y+z)+s(s(y+z))$. If $\tau = \{x \mapsto y+1\}$ then $\sigma \leq \tau$ since $\tau = \{z \mapsto 1\}\sigma$.

Definition 3.9 **[Unifier]** Two terms t and s are *unifiable* if there exists a substitution σ such that $\sigma(t) \equiv \sigma(s)$. σ is called a *unifier* of t and s. σ is a *most general unifier* of t and s iff $\sigma \leq \tau$, where τ is any unifier of t and s. $\square$

For example, if t is the term $s(x) + s(s(x))$, and s is the term $s(s(y)) + z$, then $\{x \mapsto s(y), z \mapsto s(s(s(y)))\}$ is a most general unifier of t and s.

It is well-known that every pair of terms has a most general unifier (which is unique up to renaming) or no unifier at all. This result will be explained in chapter 6, where we also will give an algorithm to compute a most general unifier of two terms.

3.4 Term rewriting systems

We are now ready for the main definition of this chapter.

Definition 3.10 **[Term rewriting system]** A *term rewriting system (TRS)* is a pair (F, R), where F is a signature, and $R \subset \mathit{Ter}(F, V) \times \mathit{Ter}(F, V)$ is a set of *rewrite rules*. Every rewrite rule (l, r) satisfies the following two constraints:

$$\textbf{(i)}\ l \notin V \quad \text{and} \quad \textbf{(ii)}\ \mathit{Var}(r) \subseteq \mathit{Var}(l)$$

$\square$

A rewrite rule (l, r) will henceforth be written as $l \to r$. We will often present a TRS as a set of rewrite rules, wihout explicitly mentioning its signature.

Some authors omit conditions (i) and/or (ii) in the definition of rewrite rule. However, rewriting systems with rules disobeying (i) or (ii) are hard to reason about.

Definition 3.11 **[Rewriting relation]** Let $\mathcal{R}$ be a TRS. Its associated *rewriting relation* $\to_{\mathcal{R}}$ is defined as follows: $s \to_{\mathcal{R}} t$ iff there exist:

- a rewrite rule $l \to r$ in $\mathcal{R}$
- a position $p \in \overline{O}(s)$
- a substitution σ

such that $s_{|p} \equiv \sigma(l)$ and $t \equiv s[\sigma(r)]_p$. The term $s_{|p}$ is called a *redex* in s, and $s \to_{\mathcal{R}} t$ is called a *rewrite step.* $\square$

The reader is encouraged to verify that this definition indeed corresponds to the operational intuition of a term rewriting system, as described in the beginning of this chapter. That is, to rewrite a term s given a TRS $\mathcal{R}$, choose a subterm u in s and a rule in $\mathcal{R}$ whose left-hand-side matches u, and replace u with the right-hand-side of the rule (with the appropriate variable bindings).

Example 3.12 Consider the following TRS:

$$\begin{aligned}
&0 + x \to x \\
&s(x) + y \to s(x + y) \\
&0 \times x \to 0 \\
&s(x) \times y \to (x \times y) + y
\end{aligned}$$

We have the following sequence of rewriting steps (the redex is underlined in each step):

$$\begin{aligned}
&\underline{s(s(0)) \times s(s(0))} \to_{\mathcal{R}} \\
&(\underline{s(0) \times s(s(0))}) + s(s(0)) \to_{\mathcal{R}} \\
&((\underline{0 \times s(s(0))}) + s(s(0))) + s(s(0)) \to_{\mathcal{R}} \\
&(\underline{0 + s(s(0))}) + s(s(0)) \to_{\mathcal{R}} \\
&\underline{s(s(0)) + s(s(0))} \to_{\mathcal{R}} \\
&s(\underline{s(0) + s(s(0))}) \to_{\mathcal{R}} \\
&s(s(\underline{0 + s(s(0))})) \to_{\mathcal{R}} \\
&s(s(s(s(0))))
\end{aligned}$$

□

A more interesting example of a TRS is the combinatory logic discussed in Section 2.5. The alphabet consists of the variables, of the constants S and K, and of the binary functor *app*, which in the notation of Section 2.5 does not appear explicitly. This TRS is thus

$$app(app(K,x),y) \to x$$
$$app(app(app(S,x),y),z) \to app(app(x,z),app(y,z))$$

This shows the power of the formalism of the term rewriting systems as a computational mechanism.

In contrast to the combinatory logic, the lambda calculus is not a term rewriting system in the sense defined above since, due to the use of the bound variables, lambda terms are not terms in the sense of Definition 3.1.

The abstract notions of the previous chapter can be used for term rewriting systems if we associate with every TRS $\mathcal{R} = (F, R)$ the ARS $(Ter(F,V), \to_{\mathcal{R}})$. In the following, when we say that a given TRS $\mathcal{R}$ is terminating, confluent, etc. , we really mean that the associated ARS has these properties.

3.5 Critical pairs

In this section, we shall briefly review some results about confluent term rewriting systems. We introduce the notion of *critical pairs*, and state the fundamental *critical pair lemma* (theorem 3.17).

We begin with an example. Consider the following TRS $\mathcal{R}$, defining a group:

$$r_1 : 0 + x \to x$$
$$r_2 : -x + x \to 0$$
$$r_3 : (x+y)+z \to x+(y+z)$$

The term $(-a+a)+b$ can be rewritten in two ways:

$$\begin{tikzcd} & (-a+a)+b \arrow[dl] \arrow[dr] & \\ 0+b & & -a+(a+b) \end{tikzcd}$$

Figure 3.3: Rewriting of $(-a+a)+b$

$0+b$ can be further reduced to b, whereas $-a+(a+b)$ is already a normal form. This example shows that $\mathcal{R}$ is not weakly confluent, and hence not confluent.

In the general case, there may be infinitely many terms that can be rewritten, as in the example above, into two different terms t_1 and t_2. To prove weak confluence, we have to show that for each such pair of terms t_1 and t_2 there is a term u such that $t_1 \twoheadrightarrow u$ and $t_2 \twoheadrightarrow u$. However, Knuth and Bendix [40] showed that only finitely many of these pairs have to be considered.

Definition 3.13 **[Critical pair]** Let $l \to r$ and $k \to s$ be two renamed versions of rewrite rules of a TRS $\mathcal{R}$, such that they have no variables in common. Suppose a subterm (which is not a variable) of l is unifiable with k, i.e. there is a position $p \in \overline{O}(l)$ such that $\sigma(l_{|p}) \equiv \sigma(k)$ for a most general unifier σ. The term $\sigma(l) \equiv \sigma(l[k]_p)$ is subject to the reduction steps $\sigma(l) \to \sigma(r)$ and $\sigma(l) \to \sigma(l[s]_p)$. The pair of reducts $(\sigma(r), \sigma(l[s]_p))$ is a *critical pair* of $\mathcal{R}$, obtained by *superposition* of $k \to s$ on $l \to r$.

If $l \to r$ and $k \to s$ are renamed versions of the same rewrite rule, we require that $p \neq \epsilon$ (to avoid trivial pairs of the form (t,t)). □

Example 3.14 In the TRS $\mathcal{R}$ above, $0 + x_1$ (a renaming of the left-hand-side of the first rule) is unifiable with $(x_2 + y_2)$ (which is a subterm of the left-hand-side of the third rule, with variables renamed). A most general unifier is $\sigma = \{x_2 \mapsto 0, y_2 \mapsto x_1\}$. Hence, from the term $(0+x_1)+z_2$ we get the critical pair $(x_1 + z_2,\ 0 + (x_1 + z_2))$.

$\mathcal{R}$ has three critical pairs:

1. $(x_1 + z_2,\ 0 + (x_1 + z_2))$ by superposition of r_1 on r_3;
2. $(0 + z_2,\ -x_1 + (x_1 + z_2))$ by superposition of r_2 on r_3;
3. $((x_1 + (y_1 + z_1)) + z_2,\ (x_1 + y_1) + (z_1 + z_2))$ by superposition of r_3 on r_3.

□

Note that the pair $(0 + b,\ -a + (a + b))$ in Fig. 3.3 is an instance of the critical pair $(0 + z_2,\ -x_1 + (x_1 + z_2))$.

Definition 3.15 **[Convergence]** A critical pair (s,t) is *convergent* if there exists a term u such that $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$. □

Example 3.16 The critical pair $(0 + z_2,\ -x_1 + (x_1 + z_2))$ of the previous example is not convergent: $-x_1 + (x_1 + z_2))$ is already a normal form, while $0 + z_2$ reduces to the normal form z_2.

$(x_1+z_2,\ 0+(x_1+z_2))$ is obviously convergent since $0+(x_1+z_2)) \to x_1+z_2$. Also $((x_1 + (y_1 + z_1)) + z_2, (x_1 + y_1) + (z_1 + z_2))$ is convergent (verify!). □

The following theorem by Huet [32] shows the significance of critical pairs.

Theorem 3.17 [Critical Pair Lemma] A TRS is weakly confluent iff all of its critical pairs are convergent.

Proof : ($\Rightarrow$) For every critical pair (t_1, t_2) there is a term t such that $t \to t_1$ and $t \to t_2$. Since the TRS is WCR, there exists a term s such that $t_1 \twoheadrightarrow s$ and $t_2 \twoheadrightarrow s$, which means that (t_1, t_2) is convergent.

($\Leftarrow$) Assume that all critical pairs are convergent. Let t, t_1, t_2 be arbitrary terms such that $t \to t_1$ and $t \to t_2$, i.e. there exists $p_1, p_2 \in O(t)$, two rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$, and two substitutions σ_1, σ_2 such that $t_{|p_1} \equiv \sigma_1(l_1)$, $t_{|p_2} \equiv \sigma_2(l_2)$, $t_1 \equiv t[\sigma_1(r_1)]_{p_1}$ and $t_2 \equiv t[\sigma_2(r_2)]_{p_2}$. We would now like to prove that there exists a term s such that $t_1 \twoheadrightarrow s$ and $t_2 \twoheadrightarrow s$.

We can distinguish three cases (see figure 3.4).

(a) (Disjoint redexes) If neither $p_1 \leq p_2$ nor $p_2 \leq p_1$, the two redexes $t_{|p_1}$ and $t_{|p_2}$ are *disjoint*, and rewriting one redex will not "destroy" the other redex. For instance, $t_{1|p_2} \equiv \sigma_2(l_2)$, so if we let $s \equiv t_1[\sigma(r_2)]_{p_2}$ then obviously $t_1 \to s$. In the same way, we see that $t_2 \to s$.

(b) (Nested redexes) Assume that $p_2 = p_1.q_1$ (that is $p_1 \leq p_2$), and that $l_{1|q_1} \equiv x$ (where x is a variable). Thus, when rewriting t into t_1, x will be bound to some term containing the redex $t_{|p_2}$. This means that if we first would rewrite t into t_2, we do not "destroy" the bigger redex $t_{|p_1}$, which is still a redex in t_2. We let s be the term obtained by rewriting $t_{2|p_1}$ with the rule $l_1 \to r_1$.

On the other hand, t_1 contains as many occurrences of the smaller redex $t_{|p_2}$ as there are occurrences of the variable x in the right-hand-side of $l_1 \to r_1$. Obviously, we can obtain s by rewriting the k redexes in t_1 with the rule $l_2 \to r_2$.

(c) (Overlapping redexes) Assume that $p_2 = p_1.q$, and that $l_{1|q}$ is not a variable. In this case, rewriting one redex in t may destroy the other redex.

Let $u \equiv t_{|p_1}$. Then $u \equiv \sigma_1(l_1)$ and $u_{|q} \equiv \sigma_2(l_2) \equiv \sigma_2(l_{1|q})$. Thus l_2 and $l_{1|q}$ (with variables renamed apart) unify, giving a most general unifier ρ. Since $l_{1|q}$ is not a variable, $(\rho(r_1), \rho(u[r_2]_q))$ is a critical pair. By assumption all critical pairs are convergent, so there exists some v such that $\rho(r_1) \twoheadrightarrow v$ and $\rho(u[r_2]_q) \twoheadrightarrow v$. Obviously, this implies that there exists an s such that $t_1 \twoheadrightarrow s$ and $t_2 \twoheadrightarrow s$ (s is an instance of $t[v]_{p_1}$).

□

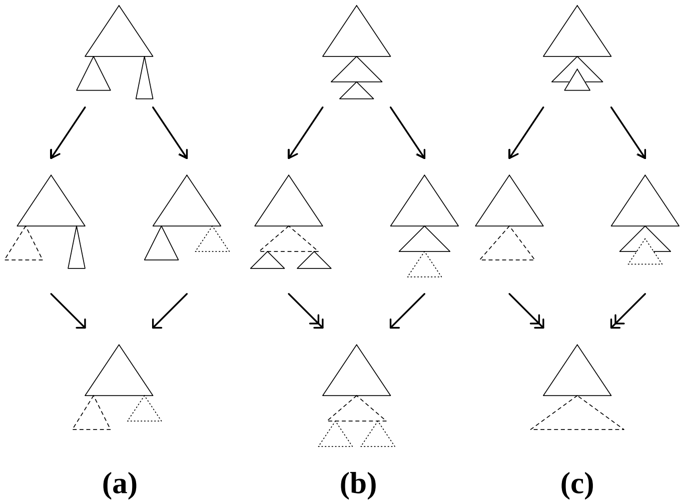


Figure 3.4: Possible cases in the Critical Pair Lemma

To illustrate the three cases in the proof of the Critical Pair Lemma, we give an example.

Example 3.18 Let $\mathcal{R}$ be the TRS:

$$f(x) \to g(x, x)$$
$$a \to b$$
$$f(b) \to a$$
$$g(b, b) \to a$$

The term $g(a, a)$ contains two disjoint redexes, $f(a)$ contains two nested redexes, and $f(b)$ contains two overlapping redexes (see figure 3.5).

□

Combining Newman's Lemma (theorem 1.7) and the Critical Pair Lemma yields Knuth's and Bendix' original (less general) result:

Theorem 3.19 A terminating TRS is confluent iff all of its critical pairs are convergent.

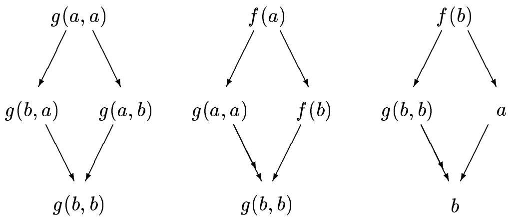


Figure 3.5: Rewriting of disjoint, nested and overlapping redexes

The termination requirement is obviously too strong for many applications. It is possible to find sufficient conditions for confluence without requiring termination. One such condition is *orthogonality*. Orthogonal term rewriting systems play a prominent role in the design and implementation of functional programming languages.

Definition 3.20 **[Orthogonal** TRS] A rewrite rule $l \to r$ is *left-linear* if l is a linear term. A TRS $\mathcal{R}$ is *orthogonal* if:

- every rewrite rule in $\mathcal{R}$ is left-linear
- $\mathcal{R}$ does not have any critical pairs

□

Theorem 3.21 (Rosen [63]) Any orthogonal TRS is confluent.

The following example shows that absence of critical pairs is not sufficient for confluency. The following TRS:

$$f(x, x) \to c$$
$$g(x) \to f(x, g(x))$$
$$a \to g(a)$$

has no critical pairs, but it is not orthogonal, as the first rule is not left-linear. It is also not confluent: $g(a) \to f(a, g(a)) \to f(g(a), g(a)) \to c$. On the other hand $g(a) \to g(g(a)) \twoheadrightarrow g(c)$. The redex $g(c)$ has no normal form, hence cannot be reduced to the normal form c. Consequently the TRS is not confluent.

Exercises

1. We introduce the shorthand $s^n(0)$ to denote $\overbrace{s(s(\ldots s}^{n \text{ times}}(0)))$. Construct a TRS that reduces the term $\mathit{half}(s^{2n}(0))$ to $s^n(0)$. (When you have succeeded, try to do it with only two rules).

2. Construct a TRS that reduces the term $\mathit{sum}(s^n(0))$ to $s^m(0)$ iff
$$m = \sum_{i=0}^{n} i$$

3. A *deterministic finite automaton (*DFA*) M* is a device that accepts a *regular language L*. M is determined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

   - Q is a finite set of *states*
   - Σ is an *alphabet*; $L \subseteq \Sigma^*$
   - δ is the *transition function*; $\delta : Q \times \Sigma \to Q$
   - $q_0 \in Q$ is the *starting state*
   - $F \subseteq Q$ is the set of *final states*

   Initially, M is in the start state q_0. M then reads the input string w one symbol at a time, and changes state according to the transition function δ. M halts when all symbols of w have been read. If M halts in a final state, w is accepted (belongs to L), otherwise w is rejected. At any given moment, M can be described by its *configuration* (a pair of current state/remaining input). The initial configuration is the pair (q_0, w). For more details, see Hopcroft and Ullman [29].

   (a) Find a construction that models an arbitrary DFA M as a TRS R. The configurations of M should be modeled as terms. The term representing the initial configuration (q_0, w) should be reduced to the constant *yes* iff the input string w is accepted by M, and to the constant *no* otherwise.

   (b) (A bit more difficult) Find a construction that models an arbitrary DFA M as a TRS R, such that R has a fixed number of rules (e.g. 3 rules), regardless of how big M is.

   (c) What have you showed with your constructions in (a) and (b)?

4. Prove confluence of the combinatory logic.

5. Let $\mathcal{R}$ be the TRS $(\{f/2, g/1, h/1, c/0\}, R)$, where R consists of the following rules:

$$
\begin{aligned}
&f(x, x) \to x \\
&f(g(x), y) \to x \\
&g(h(x)) \to c
\end{aligned}
$$

Find the critical pairs in $\mathcal{R}$. Which pairs are convergent?

6. Consider the TRS

$$f(a, f(x, y)) \to f(x, f(x, f(b, b)))$$

Decide whether this TRS is

(a) weakly normalizing
(b) terminating
(c) confluent

Chapter 4

Termination

In many term rewriting applications, termination is a crucial property, e.g. when solving so-called *word problems* in varieties of algebras (chapter 5), or in *unification* in equational theories (chapter 5). *Reduction orderings*, introduced in this chapter, are vital in the Knuth-Bendix completion procedure (section 5.4).

It is undecidable whether a TRS is terminating, even if it has only one rule. This can be shown by reducing the problem to the uniform halting problem for Turing machines (see Huet and Lankford [34], Dershowitz [20] and Dauchet [16] for different proofs). Of course, termination or non-termination can still be decided in some cases. For instance, it is clear that the one-rule TRS

$$f(x) \quad\to\quad x \tag{4.1}$$

is terminating, since the term we obtain after applying the rule is a subterm of the term we had before. An equally trivial example of a non-terminating system is

$$f(x) \quad\to\quad g(f(x)) \tag{4.2}$$

However, even for such simple systems as

$$-(x+y) \quad\to\quad (--x+y)+y \tag{4.3}$$

it is not immediately obvious whether the system is terminating or not (try a guess before looking at the solution in exercise 1). Things get even more complicated when the TRS contains more rules, as in

$$\begin{array}{rcl} \neg\neg x & \to & x \\ \neg(x \vee y) & \to & \neg x \wedge \neg y \\ \neg(x \wedge y) & \to & \neg x \vee \neg y \\ x \wedge (y \vee z) & \to & (x \wedge y) \vee (x \wedge z) \\ (x \vee y) \wedge z & \to & (x \wedge z) \vee (y \wedge z) \end{array} \tag{4.4}$$

Some rewrite sequences decrease the length of a term, e.g.

$$\neg(0 \wedge (\neg 1 \vee \neg 1)) \quad\twoheadrightarrow\quad \neg 0 \vee (1 \wedge 1)$$

while others increase it, e.g.

$$\neg(0 \wedge (1 \vee 1)) \quad\twoheadrightarrow\quad (((\neg 0 \wedge \neg 0) \vee (\neg 0 \wedge \neg 1)) \vee ((\neg 1 \wedge \neg 0) \vee (\neg 1 \wedge \neg 1)))$$

So, is system 4.4 terminating or non-terminating?

In the rest of the chapter we will give some methods of proving termination (and non-termination).

Notation: Throughout this chapter we let $>$ denote a (strict) partial ordering, i.e. a transitive and irreflexive relation, on some domain A. We write $a = b$ whenever a and b is the same element in A. $a \geq b$ means that either $a = b$ or $a > b$. □

Definition 4.1 [Well-founded sets and orderings] A partial ordering $>$ is *well-founded* on the set A if there are no infinitely decreasing chains $a_1 > a_2 > a_3 > \ldots$ The pair $(A, >)$ is a *well-founded set*. □

Definition 4.1 immediately suggests the following characterization of a terminating TRS:

Theorem 4.2 (Manna and Ness [47]) A TRS (F, R) is terminating if and only if there exists a well-founded ordering $>$ on $Ter(F, V)$ such that:

$$t \to_R s \;\;\Rightarrow\;\; t > s$$

The above characterization is awkward since an infinite number of possible rewrites have to be considered. To avoid this, we introduce the following notions:

Definition 4.3 [Reduction ordering] A partial ordering $>$ on $Ter(F, V)$ is *monotone* if:

$$t > s \;\;\Rightarrow\;\; u[t]_p > u[s]_p$$

for all $s, t, u \in Ter(F, V)$ and $p \in O(u)$. A partial ordering $>$ on $Ter(F, V)$ is *stable* if:

$$t > s \;\;\Rightarrow\;\; \sigma(t) > \sigma(s)$$

for all $s, t \in Ter(F, V)$ and all substitutions σ. A *reduction ordering* is a *monotone, stable and well-founded* ordering $>$ on $Ter(F, V)$. □

As a corollary of theorem 4.2 we now obtain:

Theorem 4.4 (Lankford [44]) A TRS (F, R) is terminating if and only if there exists a reduction ordering such that:

$$l > r$$

for all rules $l \to r$ in R.

4.1 Polynomial interpretations

It is often convenient to define a well-founded ordering on $Ter(F, V)$ by mapping terms to some known well-founded set $(A, >)$. The so-called *polynomial interpretations* constitute the standard example of this technique.

Definition 4.5 [Termination function] A *termination function* τ is a function from $Ter(F, V)$ to a well-founded set $(A, >)$ that associates every $f/n \in F$ to a function $f_\tau : A^n \to A$, such that:

$$\tau(f(t_1, \ldots, t_n)) = f_\tau(\tau(t_1), \ldots, \tau(t_n))$$

for every $f(t_1, \ldots, t_n) \in Ter(F, V)$, and

$$a_1 > a_2 \quad \Rightarrow \quad f_\tau(\ldots a_1 \ldots) > f_\tau(\ldots a_2 \ldots)$$

for every $a_1, a_2 \in A$ and every $f/n \in F$. □

Note that τ and $(A, >)$ induce a well-founded ordering $>$ on the *ground* terms in $Ter(F, V)$, namely $t > s$ iff $\tau(t) > \tau(s)$. Using this observation, we can now refine theorem 4.4 to obtain the following result:

Theorem 4.6 (Lankford [43]) A TRS (F, R) is terminating if and only if there exists a well-founded set $(A, >)$ and a termination function $\tau : Ter(F, V) \to A$, such that for each rule $l \to r$ in R:

$$\tau(\sigma(l)) > \tau(\sigma(r))$$

for every *grounding* substitution σ.

With a *grounding* substitution for a term t, we mean a substitution σ such that $\sigma(t)$ is a ground term.

When we let τ associate every $f/n \in F$ with a polynomial $p(x_1, \ldots, x_n)$ over the non-negative integers, we obtain a *polynomial interpretation*. Verifying the condition in theorem 4.6 (that $\tau(\sigma(l)) > \tau(\sigma(r))$) now amounts to checking whether the polynomial $\tau(l) - \tau(r)$ is positive when the variables are bound to non-negative integers.

Example 4.7 Consider the following TRS:

$$\begin{array}{rcl} x + 0 & \to & x \\ x + s(y) & \to & s(x + y) \\ x \times 0 & \to & 0 \\ x \times s(y) & \to & (x \times y) + x \end{array}$$

To prove termination, we define the following termination function τ:

$$\begin{array}{rcl} 0_\tau & = & 1 \\ s_\tau(n) & = & n + 1 \\ +_\tau(m, n) & = & m + 2n + 1 \\ \times_\tau(m, n) & = & 2mn + 2n + 1 \end{array}$$

The reader is encouraged to verify that the polynomials $\tau(l) - \tau(r)$ are indeed positive, for all rules $l \to r$ above. □

4.2 Recursive path orderings

It may be difficult, or even impossible, to find a polynomial interpretation that satisfies the requirements of theorem 4.6 for a given TRS. Using a polynomial interpretation means that the "size" of a term is bounded by the polynomial which is used: Since the "size" decreases in every rewrite step, the length of the rewrite sequence must be bounded by a polynomial. This is not the case for every TRS. Consider the following example:

Example 4.8 The so-called *Ackermann* function is a well-known example of a function which is not primitive recursive. It is defined as follows:

$$f_{ack}(m, n) = \begin{cases} n + 1 & \text{if } \ m = 0 \\ f_{ack}(m - 1, 1) & \text{if } \ m > 0 \ \text{ and } \ n = 0 \\ f_{ack}(m - 1, f_{ack}(m, n - 1)) & \text{if } \ m, n > 0 \end{cases}$$

The following TRS $\mathcal{R}$ computes the Ackermann function:

(1) $a(0, x) \to s(x)$

(2) $a(s(x), 0) \to a(x, s(0))$

(3) $a(s(x), s(y)) \to a(x, a(s(x), y))$

More precisely, it is not hard to show that:

$$f_{ack}(m, n) = k \quad \text{iff} \quad a(\underbrace{s(s(\ldots s(0)))}_{m}, \underbrace{s(s(\ldots s(0)))}_{n}) \twoheadrightarrow_{\mathcal{R}} \underbrace{s(s(\ldots s(0)))}_{k}$$

It is known that f_{ack} is totally defined, which implies that $\mathcal{R}$ is terminating. However, it is also known that f_{ack} grows faster than any polynomial in m and n. The normal form of the term $a(s^m(0), s^n(0))$ in $\mathcal{R}$ is of the form $s(s(\ldots s(0)))$. The rules of $\mathcal{R}$ produce it by increasing the total number of the occurrences of s in the terms rewritten at single steps of computation. Notice that this number can only be increased by (1) or by (3). (1) increases this number by 1. (3) increases it with $m-2$ when applied to a redex of the form $a(s^m(0), s^n(0))$. Any s added by (3) requires then more than one rewriting step in order to be placed in the normal form. Thus the total number of steps needed for normalisation for $m, n > 1$ is greater than or equal $f_{ack}(m,n) - (m+n)$. As f_{ack} grows faster than any polynomial, this number cannot be bounded by a polynomial.

□

We will now direct our attention to sufficient conditions for termination that focus more directly on the syntactic structure of terms. One such "syntactic" criterion was described in the beginning of this chapter: if in every rewriting step a term is rewritten into one of its subterms, then obviously the TRS must be terminating. However, this condition is crude and does not cover many TRS's. We will now describe a powerful well-founded ordering, the so-called *recursive path ordering*, that is be used in many cases to prove termination. We first give some preliminary definitions:

Definition 4.9 [Multiset] A *multiset* is an unordered list of elements, in which elements may have multiple occurrences. The set of all *finite* multisets over A is denoted by $Mul(A)$. □

For example, $[3,3,4,2]$ is a multiset, which is equal to $[4,3,2,3]$ but distinct from $[3,4,2]$. We extend the operations set difference $(-)$ and union $(\cup)$ to multisets in the following natural way: $X - Y$ is obtained by removing an occurrence of the element e in X for every occurrence of e in Y. $X \cup Y$ is obtained by adding an occurrence of e to X for every occurrence of e in Y. For instance, $[3,3,4,2] - [3,4,2,5] = [3]$, and $[3,4] \cup [3,4,5] = [3,3,4,4,5]$. We also extend the subset notion: X is a subset of Y if for every occurrence of e in X there is an occurrence of e in Y.

Definition 4.10 [Multiset ordering] Given a partial ordering $>$ on A, its induced *multiset ordering* $\gg$ is defined on $Mul(A)$ as follows: Let X and Y be multisets. Then $X \gg Y$ iff there exist multisets X' and Y' such that:

- X' is a non-empty subset of X
- $Y = (X - X') \cup Y'$

- $\forall y \in Y' \; \exists x \in X' : x > y$

□

We can now define the recursive path ordering:

Definition 4.11 [Recursive path ordering] Let $>$ be a partial ordering on a signature F. The associated *recursive path ordering* $>_{rpo}$ over $Ter(F,V)$ is defined recursively as follows:

$$s \equiv f(s_1, \ldots, s_m) >_{rpo} g(t_1, \ldots, t_n) \equiv t$$

if

$$s_i \geq_{rpo} t \quad \text{for some } i = 1, \ldots, m,$$

or

$$f > g \quad \text{and} \quad s >_{rpo} t_j \quad \text{for every } j = 1, \ldots, n,$$

or

$$f = g \quad \text{and} \quad [s_1, \ldots, s_m] \gg_{rpo} [t_1, \ldots, t_n],$$

where $\gg_{rpo}$ is the multiset ordering induced by $>_{rpo}$. □

Theorem 4.12 (Dershowitz [19]) $>_{rpo}$ is a well-founded, monotone and stable ordering on $Ter(F,V)$.

This tells us that theorem 4.4 is applicable. To prove termination with the recursive path ordering, we have the following methodology:

1. Choose a partial ordering $>$ on the signature F.

2. Show that, for every rule $l \to r$ and every substitution σ, we have $\sigma(l) >_{rpo} \sigma(r)$.

It should be stressed that if the TRS is not terminating, step 2 is impossible regardless of how we choose the ordering $>$ in step 1 (this is also a consequence of theorem 4.4).

4.3 Lexicographical orderings

Finally, in tricky cases several orderings can be combined to prove termination.

Definition 4.13 [Lexicographical combination] Given two partial orderings $>_1$ and $>_2$ defined on the same set, their *lexicographical combination* $lex(>_1, >_2) \ = >$ is defined as follows:

$$a > b$$

if

$$a >_1 b$$

or

$$a =_1 b \quad \text{and} \quad a >_2 b$$

where $a =_1 b$ whenever $a \geq_1 b$ and $b \geq_1 a$. □

Note that $lex(>_1, >_2)$ is not the same ordering as $lex(>_2, >_1)$! The definition generalizes naturally to lexicographical combinations of n orderings.

Theorem 4.14 If $>_1$ and $>_2$ are well-founded orderings, then $lex(>_1, >_2)$ is a well-founded ordering.

The above theorem suggests the following methodology to prove termination for a TRS $(F, \{r_1, \ldots, r_n\})$:

1. Find a well-founded ordering $>_1$ on $Ter(F, V)$ such that for some rules in $r_1, \ldots, r_i$, the left-hand-side (*lhs*) is greater than the right-hand-side (*rhs*) according to the order $>_1$. For the other rules $r_{i+1}, \ldots, r_n$ we must have that *lhs* $=_1$ *rhs*.

2. Find a new well-founded ordering $>_2$ and repeat the process. We stop when we have found k orderings that cover all the rules, i.e. for every rule among $r_1, \ldots, r_n$ we have that *lhs* $>_i$ *rhs* for some $i = 1, \ldots, k$, and that *lhs* $=_j$ *rhs* for every $j = 1, \ldots, i-1$.

3. The termination ordering is the lexicographical combination $lex(>_1, \ldots, >_k)$.

4.4 Non-termination

We conclude this chapter by a brief discussion on non-termination.

Definition 4.15 [Cycling] A rewriting sequence $t_1 \to \cdots \to t_i \to \cdots \to t_j \to \cdots$ *cycles* if $t_i \equiv t_j$ for some $i < j$. A TRS cycles if it has a cycling rewriting sequence. □

Cycling is a special case of *looping*:

Definition 4.16 [Looping] A rewriting sequence $t_1 \to \cdots \to t_i \to \cdots \to t_j \to \cdots$ *loops* if t_i is a subterm of t_j for some $i < j$. A TRS loops if it has a looping rewriting sequence. □

Clearly, if a TRS is cycling or looping, it is non-terminating. But a TRS need not to be looping to be non-terminating (see for instance exercise 1). We therefore define a weaker notion. Intuitively, a term s "embeds" t if t can be obtained from s by deleting certain function symbols and operands. Formally:

Definition 4.17 [Homeomorphic embedding] The *homeomorphic embedding relation* $\trianglerighteq$ on $Ter(F, V)$ is defined recursively as follows:

$$s \equiv f(s_1, \ldots, s_m) \trianglerighteq g(t_1, \ldots, t_n) \equiv t$$

if

$$s_i \trianglerighteq t \quad \text{for some } i = 1, \ldots, m$$

or

$$f = g \quad \text{and} \quad s_{i_j} \trianglerighteq t_j \quad \text{for all } j = 1, \ldots, n$$

where $1 \leq i_1, \ldots, i_n \leq m$. □

If t is embedded in s, but $t \not\equiv s$, we write $s \triangleright t$. For instance, if s and f are two unary function symbols and 0 is a constant, then $f(f(s(s(f(s(s(0))))))) \triangleright s(s(f(0)))$.

Theorem 4.18 [Tree theorem] (Kruskal [41]) If F is a signature, then any infinite sequence $t_1, t_2, \ldots$ of terms in $Ter(F, V)$ contains two terms t_i and t_j $(i < j)$ such that $t_j \trianglerighteq t_i$.

A consequence of the Tree theorem is that embedding is a necessary condition for non-termination:

Definition 4.19 [Self-embedding] A rewriting sequence $t_1 \to \cdots \to t_i \to \cdots \to t_j \to \cdots$ is *self-embedding* if $t_j \trianglerighteq t_i$ for some $i < j$. A TRS is self-embedding if it has a self-embedding rewriting sequence. □

Theorem 4.20 (Dershowitz [19]) If a finite TRS is non-terminating, then it is self-embedding.

This means that if a TRS is *not* self-embedding, it is terminating. Unfortunately, it is undecidable whether a TRS is self-embedding, as shown by Plaisted [59].

Exercises

1. Prove that the following TRS is non-terminating:

$$-(x + y) \to (- - x + y) + y$$

2. Prove, by using the polynomial interpretation method, that the following TRS is terminating:

$$\begin{array}{l} x \times (y + z) \to (x \times y) + (x \times z) \\ (x + y) \times z \to (x \times z) + (y \times z) \\ (x + y) + z \to x + (y + z) \end{array}$$

3. Prove, by using a recursive path ordering, that the following TRS is terminating:

$$\begin{array}{l} x + 0 \to x \\ x + s(y) \to s(x + y) \\ x \times 0 \to 0 \\ x \times s(y) \to (x \times y) + x \end{array}$$

4. Prove, by using any method, that the following TRS is terminating:

$$\begin{array}{l} 0 + x \to x \\ x + 0 \to x \\ -x + x \to 0 \\ x + -x \to 0 \\ (x + y) + z \to x + (y + z) \\ -0 \to 0 \\ -(x + y) \to -y + -x \\ x + (-x + y) \to y \\ -x + (x + y) \to y \\ - - x \to x \end{array}$$

Chapter 5

Equational logic and Knuth-Bendix completion

Equations and equational reasoning are used in a countless number of applications in mathematics, logic and computer science. Intuitively, an equation $M = N$ asserts that M and N denote the same object, given our intended interpretation of the symbols in M and N. For instance, the equation $2+2 = 3+1$ states that $2+2$ and $3+1$ denote the same number in our intented interpretation of integers. Equational reasoning is the well-known principle of "replacing equals by equals", e.g. $x - (2 + 2)$ may be replaced by $x - (3 + 1)$, using the previous equality.

In this context, with an equation we will understand a pair of terms (s, t), written as $s = t$. All variables occurring in an equation are (implicitly) universally quantified, so an equation $0 + x = x$ should be read "for all x, $0 + x$ is equal to x". An *equational specification* is a set of distinguished equations, called *axioms*, which are asserted to be true.

Before getting into technical details, we begin with an example. Consider the following equational specification G of a group:

$$0 + x = x$$

$$-x + x = 0$$

$$(x + y) + z = x + (y + z)$$

Suppose we are asked the question: "Does the equation $x + 0 = x$ follow from G?". How can we answer this question? Some knowledge in group theory would make us inclined to answer "yes", perhaps motivating the answer with the following sequence of equalities:

$$\begin{array}{l}
x + 0 \;=\\
x + (-x + x) \;=\; (x + -x) + x \;=\; ((0 + x) + -x) + x \;=
\end{array}$$

$$\begin{array}{l}
(((--x+-x)+x)+-x)+x \;=\; ((--x+(-x+x))+-x)+x \;=\\
((--x+0)+-x)+x \;=\; (--x+(0+-x))+x \;=\; (--x+-x)+x \;=\\
0 + x \;=\\
x
\end{array}$$

This is a *proof-theoretic* argument; we have manipulated $x + 0$ into x by equational reasoning (in section 5.1 we will formally define the inference rules). We could also give a *model-theoretic* argument; any equational specification implicitly defines an *equational theory*, i.e. the set of equations which are true in all models of the axioms (in section 5.2 we will give the formal definition of a *model* of a set of equations). By showing somehow that $x + 0 = x$ belongs to the equational theory defined by G, we have showed that $x + 0 = x$ indeed follows from G.

This dual proof-theoretic/model-theoretic view immediately raises the question of whether every equation that can be derived from an equational specification E by equational reasoning also belongs to the equational theory defined by E (soundness), and vice versa (completeness). A famous theorem by Birkhoff (theorem 5.6) states that this is indeed the case.

The problem of deciding whether or not a given equation follows from an equational specification is known as the *validity* problem. In this chapter, we will be concerned with the following two problems. We assume given an equational specification (F, E), where F is a signature and E is a set of axioms:

The word problem Do the two ground terms s and t denote the same object in all models of E?

The validity problem Does an equation e follow from E, i.e. does e belong to the equational theory presented by E? The word problem can be expressed as an instance of the validity problem as follows: "Does the ground equation $s = t$ belong to the equational theory presented by E?"

In many situations we are interested in finding *solutions* to equations. In this case, we do not consider equations as universally quantified statements, but rather as existentially quantified statements. For instance, solving the equation $x + 3 = 4$ (in integer arithmetic) amounts to showing that the statement

$$\exists x(x + 3 = 4)$$

is true, and to find the approriate value(s) for x, in this case $\{x \mapsto 1\}$.

In our context of term equations, the problem solving equations can be phrased as follows. Again we assume that (F, E) is an equational specification.

The unification problem Is the non-ground equation $s = t$ solvable, i.e. is there a substitution σ such that the equation $\sigma(s) = \sigma(t)$ belongs to the equational theory presented by E? The validity problem can be expressed as an instance of the unification problem as follows: "Is the empty substitution a solution to the equation $s = t$?"

Methods for solving the unification problem, as well as methods for computing all solutions (unifiers) to a given equation, will be presented in chapter 6.

We will now formally develop the theory of equational logic, and present methods to solve the problems mentioned above.

Definition 5.1 **[Equational specification]** An *equation* is a pair of terms (s, t), written as $s = t$. An *equational specification* is a pair (F, E), where F is a signature and E is a set of equations $s = t$, where $s, t \in Ter(F, V)$. □

5.1 Equational reasoning

Equational reasoning may be formalized as a set of inference rules.

Definition 5.2 $[\vdash, =_E]$ We say that an equation $s = t$ can be *derived* from a set E of equations, and write this as $E \vdash s = t$ or $s =_E t$, if $s = t$ can be obtained from E by means of the following inference rules:

1. **(Axiom)**
$$\frac{s = t \in E}{E \vdash s = t}$$
2. **(Substitution)**
$$\frac{E \vdash s = t}{E \vdash \sigma(s) = \sigma(t)} \text{ , for all substitutions } \sigma.$$
3. **(Context)**
$$\frac{E \vdash s_1 = t_1 \cdots E \vdash s_n = t_n}{E \vdash f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)} \text{ , for all } f/n \in F$$
4. **(Reflexivity)**
$$E \vdash t = t \text{ , for all } t \in Ter(F, V)$$
5. **(Symmetry)**
$$\frac{E \vdash s = t}{E \vdash t = s}$$
6. **(Transitivity)**
$$\frac{E \vdash s = t \quad E \vdash t = u}{E \vdash s = u}$$

□

5.2 Model theory

The inference system in the previous section provides the proof theory for equational logic. We will now give a semantics to our equations. Note that the four last inference rules imply that = must be interpreted as a congruence relation in every model of E. It is therefore not surprising that the model theory is based on the notion of *algebras*.

Definition 5.3 **[F-algebra, Interpretation, Valuation]** An *algebra* $\mathcal{A}$ consists of a set A, together with functions f from A^n to A (where n is the arity of f). $\mathcal{A}$ is an F*-algebra* if there is a signature F such that:

- there is an element $c_{\mathcal{A}} \in A$ for every constant c in F,
- there is an n-ary function $f_{\mathcal{A}} : A^n \to A$ for every function symbol $f/n \in F$.

Such a mapping from F to $\mathcal{A}$ is called an *interpretation*. A *valuation* is a mapping from a set of variables to A. □

Let (F, E) be an equational specification, and let V be the set of all variables occurring in E. We can give a meaning to the equations in E by deciding:

- an algebra $\mathcal{A}$,
- an interpretation I,
- a valuation $v : V \to A$

Now given a term $t \in Ter(F, V)$, we define its meaning $t^{(\mathcal{A},I,v)}$ as follows:

- If t is a variable x, then $t^{(\mathcal{A},I,v)} = v(x)$.
- If t is a constant c, then $t^{(\mathcal{A},I,v)} = I(c)$.
- If $t \equiv f(t_1, \ldots, t_n)$, then $t^{(\mathcal{A},I,v)} = I(f)(t_1^{(\mathcal{A},I,v)}, \ldots, t_n^{(\mathcal{A},I,v)})$.

Definition 5.4 **[Model, Validity, $\models$]** Let $s, t \in Ter(F, V)$, and let $\mathcal{A}$ be a F-algebra with an associated interpretation I. We say that $\mathcal{A}$ is *model* of $s = t$, or that $s = t$ is *valid* in $\mathcal{A}$ if:

$$\forall v : s^{(\mathcal{A},I,v)} = t^{(\mathcal{A},I,v)}$$

We write this as $\mathcal{A} \models s = t$. The notions of model and validity are naturally extended to sets E of equations, in which case we write $\mathcal{A} \models E$. □

Definition 5.5 [Logical consequence, Equational theory] Let (F, E) be an equational specification. An equation $s = t$ is a *logical consequence* of E if $s = t$ is valid in every model of E. This is written as $E \models s = t$. The *equational theory* presented by (F, E) is the set $\{s = t \mid E \models s = t\}$. □

Theorem 5.6 [Completeness of equational logic] (Birkhoff [3]) Let (F, E) be an equational specification and $s = t$ an equation. Then

$$E \vdash s = t \quad \text{iff} \quad E \models s = t$$

5.3 The validity problem

Recall that the *validity problem* for a given equational specification (F, E) is stated as follows:

> Given $s, t \in Ter(F, V)$, decide whether or not $E \models s = t$.

According to Birkhoff's completeness theorem, this amounts to deciding $E \vdash s = t$. Now it becomes clear why *complete* TRSs are important: Suppose we can find a TRS (F, R) such that for all terms $s, t \in Ter(F, V)$:

$$s =_R t \quad \text{iff} \quad E \vdash s = t \tag{5.1}$$

As a consequence of theorem 1.5, we have the following result for complete TRSs:

$$s =_R t \quad \text{iff} \quad \exists u : s \twoheadrightarrow u \quad \text{and} \quad t \twoheadrightarrow u \tag{5.2}$$

Combining (5.1) and (5.2) and Birkhoff's completeness theorem gives us the following simple decision algoritm for the validity problem:

1. Reduce s and t to their respective normal forms s' and t'
2. Compare s' and t'. $E \models s = t$ iff $s' \equiv t'$

To summarize: If it is possible to find a *complete* TRS satisfying (5.1), then the equational specification (F, E) has a decidable validity problem. The converse is not true: the specification:

$$0 + x = x$$
$$s(x) + y = s(x + y)$$
$$x + y = y + x$$

has a decidable validity problem, but it is impossible to find a complete TRS that satisfies (5.1).

If we restrict our attention to equations between *ground* terms, we are considering the *word problem* for (F, E):

> Given an equation $s = t$ between ground terms s and t, decide whether or not $E \models s = t$.

Also for the word problem, complete TRSs provide a solution. Actually, we only have to require confluence and termination for ground terms, not for all terms (see exercise 4 for an example where this makes a difference).

The important question is now: Given an equational specification (F, E), how do we find a complete TRS (F, R) satisfying (5.1)? In the next section, we present the so-called *Knuth-Bendix completion* procedure, that (semi-)automatically "compiles" an equational specification into a complete TRS with the desired properties (if possible).

5.4 Knuth-Bendix completion

As mentioned in the previous section, a term rewriting system that is complete (confluent and terminating) is a tool for solving the validity problem for a given specification E. Let us assume that the equations in E can be oriented as rules so that the resulting rewrite system $\mathcal{R}$ is terminating. For instance, in the group theory example we might obtain:

$$r_1 : \ 0 + x \rightarrow x$$
$$r_2 : \ -x + x \rightarrow 0$$
$$r_3 : \ (x + y) + z \rightarrow x + (y + z)$$

In general, $\mathcal{R}$ will not be confluent (for instance, we know from section 3.5 that the above system is not confluent). Intuitively, the reason for this is that there are "too few" rules. From a semantical point of view, the rules of $\mathcal{R}$ are sufficient since $s =_\mathcal{R} t$ iff $E \vdash s = t$. This is easily realized: $s =_\mathcal{R} t$ iff s can be rewritten into t by using the rules backwards and forwards an arbitrary number of times, that is by using equational reasoning. Operationally however, $s =_\mathcal{R} t$ cannot be established by reducing s and t to their respective normal forms s' and t', and by checking whether $s' \equiv t'$. But if $s =_\mathcal{R} t$, then surely $s' =_\mathcal{R} t'$, so adding the rule $s' \rightarrow t'$ to $\mathcal{R}$ does not change the underlying equational theory. This observation is the cornerstone of the Knuth-Bendix procedure.

More formally, let $\mathcal{R}' = \mathcal{R} \cup \{s' \rightarrow t'\}$. Then (provided that $s =_\mathcal{R} t$) for any terms $u, v \in Ter(F, V)$:

$$u =_{\mathcal{R}} v \Leftrightarrow u =_{\mathcal{R}'} v$$

which in particular means that for any terms $u, v \in Ter(F, V)$:

$$u =_{\mathcal{R}'} v \Leftrightarrow E \vdash u = v$$

(note that semantically the added rule is redundant). This inspires to the thought that $\mathcal{R}$ could be *completed* by adding sufficiently many rules. A rough sketch of a completion procedure is given in figure 5.1.

begin
    for $s, t \in Ter(F, V)$ **do begin**
        if $s =_{\mathcal{R}} t$ **then**
            reduce s and t to respective normal forms s' and t' w.r.t. $\mathcal{R}$;
        if (not $s' \equiv t'$) **then**
            $\mathcal{R} := \mathcal{R} \cup s' \to t'$
    end
end

Figure 5.1: Naïve completion algorithm

Obviously this naïve algorithm is not effectively computable. Since there are in general infinitely many terms, it requires infinitely many checks of the form $s =_{\mathcal{R}} t$ (which is undecidable), and the resulting system $\mathcal{R}$ will typically be infinite. But as a first approximation, this procedure shows the basic idea of completion.

At this point we remember Newman's Lemma (theorem 1.7) and the Critical Pair Lemma (theorem 3.17). The key to a realistic completion algorithm is the observation that for a finite rewrite system $\mathcal{R}$, the set of critical pairs is finite. So instead of investigating all terms $s, t \in Ter(F, V)$, we consider only the critical pairs (t, u) and add only rules $t' \to u'$ for non-convergent critical pairs (where t' and u' are the normal forms of t and u respectively). However, by adding a rule, new critical pairs may be generated which will have to be considered in due course. Thus, there is no guarantee that a completion procedure will eventually terminate, even though at every instant there are only finitely many critical pairs to investigate.

If the rewrite system at some point becomes confluent, then all critical pairs will be convergent, so no new rules are added and the procedure terminates successfully. But it may happen that by adding new rules, non-convergent critical pairs are generated ad infinitum, in which case the procedure loops.

When we add a rule to the TRS, we obviously have to decide which component of the critical pair should be the left-hand-side of the rule, and which component should be the right-hand-side. To ensure termination of the final system, we require that every rule is oriented according to the same reduction ordering (see definition 4.3). This may be impossible, in which case the procedure halts with failure.

The basic version of the Knuth-Bendix completion procedure is shown in figure 5.2. If it terminates successfully, the resulting TRS is complete (to show this requires a rather complex proof; see Huet [33]). To illustrate the use of the procedure, we continue the group theory example.

By superposition of r_2 on r_3, we obtain the non-convergent critical pair $(0 + z, -x + (x + z))$. The term $0 + z$ can be further reduced to z. We add the rule:

$$r_4 : \ -x + (x + z) \to z$$

From r_2 and r_4 we now get the critical pair $(x, - - x + 0)$, which cannot be further reduced. We add the rule

$$r_5 : \ - - x + 0 \to x$$

Superposition of r_4, r_1:

$$\begin{tikzcd} & z \\ -0 + (0 + z) \arrow[ur, "r_4"] \arrow[dr, "r_1"'] & \\ & -0 + z \end{tikzcd}$$

Add the rule

$$r_6 : \ -0 + z \to z$$

Superposition of r_3, r_5:

$$\begin{tikzcd} & - - x + (0 + z) \arrow[r, "r_1"'] & - - x + z \\ (- - x + 0) + z \arrow[ur, "r_3"] \arrow[dr, "r_5"'] & & \\ & x + z & \end{tikzcd}$$

Add the rule

$$r_7 : \ - - x + z \to x + z$$

Superposition of r_5, r_7:

$$\begin{tikzcd} & x + 0 \\ - - x + 0 \arrow[ur, "r_7"] \arrow[dr, "r_5"'] & \\ & x \end{tikzcd}$$

Add the rule

$$r_8 : \ x + 0 \to x$$

Input:
  an equational specification (F, E)
  a reduction ordering $>$
Output:
  a complete TRS $\mathcal{R}$ such that for all $s, t \in Ter(F, V)$:
    $s =_{\mathcal{R}} t \Leftrightarrow E \vdash s = t$

begin
  $R := \emptyset$;
  while $E \neq \emptyset$ **do begin**
    choose an equation $s = t \in E$;
    $E := E - \{s = t\}$;
    reduce s and t to respective normal forms s' and t' w.r.t. R;
    if (not $s' \equiv t'$) **then begin**
      if $s' > t'$ **then begin**
        $l := s'$; $r := t'$ **end**
      else if $t' > s'$ **then begin**
        $l := t'$; $r := s'$ **end**
      else
        $failure$;
      $CP := \{p = q \mid (p, q)$ is a critical pair between the rules in R and $l \to r\}$;
      $R := R \cup \{l \to r\}$;
      $E := E \cup CP$
    end
  end
  $success$
end

Figure 5.2: Basic version of the Knuth-Bendix completion procedure

Superposition of r_5, r_8:

$$\begin{tikzcd}[row sep=tiny] & x \\ --x+0 \arrow[ur, "r_5"] \arrow[dr, "r_8"'] & \\ & --x \end{tikzcd}$$

Add the rule

$$r_9 : \; --x \to x$$

The TRS is not completed yet, but let us pause and make an observation. The left-hand-side of r_5 can now be reduced to the right-hand-side by

$$--x+0 \;\to_9\; x+0 \;\to_8\; x$$

why the specialized rule r_5 is not needed and can be removed. Similarly, since

$$--x+z \;\to_9\; x+z$$

the more specialized rule r_7 can be removed. This is a typical phenomenon in the completion process: a rule that was added earlier may at a later stage be subsumed by a (combination of) new rule(s), and can then be removed. This observation motivates the optimized version of the Knuth-Bendix completion algorithm shown in figure 5.3.

In practice, performing this *interreduction* is absolutely crucial for the efficiency of the procedure. Not only does the basic algorithm generate an unnecessarily large TRS, but the amount of critical pairs that have to be considered grows extremely fast.

Note that the optimized algorithm not only removes unnecessary rules, but it also "simplifies" the rules.

Definition 5.7 A TRS $\mathcal{R}$ is called *irreducible* if for every rewrite rule $l \to r$ of $\mathcal{R}$:

(i) r is a normal form w.r.t. $\mathcal{R}$;

(ii) l is a normal form w.r.t. $\mathcal{R} - \{l \to r\}$.

□

Theorem 5.8 (Métivier [52]) Let $\mathcal{R}$ be a complete TRS. Then we can find an irreducible complete TRS $\mathcal{R}'$ such that for all $s, t \in Ter(F, V)$

$$s =_{\mathcal{R}'} t \;\text{ iff }\; s =_{\mathcal{R}} t$$

Input:
  an equational specification (F, E)
  a reduction ordering $>$
Output:
  a complete irreducible TRS $\mathcal{R}$ such that for all $s, t \in Ter(F, V)$:
    $s =_{\mathcal{R}} t \Leftrightarrow E \vdash s = t$

begin
  $R := \emptyset$;
  while $E \neq \emptyset$ **do begin**
    choose an equation $s = t \in E$;
    $E := E - \{s = t\}$;
    reduce s and t to respective normal forms s' and t' w.r.t. R;
    if (not $s' \equiv t'$) **then begin**
      if $s' > t'$ **then begin**
        $l := s'$; $r := t'$ **end**
      else if $t' > s'$ **then begin**
        $l := t'$; $r := s'$ **end**
      else
        $failure$;
      $R := \{k \to u' \mid k \to u \in R$ and u' is a normal form of u w.r.t. $R \cup \{l \to r\}\}$;
      $CP := \{p = q \mid (p, q)$ is a critical pair between the rules in R and $l \to r\}$;
      $E := E \cup CP \cup \{k = u \mid k \to u \in R$ and k is reducible by $l \to r\}$;
      $R := R \cup \{l \to r\} - \{k \to u \mid k \to u \in R$ and k is reducible by $l \to r\}$;
    end
  end
  $success$
end

Figure 5.3: More efficient version of the Knuth-Bendix completion procedure

The optimized algorithm generates such an irreducible complete TRS.

Let us continue the example:

Superposition of r_6, r_8:

$$\begin{tikzcd} & 0 \\ -0+0 \arrow[ur, "r_6"] \arrow[dr, "r_8"'] & \\ & -0 \end{tikzcd}$$

Add the rule

$$r_{10}: \; -0 \to 0$$

and remove r_6 which is now subsumed by r_{10} and r_1.

Superposition of r_2, r_9:

$$\begin{tikzcd} & 0 \\ --x+-x \arrow[ur, "r_2"] \arrow[dr, "r_9"'] & \\ & x+-x \end{tikzcd}$$

Add the rule

$$r_{11}: \; x+-x \to 0$$

Superposition of r_3, r_{11}:

$$\begin{tikzcd} & x+(-x+y) & \\ (x+-x)+y \arrow[ur, "r_3"] \arrow[dr, "r_{11}"'] & & \\ & 0+y \arrow[r, "r_1"'] & y \end{tikzcd}$$

Add the rule

$$r_{12}: \; x+(-x+y) \to y$$

Superposition of r_3, r_{11} (again):

$$\begin{tikzcd} & x+(y+-(x+y)) \\ (x+y)+-(x+y) \arrow[ur, "r_3"] \arrow[dr, "r_{11}"'] & \\ & 0 \end{tikzcd}$$

Add the rule

$$r_{13}: \; x+(y+-(x+y)) \to 0$$

Superposition of r_4, r_{13}:

$$\begin{tikzcd} & y+-(x+y) & \\ -x+(x+(y+-(x+y))) \arrow[ur, "r_4"] \arrow[dr, "r_{13}"'] & & \\ & -x+0 \arrow[r, "r_8"'] & -x \end{tikzcd}$$

Add the rule

$$r_{14}: \; y+-(x+y) \to -x$$

and remove r_{13} which is now subsumed by r_{14} and r_{11}.
Superposition of r_4, r_{14}:

$$\begin{tikzcd} & -(x+y) \\ -y+(y+-(x+y)) \arrow[ur, "r_4"] \arrow[dr, "r_{14}"'] & \\ & -y+-x \end{tikzcd}$$

Add the rule

$r_{15}:\ -(x+y) \to -y+-x$

and remove r_{14} which is now subsumed by r_{15} and r_{12}.

At this point, the TRS consisting of the remaining rules (see figure 5.4) has only convergent critical pairs (it is also irreducible). We can now use this TRS to decide validity problems in group theory. For instance, our original question of whether $x+0=x$ holds in all groups can now be decided by a single application of r_8.

$r_1:\ 0+x \to x$

$r_2:\ -x+x \to 0$

$r_3:\ (x+y)+z \to x+(y+z)$

$r_4:\ -x+(x+z) \to z$

$r_8:\ x+0 \to x$

$r_9:\ --x \to x$

$r_{10}:\ -0 \to 0$

$r_{11}:\ x+-x \to 0$

$r_{12}:\ x+(-x+y) \to y$

$r_{15}:\ -(x+y) \to -y+-x$

Figure 5.4: A complete TRS for group theory

Actually, the example concealed some important practical aspects.

Firstly, we only investigated the important critical pairs. The strategy for selecting critical pairs has a big impact on the overall performance, and with a bad heuristics one may have to examine many critical pairs which turn out to be convergent, or (worse still) add a lot of rules which will later be retracted. A *fair* selection strategy (i.e. a strategy that guarantees that every critical pair will eventually be examined) is needed for the procedure to terminate as often as possible.

Secondly, we did not provide any reduction ordering. In practice, this is almost impossible to give a priori, and most computer systems for completion

allow the user to interactively orient the rules. However, this may also be very difficult. For most rules in the example above the choice was clear, but for r_{15} the other orientation was as plausible as the chosen one. But the other orientation would have led to non-termination of the produced TRS.

Finally, we address the question of uniqueness of the produced TRS.

Definition 5.9 Let $>$ be a reduction ordering. A TRS $\mathcal{R}$ is said to be *compatible* with $>$ if for every rule $l \to r$ of $\mathcal{R}$ we have $l > r$. □

Theorem 5.10 (Métivier [52]) *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two complete irreducible TRSs compatible with a given reduction ordering $>$. Suppose $s =_{\mathcal{R}_1} t$ iff $s =_{\mathcal{R}_2} t$. Then $\mathcal{R}_1$ and $\mathcal{R}_2$ are identical up to renaming of variables.*

Thus, the output of the optimized version of the Knuth-Bendix procedure is unique.

Exercises

1. Let E be the set of axioms:

$$f(f(f(x))) = x$$
$$f(f(f(f(f(x))))) = x$$

Prove that $E \vdash f(x) = x$.

2. Let E be the set of axioms:

$$x+0=x$$
$$x+s(y)=s(x+y)$$
$$x\times 0=0$$
$$x\times s(y)=(x\times y)+x$$

Find an interpretation which is a model of E but not of the equation $x+y=y+x$.

3. Let E be the set of axioms:

$$x+0=x$$
$$x+s(y)=s(x+y)$$

(a) Prove that $0+x=x$ is not a logical consequence of E.

(b) Prove, for every ground term t over $\{0, s\}$, that $0+t=t$ is a logical consequence of E.

4. Let E be the set of axioms:

$$\begin{aligned} &x + 0 = x \\ &x + s(y) = s(x + y) \\ &x + y = y + x \end{aligned}$$

   Give a TRS R that solves the word problem for $(\{0, s, +\}, E)$, i.e. such that for ground terms s and t: $s =_R t$ iff $E \models s = t$, and such that R is confluent and terminating for ground terms.

5. Let E be the set of axioms:

$$\begin{aligned} &0 + x = x \\ &x + -x = 0 \\ &(x + y) + z = x + (y + z) \end{aligned}$$

   (a) Show a few initial steps of Knuth-Bendix completion procedure applied to E,

   (b) Investigate whether $E \models x + 0 = x$.

Chapter 6

Unification

The *unification* problem can in its simplest form be phrased as follows:

> Given two terms s and t, is there a substitution σ such that $\sigma(s) \equiv \sigma(t)$?

The need for solving this problem arises frequently in (AI) applications such as deductive databases, expert systems, natural language processing, logic programming, knowledge representation, etc. (see Siekmann [65] for a extensive list of applications). We have already encountered the problem in sections 3.5 and 5.4, where unification was an essential part of the process of finding critical pairs.

When our goal is to make the two terms s and t identical, the problem is referred to as the *syntactic* unification problem. In this chapter, we will study the unification problem in a more general perspective: Given a set E of equational axioms and two terms s and t, is there a solution to the equation $s = t$, i.e. is there a substitution σ such that

$$\sigma(s) =_E \sigma(t)$$

We refer to this generalized problem as the E-unification problem. Note that by letting $E = \emptyset$, we obtain the syntactic unification problem as a special case.

To exemplify, consider the terms $s \equiv +(x, +(a, b))$ and $t \equiv +(+(y, b), x)$. To begin with, we let $E = \emptyset$. In this case s and t are unifiable, and the most general unifier (see definition 3.9) is the substitution $\sigma_1 = \{y \mapsto a, x \mapsto +(a, b)\}$

Now assume that our intented interpretation is that the operator $+$ is a set-constructor, i.e. that a term $+(a, b)$ denotes the set $\{a, b\}$. Then we do not want to distinguish between the terms $+(a, b)$ and $+(b, a)$, since the operator $+$ is *commutative*. We express this by letting $E = \{+(x, y) = +(y, x)\}$. Now

σ_1 is still a unifier of s and t, but $\sigma_2 = \{y \mapsto a\}$ should be a more general unifier, since

$$\sigma_2(s) \equiv +(x, +(a, b)) =_E +(+(a, b), x) \equiv \sigma_2(t)$$

Obviously σ_2 is not a unifier in the sense of definition 3.9, so we have to extend the notion of a unifier in some way. We say that σ_2 (and σ_1) is an *E-unifier* of s and t (see definition 6.5).

An interesting question is how we can represent the set of E-unifiers of two terms s and t. As we will see in the next section, in the syntactic unification case there is always a most general unifier, implicitly representing all other unifiers of s and t. This is not true in general for commutative unification, but it is at least always possible to find a finite set of E-unifiers which represent all other E-unifiers of the two terms.

The situation changes again if we assume that $+$ is associative, i.e. we let $E = \{+(+(x, y), z) = +(x, +(y, z))\}$. This is the case if (for instance) we interpret all constants as characters, and interpret $+$ as the concatenation operator. Now σ_1 (but not σ_2!) is again an E-unifier of s and t, but so is $\tau_1 = \{x \mapsto +(+(a, b), +(a, b)), y \mapsto a\}$, since:

$$+(+(+(a, b), +(a, b)), +(a, b)) =_E +(+(a, b), +(+(a, b), +(a, b)))$$

But, as the reader may verify, $\tau_2 = \{x \mapsto +(+(a, b), +(+(a, b), +(a, b))), y \mapsto a\}$ is also an E-unifier of s and t. In fact, it is not difficult to see that there are *infinitely* many E-unifiers of s and t, which cannot be expressed as instances of each other. In this sense, associative unification and commutative unification are very different in nature.

The perhaps most studied case of E-unification is when the operator $+$ is assumed to be both commutative and associative. This case arises if (for instance) we interpret all constants as sets, and the operator $+$ as set union. Here it is again possible to find a finite set of E-unifiers that represent all E-unifiers of two terms.

The study of the unification problem in various equational theories constitutes the core of *unification theory*, which is briefly touched upon in section 6.2. A more concrete task is to find procedures that actually compute all E-unifiers of two terms. We will present such procedures in sections 6.3 and 6.4.

6.1 Syntactic unification

In 1930, Herbrand [27] presented an algorithm to solve equations over ground terms. This algorithm may be regarded as the first unification algorithm, although Herbrand did not explicitly introduce the notion of a unifier. The modern study of unification started with the seminal paper by Robinson [61], who gave an algorithm to compute the most general unifier of two terms. However, the definition of a most general unifier which has become predominant in the literature (definition 3.9) seems to have appeared first in Chang and Lee [8].

Robinson's initial algorithm was exponential both in time and space. Since 1965, considerable effort has been devoted to finding better algorithms. Huet [30] presented an almost linear algorithm in his thesis. His method is based on efficient programming techniques for manipulating equivalence relations, and runs in $O(n\alpha(n))$ time, where $\alpha(n)$ is an extremely slowly growing function. By adding a clever pointer scheme to Huet's method, Paterson and Wegman [57] succeeded in finding the first truly linear unification algorithm. Independently, Martelli and Montanari [49] found another linear unification algorithm based on slightly different principles.

Below we give a non-deterministic unification algorithm, based on Herbrand's original algorithm. The presentation is due to Martelli and Montanari [50], who used this description as a starting point when deriving a unification algorithm which is claimed to be very efficient in practice.

It is interesting to note that although several linear unification algorithms exist, most implementations (e.g. of the language PROLOG) still use a version of Robinson's original algorithm. Efficiency is obtained by sacrificing soundness; the so-called *occur-check* test is usually omitted (in algorithm 6.2, this amounts to not testing if x occurs in t in case 5).

As mentioned before, we will view unification as equation solving. To unify two terms s and t, we will start with the singleton set $\{s = t\}$, and in a number of steps transform this set into a set in *solved form*. From this solved form we may then extract a most general unifier.

Definition 6.1 [Solved form] A (possibly empty) set U of equations is said to be on *solved form* if U is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$, where the x_i's are distinct variables not occurring in any right hand side of any equation in U. □

Solved forms can be computed with the following non-deterministic algorithm.

Algorithm 6.2 [Solved Form Algorithm] Given a set of equations, select non-deterministically an equation e, and perform the transformation which corresponds to the form of e.

1. $f(s_1, ..., s_n) = f(t_1, ..., t_n)$:
   replace by the equations $s_1 = t_1, \ldots, s_n = t_n$

2. $f(s_1, ..., s_n) = g(t_1, ..., t_m)$, where $f \not\equiv g$:
   halt with failure
3. $x = x$:
   delete the equation
4. $t = x$, where t is not a variable :
   replace by the equation $x = t$
5. $x = t$, where $t \not\equiv x$ and x has another occurrence in the set of equations:
   if x appears in t, then halt with failure;
   otherwise, replace x by t in every other equation.

Repeat until no case applies to any equation, then halt with success. □

It is easily verified that the solved form algorithm always terminates, and returns either failure or a set of equations in solved form (note that the algorithm, seen as an ARS, is confluent).

Let $U = \{x_1 = u_1, \ldots, x_n = u_n\}$ be the solved set of equations obtained by applying the solved form algorithm to the set $\{s = t\}$. Then we can straightforwardly extract the substitution $\sigma = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$. As we will show, σ is in fact a most general unifier of s and t.

First note that, by the definition of solved form, no x_i occurs in any u_j. We say that σ is *idempotent* (an equivalent definition is that $\sigma = \sigma\sigma$). We will use this fact in the proof of the following theorem, which shows that the solved form algorithm actually computes a most general unifier.

Theorem 6.3 Let $U = \{x_1 = u_1, \ldots, x_n = u_n\}$ be the solved set of equations obtained by applying the solved form algorithm to the set $\{s = t\}$. Then the substitution $\sigma = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$ is a most general unifier of s and t.

Proof : It is helpful to generalize the notion of unifier in the following way: If $E = \{s_1 = t_1, \ldots, s_k = t_k\}$ is a set of equations, then θ is a unifier for E if $\theta(s_1) \equiv \theta(t_1), \ldots, \theta(s_k) \equiv \theta(t_k)$.

We first prove that σ is a unifier of s and t. Note that if E is any set of equations, and E' is a set of equations obtained by transforming E once with one of the steps in the solved form algorithm, then θ is a unifier for E iff θ is a unifier for E' (this is easily verified with a simple case analysis). Since σ obviously is a unifier of U, we conclude that σ is also a unifier of $\{s = t\}$.

We proceed to prove that σ is an mgu. Let τ be any unifier of U (and hence a unifier of s and t). Then for every i, $\tau(x_i) \equiv \tau(u_i)$, and since σ is idempotent, $\tau(\sigma(x_i)) \equiv \tau(u_i) \equiv \tau(x_i)$. If x is a variable different from $x_1 \ldots x_n$, then certainly $\sigma(x) \equiv x$ and $\tau(\sigma(x)) \equiv \tau(x)$. Consequently $\tau = \tau\sigma$ and $\sigma \leq \tau$. Since τ was arbitrarily chosen, we conclude that σ is an mgu of U, and hence an mgu of s and t. □

Example 6.4 When unifying the two terms $f(g(y), a, y)$ and $f(x, a, g(z))$, the solved form algorithm could operate as follows:

$$\begin{array}{ll}
\{f(g(y),a,y) = f(x,a,g(z))\} & \\
\{g(y) = x,\ a = a,\ y = g(z)\} & \text{(step 1)} \\
\{g(y) = x,\ y = g(z)\} & \text{(step 1)} \\
\{x = g(y),\ y = g(z)\} & \text{(step 4)} \\
\{x = g(g(z)),\ y = g(z)\} & \text{(step 5)}
\end{array}$$

The result is in solved form, and the most general unifier $\{x \mapsto g(g(z)), y \mapsto g(z)\}$ can be extracted. □

A well-known and often quoted fact is that *a most general unifier of two terms is unique up to renaming of variables*. This means that if γ and τ are two mgus of s and t, then there exist two variable renaming substitutions (see section 3.3) θ and ρ, such that $\theta\gamma = \tau$ and $\rho\tau = \gamma$.[1] The proof of this fact is not difficult but somewhat techniqual. The reader is referred to Lassez *et al* [45].

To summarize the most important properties of syntactic unification:

- It is decidable whether two terms s and t are unifiable or not.
- It is possible to compute a most general unifier, implicitly representing all other unifiers of s and t.
- Unification can be done deterministically (due to the confluence of the solved form algorithm).

6.2 E-unification

We now would like to generalize the discussion as described in the beginning of this chapter. Our starting point is a set E of equational axioms. As explained in chapter 5, E presents an equality theory $=_E$, and we now address the problem of performing unification modulo this equality theory. We first discuss some general properties of substitutions.

It is natural to extend the $=_E$ relation to substitutions as follows:

$$\sigma =_E \tau \quad \text{if} \quad \forall x : \sigma(x) =_E \tau(x)$$

[1]The definition of a mgu, and the property of being "unique up to renaming of variables" has been subject to some confusion in the literature. The excellent essay by Lassez *et al.* [45] explains the different existing definitions of mgus, and the different corresponding notions of variable renaming substitution.

Note that in the extreme case when $E = \emptyset$, two substitutions are equal only when they are identical. We further generalize the "more-general" relation:

$$\sigma \leq_E \tau \quad \text{if} \quad \exists\gamma : \gamma\sigma =_E \tau$$

Note that $\sigma \leq_E \tau$ and $\tau \leq_E \sigma$ does *not* imply that $\sigma =_E \tau$ (see exercise 2). Instead we introduce the following relation:

$$\sigma \sim_E \tau \quad \text{if} \quad \sigma \leq_E \tau \quad \text{and} \quad \tau \leq_E \sigma$$

Next we generalize the notion of unifier.

Definition 6.5 [**E-unifier**] Let E be a set of equational axioms. A substitution σ is an *E-unifier* of two terms s and t if $\sigma(s) =_E \sigma(t)$. If $U = \{s_1 = t_1, \ldots, s_n = t_n\}$ is a set of equations, then σ is an E-unifier for U if $\sigma(s_1) =_E \sigma(t_1), \ldots, \sigma(s_n) =_E \sigma(t_n)$. □

Unfortunately, unlike the syntactic unification case there does not (in general) exist a most general E-unifier of two terms s and t. The closest we get to such a canonical representation of all unifiers is a *complete and minimal set S of E-unifiers*. Intuitively, *complete* means that every E-unifier of s and t is an instance of some E-unifier in S, and *minimal* means that S contains no redundancy.

Definition 6.6 [**Complete and minimal set of E-unifiers**] Let s and t be two terms, and let S be a set of E-unifiers of s and t.

1. S is said to be *complete* if for every E-unifier γ of s and t, there exists $\sigma \in S$ such that $\sigma \leq_E \gamma$
2. S is said to be *minimal* if $\forall\sigma, \gamma \in S : \sigma \leq_E \gamma \Rightarrow \sigma = \gamma$

□

Note that in definition 6.6:2 it says $\sigma = \gamma$ and not $\sigma =_E \gamma$.

Example 6.7 Let $E = \{x + y = y + x\}$ be a set of axioms, and consider the equation $0{+}x = y{+}z$. Now $\sigma_1 = \{y \mapsto 0,\ z \mapsto x\}$ and $\sigma_2 = \{z \mapsto 0,\ y \mapsto x\}$ are two E-unifiers such that neither $\sigma_1 \leq_E \sigma_2$ nor $\sigma_2 \leq_E \sigma_1$. It is easily seen that for every other E-unifier τ, we have either $\sigma_1 \leq_E \tau$ or $\sigma_2 \leq_E \tau$. As an example, consider the E-unifier $\sigma_3 = \{y \mapsto 0,\ x \mapsto 0,\ z \mapsto 0\}$, which is an instance of both σ_1 and σ_2. Thus $\{\sigma_1, \sigma_2\}$ is a complete and minimal set of E-unifiers to $0{+}x = y{+}z$. The set $\{\sigma_1\}$ is minimal but not complete, $\{\sigma_1, \sigma_2, \sigma_3\}$ is complete but not minimal, and finally $\{\sigma_1, \sigma_3\}$ is neither complete nor minimal. □

The following theorem states that a complete and minimal set of E-unifiers (when it exists) is "more or less" unique.

Theorem 6.8 If S_1 and S_2 are complete and minimal sets of E-unifiers of s and t, then there exists a bijection $b : S_1 \to S_2$ such that $\forall \sigma \in S_1 : \sigma \sim_E b(\sigma)$.

Proof : See exercise 4. □

Fages and Huet [23] showed the surprising result that complete and minimal sets of E-unifiers need not always exist. That is, for some terms s and t, every attempt to list the E-unifiers of s and t will either be incomplete, or contain redundant unifiers.

Theorem 6.9 There exists a set of axioms E and two terms s and t, such that there exists no complete and minimal set of E-unifiers of s and t.

Proof : Let E be the following set of axioms

$$\{f(0,x) = x, \quad g(f(x,y)) = g(y)\}$$

where f and g are function symbols, 0 is a constant, and x and y are variables. Let $s \equiv g(x)$ and $t \equiv g(0)$. We will now show that there does not exist a complete and minimal set of E-unifiers of s and t.

First we construct the TRS R:

$$\begin{cases} f(0,x) \to x \\ g(f(x,y)) \to g(y) \end{cases}$$

It is easily verified that R is both confluent and terminating. Hence we have that $u_1 =_E u_2$ iff u_1 and u_2 have the same normal form in R (see section 5.3). Now let

$$\begin{aligned} \sigma_0 &= \{x \mapsto 0\} \\ \sigma_1 &= \{x \mapsto f(x_1, 0)\} \\ \sigma_2 &= \{x \mapsto f(x_2, f(x_1, 0))\} \\ &\vdots \end{aligned}$$

where $x_1, x_2, \ldots$ are variables. The proof is now accomplished in three steps:

1. We show that every σ_i is an E-unifier of s and t;
2. We show that the set $S = \{\sigma_i \mid i \geq 0\}$ is a complete set of E-unifiers of s and t;
3. We show that for all $i \geq 1$, $\sigma_i <_E \sigma_{i-1}$.

These three observations will directly enable us to prove the theorem.

1. Every σ_i is an E-unifier of s and t since
$$\sigma_i(g(x)) \equiv g(f(x_i, f(x_{i-1}, \ldots, f(x_1, 0) \ldots)))$$
and by applying the second rule in R i times, one obtains $g(0)$. Thus $\sigma_i(g(x)) =_E \sigma_i(g(0))$.

2. Let τ be an E-unifier of s and t. Obviously, $\tau = \{x \mapsto u\}$ for some term u. We prove by structural induction on u that there exists a $\sigma_i \in S$ such that $\sigma_i \leq_E \tau$.
   - If u is a variable or a constant, then $g(u)$ cannot be rewritten with any of the rules in R, so $g(u) =_E g(0)$ only if $u \equiv 0$. Then $\sigma_0 \leq_E \tau$.
   - If $u \equiv g(s)$, then obviously τ is not a E-unifier of s and t. This case will not arise.
   - If $u \equiv f(s_1, s_2)$, then $g(u) =_E g(s_2)$ by the second axiom. By the induction hypothesis, there is a $\sigma_j \in S$ such that $\sigma_j \leq_E \{x \mapsto s_2\}$. Then for $i = j+1$ we have $\sigma_i \leq_E \tau$, since $\sigma_i(x) \equiv f(x_i, \sigma_j(x))$, but $\tau(x) \equiv f(s_1, \sigma_j(x))$. Thus $\tau =_E \{x_i \mapsto s_1\} \circ \sigma_i$, which is the same as saying that $\sigma_i \leq_E \tau$.

3. For $i \geq 1$, let $\tau_i = \{x_i \mapsto 0\}$. Then $\tau_i(\sigma_i(x)) \equiv f(0, \sigma_{i-1}(x))$, and since $f(0, \sigma_{i-1}(x)) =_E \sigma_{i-1}(x)$ by the first axiom, we conclude that $\sigma_i \leq_E \sigma_{i-1}$.

   On the other hand, $\sigma_{i-1} \not\leq_E \sigma_i$. Consider $\sigma_{i-1}(x) \equiv f(x_{i-1}, \ldots, f(x_1, 0))$. It is not possible to instantiate this term to obtain a term which can be rewritten into $\sigma_i(x)$, using the rules in R. Thus $\sigma_i <_E \sigma_{i-1}$.

Using observations 1–3, we can now conclude the proof: Let S' be any complete set of E-unifiers of s and t. Since S is complete, we have that $\forall \tau \in S' \; \exists \sigma_i \in S : \sigma_i \leq_E \tau$, and since $\sigma_{i+1} <_E \sigma_i$, we obtain $\sigma_{i+1} <_E \tau$. Since S' is complete, we have that $\exists \eta \in S' : \eta \leq_E \sigma_{i+1}$, and hence $\eta <_E \tau$: S' is not minimal. □

An equational theory E can be classified according to the properties of its complete and minimal sets of E-unifiers:

Definition 6.10 Let E be a set of equational axioms, and let S denote a complete and minimal set of E-unifiers for two terms s and t. Then E is said to be

- *unitary* if S exists and has at most one element for every pair of terms s and t;

- *finitary* if S exists and is finite for every s and t;

- *infinitary* if S exists for all s and t, and S is infinite for some choice of s and t;

- *nullary* if S does not exist for some choice of s and t

□

We have seen examples of all four types; syntactic unification is unitary, commutative and associative-commutative unification is finitary (see section 6.4), associative unification is infinitary, and the theory in the proof of theorem 6.9 is nullary. This classification is important in the field of *unification theory*, as the type of an equational theory tells us how complex the unification problem is, if we can hope to find a terminating unification algorithm, etc.

The general E-unification problem, as phrased in the beginning of this chapter, is undecidable. (We recall its formulation: Given E, s and t, is there a substitution σ such that $\sigma(s) =_E \sigma(t)$?) The problem is even undecidable for some specific E, for instance if we let E be an axiomatization of the natural numbers with $+$ and $\times$. The E-unification problem then amounts to determining the solvability of Diophantine equations, which was shown to be undecidable by Matiyasevitch [51]. Of course, E-unification is decidable for many E, and in yet some other cases decidability is a difficult open problem.

Generally, unification theory deals with the following three problems. Given a set of equational axioms E,

1. is the E-unification problem decidable?

2. what is the type of E?

3. construct a unification procedure that enumerates a complete set of E-unifiers. Preferably, the set should also be minimal. If E is finitary, then the procedure should preferably be terminating.

The reader is referred to Siekmann [65] for an overview of decidability results and a type classification of various equational theories.

As concerns item 3, unification procedures have been conceived for many interesting theories (see section 6.4 for two examples). An interesting question is the following: *Is there a* general *unification procedure, that on input s, t and E computes a complete set of E-unifiers of s and t?* In the next section, we will present a general unification procedure that works for many (but not all) equational theories.

6.3 Narrowing

Given two terms s and t, and a set of equational axioms E, how do we compute the E-unifiers of s and t? The *narrowing* procedure is based on the following two observations:

1. Every syntactic unifier of s and t is also an E-unifier of s and t.

2. Suppose R is a confluent and terminating TRS defining the same equality theory as E, i.e. $\forall a, b : a =_E b \Leftrightarrow a =_R b$. ($R$ can for instance be construced from E with the Knuth-Bendix procedure). Then $a =_E b$ iff a and b have the same normal form in R (see section 5.3). Thus we may conclude that σ is an E-unifier of s and t if $\sigma(s)$ and $\sigma(t)$ has the same normal form in R.

Using observation 1, we may generalize observation 2: if $\sigma(s)$ and $\sigma(t)$ have normal forms which are syntactically unifiable with mgu τ, then $\tau\sigma$ is an E-unifier of s and t.

The question is now: How do we find the appropriate substitutions σ? If s and t are not directly syntactically unifiable, we have to instantiate s and t in such a way that a subterm in s or in t (or both) can be rewritten with a rule in R. This is attained by syntactically unifying a non-variable subterm in s (or t) with a left hand side of some rule in R. The following definition captures the intuition:

Definition 6.11 [Narrowing relation] A term t is said to be *narrowable* to t' iff there is a position $p \in \overline{O}(t)$, a renamed version [2] $l \to r$ of a rule in R, and a substitution σ such that

- σ is a (syntactical) mgu of $t_{|p}$ and l

- $t' \equiv \sigma(t[r]_p)$

We write $t \rightsquigarrow_{[p,l\to r,\sigma]} t'$, or more simply $t \rightsquigarrow_\sigma t'$. $\rightsquigarrow$ is called the *narrowing relation.* □

Note that narrowing can be viewed as a generalization of term rewriting, where matching is replaced by unification.

Notation: We write $t_1 \rightsquigarrow^*_\sigma t_n$ if there exists a narrowing derivation

$$t_1 \rightsquigarrow_{\sigma_1} t_2 \rightsquigarrow_{\sigma_2} \dots \rightsquigarrow_{\sigma_{n-1}} t_n$$

[2] For completeness of the narrowing procedure we require that the variables in the rule used are renamed into "fresh" variables (cf. definition 3.13).

such that $\sigma = \sigma_{n-1} \dots \sigma_2\sigma_1$. If n=1, then σ is the empty substitution. □

For the sake of the presentation, it is useful to extend the signature with a new binary function symbol $=^?$ and a new constant *true*. Furthermore, we assume that R contains the rule $x =^? x \to true$. Terms of the form $s =^? t$ will be called *goals*. To E-unify two terms s and t, we try to construct all narrowing derivations $s =^? t \leadsto_\sigma^* true$. As we will see, σ is an E-unifier of s and t.

Example 6.12 Let E be the set of axioms

$$\{0 + x = x, \quad s(x) + y = s(x + y)\}$$

We construct the confluent and terminating TRS R by orienting the equations:

$$\begin{cases} 0 + x \to x \\ s(x) + y \to s(x + y) \end{cases}$$

Suppose we want to compute the E-unifiers of $z + z$ and $s(s(0))$. Figure 6.1 shows that we can find the only E-unifier $\{z \mapsto s(0)\}$ by systematically constructing all narrowing derivations from the goal $z + z =^? s(s(0))$.
□

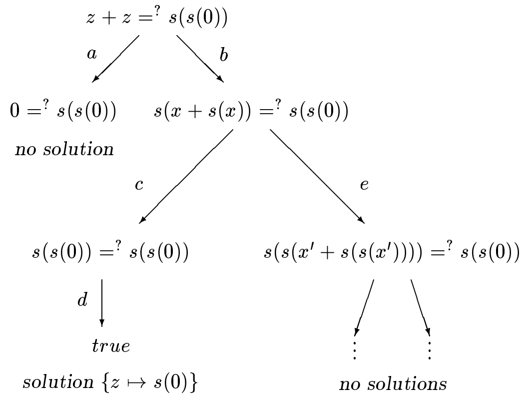
Theorem 6.13 [Soundness of narrowing] Let E be a set of axioms, and let R be a TRS defining the same equality theory as E. If $s =^? t \leadsto_\sigma^* true$, then σ is an E-unifier of s and t.

Proof : See exercise 7. □

The proof of the soundness theorem is simple. It is much more complicated to show that narrowing computes *all* E-unifiers, given that the TRS R is confluent and terminating. The following theorem was first given by Hullot [35]. For a proof we refer to Middeldorp [53].

Theorem 6.14 [Completeness of narrowing] Let E be a set of axioms, and let R be a *confluent and terminating* TRS defining the same equality theory as E. If $\sigma(s) =_E \sigma(t)$, then there exists a narrowing derivation $s =^? t \leadsto_\tau^* true$, such that $\tau \leq_E \sigma_{|Var(s) \cup Var(t)}$ (where $\sigma_{|Var(s) \cup Var(t)}$ means the restriction of σ to the variables occurring in s and t).

That is, by collecting all E-unifiers computed by narrowing, we obtain a complete (but not necessarily minimal) set of E-unifiers. We say that narrowing is *complete* for confluent and terminating TRSs.



step	rewrite rule	narrowing substitution
a	$0 + x \to x$	$\{x \mapsto 0,\ z \mapsto 0\}$
b	$s(x) + y \to s(x + y)$	$\{y \mapsto s(x),\ z \mapsto s(x)\}$
c	$0 + x' \to x'$	$\{x \mapsto 0,\ x' \mapsto s(0)\}$
d	$x =^? x \to true$	$\{x \mapsto s(s(0))\}$
e	$s(x') + y' \to s(x' + y')$	$\{x \mapsto s(x'),\ y' \mapsto s(s(x'))\}$

Figure 6.1: The search space for the goal $z + z =^? s(s(0))$

Narrowing was initially proposed by Slagle [66] in the more general context of first-order theorem proving. Fay [24] was the first to (essentially) make the the two observations in the beginning of this section. Hullot introduced (in 1980) the narrowing relation, and stated the completeness theorem above. It was soon realized that narrowing was well suited as operational semantics for programming languages that combine logic programming and functional programming. In the decade that followed, many such languages were proposed (see Hanus [25] for a recent and well-written survey).

However, the search space of narrowing is too large for any practical purposes. In fact, narrowing seldom terminates. Therefore several optimizations have been proposed; here we can only mention the most important ones (we refer again to Hanus [25] for a more complete treatment of the subject).

A first idea is to restrict the number of subterms at which narrowing is performed. Hullot [35] introduced the so-called *basic* narrowing, in which narrowing is never applied to a subterm introduced by previous narrowing substitutions (that is to a subterm that appeared as the binding of a variable at some previous steps). This results in a dramatic cut of the search space, without losing completeness. The formal definition is somewhat involved, and refers to the prefix ordering of positions (definition 3.7).

Definition 6.15 [Basic narrowing] A narrowing sequence

$$t_1 \rightsquigarrow_{[p_1, l_1 \to r_1, \sigma_1]} \cdots \rightsquigarrow_{[p_{n-1}, l_{n-1} \to r_{n-1}, \sigma_{n-1}]} t_n$$

is *basic* iff $p_i \in B_i$ $(1 \leq i \leq n-1)$, where the sets of positions $B_1, \ldots, B_{n-1}$ are definied by

$$B_1 = \bar{O}(t_1)$$
$$B_{i+1} = (B_i - \{q \in B_i \mid p_i \leq q\}) \cup \{p_i w \mid w \in \bar{O}(r_i)\} \qquad \square$$

Example 6.16 Consider the one-rule TRS

$$f(f(x)) \to x$$

If we try to solve (the unsolvable) goal $f(z) =^? z$ with ordinary narrowing, the computation loops even though the TRS is terminating:

$$f(z) =^? z \ \rightsquigarrow_{\{z \mapsto f(x)\}} \ x =^? f(x) \ \rightsquigarrow_{\{x \mapsto f(x')\}} \ f(x') =^? x' \ \ldots$$

In the second step, narrowing is performed at the subterm $f(x)$, which was introduced by the preceding narrowing substitution. Therefore the derivation is not basic. The only basic narrowing derivation is

$$f(z) =^? z \ \rightsquigarrow_{\{z \mapsto f(x)\}} \ x =^? f(x)$$

Since x and $f(x)$ are not syntactically unifiable, we may conclude that the initial goal is not solvable. Thus the infinite search space was reduced to a (small) finite one. $\square$

The restriction to basic positions has no effect on the completeness; basic narrowing is complete for *confluent and terminating* TRSs (for a proof, see Middeldorp [53]).

A second idea for reducing the search space of narrowing is to mix narrowing steps with rewriting steps (i.e. where the variables in the goal are not instantiated). Since R is confluent and terminating, rewriting is deterministic. Therefore it is reasonable to rewrite the goal into normal form before every

narrowing step (this is usually referred to as *normalizing narrowing*, and was introduced by Fay [24]).

Finally, a third idea is to try to prune useless branches (i.e. those not leading to any solutions) in the search space. Consider example 6.12, and the infinite (rightmost) branch in the search space. After a couple of steps, we reach the goal

$$s(s(s(x'' + s(s(s(x'')))))) =^? s(s(0))$$

The term on the left has the subterm $s(x'' + s(s(s(x''))))$, while the corresponding subterm in the term to the right is 0. Since a term of the form $s(\ldots)$ never can be rewritten into 0 with the rules in R, we conclude that the branch is useless and can be pruned (making the search space finite).

6.4 Some unification algorithms

The narrowing procedure, as described in the previous section, is a *general* unification procedure. Given two terms s and t, and a set E of axioms, it computes all E-unifiers of s and t, i.e. all substitutions σ such that $\sigma(s) =_E \sigma(t)$. However, narrowing has some properties that limit its usefulness:

1. Narrowing is only complete (i.e. computes all unifiers) if E can be compiled into a confluent and terminating term rewriting system. This excludes, among others, all theories containing the commutativity axiom $f(x, y) = f(y, x)$.

2. Narrowing is not a unification *algorithm*, i.e. it is in general unable to answer "no" when s and t are *not* unifiable, even if the unification problem is decidable.

These drawbacks render it interesting to look for specialized unification algorithms that only work for one, or a limited number of equational theories. In fact, the first such algorithm can be dated as far back as 1847, when Boole presented a unification algorithm for boolean rings [4]. Since then, unification algorithms have been devised for may equational theories (or the unification problem has been shown to be undecidable; see Siekmann [65] for a survey). In this chapter we will briefly discuss two theories: commutative (C) unification, and associative-commutative (AC) unification.

C-unification

The commutativity (C) axiom

$$f(x, y) = f(y, x)$$

implies that arguments of the function symbol f can be swapped. Given a term t, e.g.

$$f(a, f(b, f(f(x, c), a)))$$

with k occurrences of f, there are 2^k terms s such that $s =_C t$ (for every occurrence of f we can choose either to swap or not to swap its arguments). We call every such term s a *C-permutation* of t. The example term above has $2^4 = 16$ C-permutations.

This observation suggests the following (naïve) unification algorithm:

1. Given two terms; s and t, compute all C-permutations of s and t.

2. Unify syntactically every C-permutation of s with every C-permutation of t. When the unification is successful, giving a most general unifier σ, add σ to a set U.

3. By filtering away all duplicate substitutions in U, we obtain a complete and minimal set (see definition 6.6) of C-unifiers of s and t.

It is easily realized that C is a finitary theory (see definition 6.10), and that the above algorithm indeed computes all C-unifiers. Practically however, the algorithm is very inefficent; if s contains i occurrences of f, and t contains j occurrences of f, the algorithm will perform $2^i \cdot 2^j$ unifications, at least half of which are unnecessary.

AC-unification

Associative-commutative (AC) unification, i.e. unification modulo the axioms

$$f(f(x, y), z) = f(x, f(y, z)) \qquad f(x, y) = f(y, x)$$

has applications primarily in automated theorem proving. AC is a finitary theory, and Stickel gave an algorithm in 1975 that returns a complete and minimal set of AC-unifiers of two terms [67]. It should be noted that the associativity axiom complicates matters, and two terms may have several unobvious AC-unifiers. For example, consider the equation

$$f(x, y) = f(a, z)$$

One AC-unifier is $\{x \mapsto f(a, v), z \mapsto f(v, y)\}$, where v is a new variable. This shows that the value of z does not have to be wholly contained in either the value of x or in the value of y.

We now describe Stickel's algorithm. First note that we can "flatten" a term by removing all nested occurrences of f. For instance, the term $f(f(a, b), f(f(x, g(y)), b))$ can be written as $f(a, b, x, g(y), b)$. This is because the associativity axiom implies that nesting of f's is irrelevant. Note also that the commutativity axiom implies that the order in which we list the arguments is also irrelevant. All this is easy to understand if we note the analogy with ordinary addition; one usually writes $a + b + c$ rather that $+(+(a, b), c)$.

Another prepatory step is to remove arguments which occur in both terms to be unified. For instance, from $f(a, b, x, g(y), b)$ and $f(b, x, x, x, z)$ we obtain $f(a, b, g(y))$ and $f(x, x, z)$ by removing one occurrence of b and one occurrence of x from each term. It is easily realized that the set of unifiers is not changed by this transformation (see exercise 8).

First assume that the arguments are all variables (the general case will be treated later). Stickel's key idea is now that, given two terms s with variables $x_1, \ldots x_m$ and t with variables $y_1, \ldots y_n$, we set up a linear Diophantine equation

$$c_1 x_1 + \ldots + c_m x_m = d_1 y_1 + \ldots + d_n y_n \tag{6.1}$$

where c_i is the number of occurrences of x_i in s, and d_i is the number of occurrences of y_i in t. For instance, from the unification problem

$$f(x_1, x_1, x_2, x_3) = f(y_1, y_1, y_2) \tag{6.2}$$

we get the Diophantine equation

$$2x_1 + x_2 + x_3 = 2y_1 + y_2 \tag{6.3}$$

The meaning of this Diophantine equation is not hard to understand. If σ is a AC-unifier of s and t, and σ assigns some variable the value u, then u must be present an equal number of times in $\sigma(s)$ and $\sigma(t)$. Thus, every AC-unifier in (6.2) corresponds to a non-negative solution to equation (6.3).

An equation of the form (6.1) has an infinite number of solutions, but all solutions can be expressed as a sum of elements of a particular finite subset of all solutions; a so-called *basis* for the equation. Stickel gave a straightforward method of generating this basis in [67]; a much improved method was later given by Huet [31]. For small equations, the basis is not difficult to generate by hand. For instance, the basis for equation (6.3) is given below.

Row	x_1	x_2	x_3	y_1	y_2	$2x_1 + x_2 + x_3$	$2y_1 + y_2$	
1	0	0	1	0	1	1	1	z_1
2	0	1	0	0	1	1	1	z_2
3	0	0	2	1	0	2	2	z_3
4	0	1	1	1	0	2	2	z_4
5	0	2	0	1	0	2	2	z_5
6	1	0	0	0	2	2	2	z_6
7	1	0	0	1	0	2	2	z_7

We associate a new variable z_i to each row i. Every solution to equation 6.3 can be expressed as (a multiple of) a sum of some basis elements (rows in the table). In the example, there are $2^7 = 128$ such sums. As an example, we may choose to add row 1,5 and 6, which corresponds to the following subtable:

Row	x_1	x_2	x_3	y_1	y_2	$2x_1 + x_2 + x_3$	$2y_1 + y_2$	
1	0	0	1	0	1	1	1	z_1
5	0	2	0	1	0	2	2	z_5
6	1	0	0	0	2	2	2	z_6

The corresponding unifier to equation 6.2 is obtained by reading down every variable's column. For instance, x_1 is bound to z_6, x_2 is bound twice to z_5 (that is, x_2 is bound to $f(z_5, z_5)$), etc. The unifier is:

$$\{x_1 \mapsto z_6,\ x_2 \mapsto f(z_5, z_5),\ x_3 \mapsto z_1,\ y_1 \mapsto z_5,\ y_2 \mapsto f(z_1, z_6, z_6)\}$$

Since every variable must be bound to something, we only have to consider those sums where no column only contains zeros. This leaves us with 69 sums, each of which corresponds to a unifier in the above sense.

This completes the description of Stickel's algorithm in the variable-only case. We now move to describe the general case. When we want to unify two terms s and t, we first create two "generalized" terms s' and t' by uniformly replacing arguments in s and t with new variables. For example, if $s \equiv f(x, x, y, a)$ and $t \equiv f(b, b, z)$, then $s' \equiv f(x_1, x_1, x_2, x_3)$ and $t' \equiv f(y_1, y_1, y_2)$. The next step is to AC-unify s' and t' with the variable-only algorithm, which gave us 69 unifiers in our example. Given such a unifier σ, we syntactically unify the value of each $\sigma(x_i)$ and $\sigma(y_j)$ with the original value of x_i and y_j, respectively. In the example, we thus make the following unifications:

$$x = z_6,\ y = f(z_5, z_5),\ a = z_1,\ b = z_5,\ z = f(z_1, z_6, z_6)$$

giving us the unifier

$$\{x \mapsto z_6,\ y \mapsto f(b, b),\ z \mapsto f(a, z_6, z_6)\}$$

or rather

$$\{y \mapsto f(b, b),\ z \mapsto f(a, x, x)\}$$

A useful observation is that any unifier of s' and t' that binds x_3 or y_1 to something else than a variable cannot be a unifier of s and t, because an expression of the form $f(\ldots)$ will not unify with a nor b. This enables us to cut down the search space, and only consider 6 of the 69 unifiers of s' and t'. Eventually, we obtain 4 unifiers of s and t.

Exercises

1. Is it true that $\{x \mapsto y\} \leq \{x \mapsto a\}$ (i.e. is the substitution $\{x \mapsto y\}$ more general than the substitution $\{x \mapsto a\}$)? Motivate your answer.

2. Let t be any term and let σ, γ and δ be substitutions. Prove or disprove the following statements.

   (a) $\sigma(\gamma(t)) = \sigma(\delta(t)) \Rightarrow \gamma = \delta$
   (b) $\gamma(\sigma(t)) = \delta(\sigma(t)) \Rightarrow \gamma = \delta$
   (c) $\gamma = \delta \Rightarrow \sigma(\gamma(t)) = \sigma(\delta(t))$
   (d) $\gamma = \delta \Rightarrow \gamma(\sigma(t)) = \delta(\sigma(t))$
   (e) $\gamma = \delta \Rightarrow \gamma \sim_\emptyset \delta$
   (f) $\gamma \sim_\emptyset \delta \Rightarrow \gamma = \delta$

3. Let $f/2$ and $g/2$ be function symbols, a and b constants and x, y and z variables. Find an E-unifier for $f(y, f(g(a, y), g(z, b)))$ and $f(x, f(z, y))\}$, or prove that none exists, if

   (a) $E = \emptyset$
   (b) $E = \{f(x, y) = f(y, x)\}$
   (c) $E = \{f(f(x, y), z) = f(x, f(y, z))\}$
   (d) $E = \{f(x, y) = f(y, x),\ f(f(x, y), z) = f(x, f(y, z))\}$
   (e) $E = \{g(x, y) = x\}$

4. Prove theorem 6.8.

5. Consider the set E of axioms

$$\begin{aligned}&append(nil, x) = x\\ &append(cons(x, y), z) = cons(x, append(y, z))\end{aligned}$$

where *cons* and *append* are binary function symbols, and nil, 1 and 2 are constants. Use narrowing to compute all E-unifiers of $append(x, cons(2, nil))$ and $cons(1, cons(2, nil))$.

6. Let E be the equational theory

$$\begin{array}{rr} f(x) = g(x,x) & a = b \\ g(a,b) = c & g(b,b) = f(a) \end{array}$$

If we orient the equations left to right, we obtain the following TRS:

$$\begin{array}{rr} f(x) \rightarrow g(x,x) & a \rightarrow b \\ g(a,b) \rightarrow c & g(b,b) \rightarrow f(a) \end{array}$$

Use this TRS to E-unify $f(a)$ and c by:

(a) narrowing

(b) basic narrowing

Explain your results.

7. Prove theorem 6.13.

8. One of the steps in Stickel's AC-unification algorithm is to remove subterms that occur in both terms to be unified. Let s and t be two terms which both has u as a subterm. Prove that the set of AC-unifiers of s and t is preserved when we remove an occurrence of u from s, and an occurrence of u from t.

9. Find the remaining three AC-unifiers in the example of section 6.4.

Chapter 7

Algebraic Specifications

Chapter 5 discusses set of equations as specifications of equational theories. The validity techniques presented therein concern *all* models of an equational theory defined in that way. This chapter shows that a set of equations can be considered a specification of an algebra, which is a particular model of the equational theory. The original motivation for such an approach came from the attempts to formalize the notion of abstract data type.

The chapter gives only a few basic concepts. A more systematic introduction to the field of algebraic specifications can be found in e.g. [22] or in [70]. Several specification and programming languages are based on these concepts.

7.1 Many-sorted algebras

Sets of data stored in the computer are often organized in some structures, like arrays, lists, trees etc. To manipulate such structures certain operations specific for a given structure are used. The data structures together with these specific operations are often called *data types*. It is natural to assume that all data items of a data type are generated by the operations of the data type from a number of basic items. Thus, one has a number of data domains and a number of operations, each of which takes arguments in some of the domains and produces values in one of the domains. Such an object can be thought of as an algebra. Such a kind of algebra is called *many-sorted* or *heterogeneous* algebra.

The starting point for defining a heterogeneous algebra is to choose names which will later be used for referring to the data domains. Formally one can define an index set S, whose elements are called *sorts*. [1] The domains are not defined yet, hence they are not referred to in the formal definition.

[1]The traditional algebras where the set of sorts has only one element constitute a subclass of heterogeneous algebras and are called *homogeneous algebras*.

For example one can take $S = \{\underline{nat}, \underline{bool}, \underline{nlist}\}$ as a set of sorts.

Each operation to be defined takes arguments from a finite number of data domains and produces values in a data domain. Before defining such an operation we choose its name and indicate the names of the domains involved.

Definition 7.1
An S-*sorted signature* Σ, is a family of sets $\Sigma_{w,s}$ where $s \in S$ and $w \in S^*$. An element $f \in \Sigma_{w,s}$ is called a *function symbol* or an *operation symbol* of *rank* (w, s), where w is called the *arity* of f and s is called its *sort*. □

For the set S considered above one can for example consider the following S-signature:

Example 7.2
$\Sigma = \{\Sigma_{\epsilon,\underline{nat}}, \Sigma_{\underline{nat},\underline{nat}}, \Sigma_{\epsilon,\underline{bool}}, \Sigma_{\underline{bool}\ \underline{bool},\underline{bool}}, \Sigma_{\underline{nlist}\ \underline{nlist},\underline{bool}}, \Sigma_{\epsilon,\underline{nlist}}, \Sigma_{\underline{nat}\ \underline{nlist},\underline{nlist}}\}$
where

$\Sigma_{\epsilon,\underline{nat}} = \{0\}$,
$\Sigma_{\underline{nat},\underline{nat}} = \{s\}$,
$\Sigma_{\epsilon,\underline{bool}} = \{true, false\}$
$\Sigma_{\underline{bool}\ \underline{bool},\underline{bool}} = \{\vee, \Rightarrow\}$,
$\Sigma_{\underline{nlist}\ \underline{nlist},\underline{bool}} = \{\#_{\underline{nlist}}\}$
$\Sigma_{\epsilon,\underline{nlist}} = \{null\}$ and
$\Sigma_{\underline{nat}\ \underline{nlist},\underline{nlist}} = \{cons\}$.

□

Now, to define an algebra with a given signature one has to specify the domains denoted by the sorts and the operations on these domains denoted by the respective operation symbols of the signature.

Definition 7.3
Let Σ be an S-sorted signature. Then a Σ-algebra, A, consists of a set A_s for each $s \in S$ (called the *carrier of A of sort s*) and a function $f_A : A_{s_1} \times \dots \times A_{s_n} \to A_s$ for each $f \in \Sigma_{w,s}$ where $w = s_1 s_2 \dots s_n$ (called the *operation of A named by f*). For $f \in \Sigma_{\epsilon,s}$ we have $f_A \in A_s$. That is, $\Sigma_{\epsilon,s}$ is the set of constant symbols of sort s. □

Comparing the concept of Σ-algebra with the notion of F-algebra (Definition 5.3) we see that the former covers the latter as a special case when the signature is one-sorted.

Example 7.4
Let Σ be the S-sorted signature of example 7.2. The sets $A_{\underline{nat}} = \{0, 1, 2, 3, \dots\}$, $A_{\underline{bool}} = \{0, 1\}$ and $A_{\underline{nlist}} = \{[], [0, 15], [3, 75, 1, 4], \dots\}$, are carriers in the Σ-algebra where for instance $\Rightarrow: A_{\underline{bool}} \times A_{\underline{bool}} \to A_{\underline{bool}}$ is defined by

$\Rightarrow(0,0) = 1$
$\Rightarrow(0,1) = 1$
$\Rightarrow(1,0) = 0$
$\Rightarrow(1,1) = 1$

□

Clearly, for a given signature Σ there are different Σ-algebras.

7.2 Specifications

It is often stressed that a data type should be a black box in the sense that the user should not know how it is implemented. Taking the algebraic view one would expect a data type to be a many-sorted algebra. It should be clear that the user has to specify the signature Σ of the data type to be defined. The data type is then expected to be a Σ-algebra. The question is how to define this algebra. The approach presented below is based on many-sorted equational specifications. Existence of a specific model of any such a specification is then discussed. In the algebraic specification approach this model is defined to be the algebra specified by a given set of equations.

We now extend the notions of term and equation discussed for one-sorted signatures to the many-sorted case.

Let Σ be an S-sorted signature and let V be a family of denumerable sets, V_s, of variables of sort $s \in S$. The sets $Ter_s(\Sigma, V)$ of terms of sorts $s \in S$ are defined by mutual recursion as follows:

- if $x \in V_s$ then $x \in Ter_s(\Sigma, V)$
- if $f \in \Sigma_{s_1 \dots s_n, s}$ and $t_i \in Ter_{s_i}(\Sigma, V)$ then $f(t_1, \dots, t_n) \in Ter_s(\Sigma, V)$

For example, for the signature of Example 7.2:

$s(0)$ is a term of sort <u>*nat*</u>,
$cons(x, y)$ is a term of sort <u>*nlist*</u> provided that x and y are variables of the sorts, respectively, <u>*nat*</u> and <u>*nlist*</u>,
etc.

Now a Σ-*equation* can be defined as a pair of terms of the same sort and we can give a formal (syntactic) definition of a specification.

Definition 7.5 A *specification* is a triple $\langle S, \Sigma, E\rangle$, where Σ is an S-sorted signature and E is a set of Σ-equations. □

For convenience we often write $\langle\Sigma, E\rangle$. Notice, that in the one-sorted case such specifications are not syntactically different from the equational specifications discussed in Chapter 5. However, the latter have been used for specifying equational theories while the specifications introduced here are to be used for defining specific algebras. Thus, for any given specification $\langle\Sigma, E\rangle$ we will choose a particular Σ-algebra as *the* algebra defined by the specification.

7.3 An intuition of a data specification

This section motivates the choice of the algebra to be linked to a given specification.

Intuitively we put two requirements: (1) it should be possible to represent each data item by a ground term in the signature; thus in our algebra there is no "junk" inexpressible in our language, (2) two ground terms represent the same data item iff this follows from the equational specification.

Formally (2) should be defined by referring to many-sorted extension of the concept of equational theory discussed in Chapter 5: groud terms t_1 and t_2 represent the same data item iff the equation $t_1 = t_2$ belongs to the theory.

The requirement (2) is a kind of "minimality" assumption: the algebra we want to define should not satisfy more equations than those following from the specification. A simple example should clarify this point.

Example 7.6 Consider the one-sorted specification:

$0 + x = x$
$s(x) + y = s(x + y)$

The data items will be represented by ground terms, such as $0, s(0), 0 + 0, etc.$ Due to the equations every ground term represents the same data item as a term of the form $s^n(0)$ for some $n = 0, 1, \dots$. On the other hand, according to the assumption, the items represented by the terms $s^k(0)$ and $s^n(0)$ for $k \neq n$ should be different. It can be checked that the relation "to represent the same item" is the equivalence relation on ground terms. The set of all terms representing the same item as a given term t will be denoted $[t]$.

Thus, the data items can be linked to the equivalence classes of the ground terms defined by the equations. Define the operations s and $+$ on this domain as follows:

$s([t]) = [s(t)]$
$[t] + [u] = [t + u]$

Clearly, the algebra A defined in this way is a model of the theory defined by the specification. It is, however, one specific model. Consider for example the algebra B with the carrier $0, 1$ and with the operations

s: successor *modulo* 2, i.e. $s(0) = 1, s(1) = 0$
$+$: addition *modulo* 2, i.e. $0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1 = 0$

The algebra B is also a model of the theory defined by the example specification. However, the equation $s(1) = 0$ is not a logical consequence of the theory. Therefore B does not satisfy the minimality assumption.

Yet another model of the theory is the algebra N having natural numbers as its carrier, the constant 0 of the signature interpreted as zero, s interpreted as the successor, and $+$ as the addition. It can be checked that N satisfies the minimality assumptions, and is in this respect similar to A. As discussed below, N and A are *isomorphic*. □

7.4 The initial algebra semantics of specifications

We now survey results which show that a construction similar to that of Example 7.6 can be done for any specification. The objective is to define a unique (up to isomorphism) algebra which can be considered as the algebra defined by a specification.

It turns out that the "minimality" assumption discussed above can be expressed in terms of homomorphisms. In the case of many-sorted algebras the usual notion of homomorphism has to be extended as follows:

Definition 7.7
If A and B are both Σ-algebras, a Σ-*homomorphism* $h : A \to B$ is a family of functions $\langle h_s : A_s \to B_s \rangle_{s \in S}$ that preserve the operations, i.e., that satisfy

1. If $f \in \Sigma_{\epsilon,s}$ then $h_s(f_A) = f_B$
2. If $f \in \Sigma_{s_1 \ldots s_n,s}$ and $\langle a_1, \ldots, a_n \rangle \in A_{s_1} \times \cdots \times A_{s_n}$, then $h_s(f_A(a_1, \ldots a_n)) = f_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$.

□

A homomorphism h such that every h_s is a one-one mapping from A_s onto B_s is called an *isomorphism*. Intuitively, a homomorphism mimics the operations of A in B. If it is an isomorphism, the results of operations in A can be reconstructed from the results obtained in B, and vice-versa. We have then an exact "simulation" of each algebra by the other. It is not the case if the homomorphism is not an isomorphism.

Example 7.8 Consider the (one-sorted) algebras A, B and N of Example 7.6, and the mappings defined as follows:

$h^{AN}(s^n(0)) = n$ for every natural number n,
$h^{NB}(n) = 0$ if n is even and 1 if n is odd.

It is easy to check that h^{AN} is an isomorphism from A to N, while h^{NB} is a homomorphism from N to B. The mapping h^{NB} is not an isomorphism, since it is not one-one: e.g. all odd numbers in the carrier of N are mapped to 1.

□

How is the concept of homomorphism related to our minimality assumption? As illustrated by Example 7.6, some algebras may satisfy more equations than those included in the equational theory defined by the specification. To study the question we first have to define the notion of equation satisfied by an algebra. For this we will use the notion of homomorphism.

We first introduce some auxiliary notions. Let V be a set of variables. We denote by $T_\Sigma(V)$ the Σ-algebra defined as follows:

- for every sort s the corresponding carrier is the set of terms $Ter_s(\Sigma, V)$,
- for every $f \in \Sigma_{s_1 \ldots s_n, s}$ the operation f in the algebra $T_\Sigma(V)$ applied to the arguments $t_i \in Ter_{s_i}(\Sigma, V)$ results in the term $f(t_1, \ldots, t_n) \in Ter_s(\Sigma, V)$

In particular, the Σ-algebra $T_\Sigma(\emptyset)$ has the sets $Ter_s(\Sigma, \emptyset)$ of ground (many-sorted) terms as its carriers.

For any S-sorted Σ-algebra, A, and S-indexed family V of sets of variables, an *assignment* $\sigma : V \to A$ of values of sort s in A to variables of sort s in V, is a family of mappings $\langle \sigma_s : V_s \to A_s \rangle_{s \in S}$.

So, an assignment valuates variables according to their sorts in the carriers of some Σ-algebra A. This valuation provides a unique interpretation of terms in A, since every operation symbol in a term determines an operation of A. For example, an assignment $\sigma : \{x \mapsto 2, y \mapsto 3\}$ of variables to natural numbers maps the term $x + y$ into the natural number $\sigma(x + y) \equiv 5$.

The following formal definition identifies the class of Σ-algebras characterized by a given set of Σ-equations.

Definition 7.9
A Σ-algebra A *satisfies* a Σ-equation $s = t$ iff $\sigma(s) = \sigma(t)$ for all assignments $\sigma : V \to A$. If E is a set of Σ-equations, then A satisfies E iff A satisfies every equation in E. An algebra satisfying E is called a (Σ, E)-algebra. □

The set of (Σ, E)-algebras is of course the set of models of E, also called the *variety* of E. This is because the definition of what a Σ-algebra is, together with the definition of an assignment, is essentially the many-sorted version of the definition of an interpretation discussed in Section 5.2. If an algebra A satisfies an equation $s = t$, we write this as $A \models s = t$. This is extended to sets of equations so that if $A \models E$, then A is a (Σ, E)-algebra. Also, let C be a class of algebras. If $C \models s = t$ we mean that every algebra in C satisfies $s = t$. If C is the set of all (Σ, E)-algebras, we write $E \models s = t$ instead of $C \models s = t$.

If there exists a Σ-homomorphism from A to B then B satisfies all equations satisfied by A, and may be more. For example the equation $s(s(x)) = x$ is not satisfied in the algebra N of Example 7.6 but it is satisfied by the algebra B. Such a situation cannot happen if A and B are isomorphic.

Thus, our minimality requirement means that if the algebra to be chosen is a homomorphic image of another one then they should be isomorphic. Consequently, the following notion specialized to the class of (Σ, E)-algebras would be very relevant for our objectives.

Definition 7.10 A Σ-algebra I is called *initial* in a class $\mathcal{C}$ of Σ-algebras iff for every Σ-algebra A in $\mathcal{C}$ there exists a unique Σ-homomorphism from I to A. □

A direct consequence of this definition is that different initial algebras of $\mathcal{C}$ must be isomorphic. The question is whether for every $\langle \Sigma, E \rangle$ there exists an algebra initial in the class of (Σ, E)-algebras. The answer is positive and we now show how it can be constructed.

The following result can be obtained from the definitions:

Theorem 7.11
Let Σ be an S-sorted signature. Then $T_\Sigma(\emptyset)$ is initial in the class of *all* Σ-algebras.

Indeed, the unique homomorphism into an algebra A is defined by the mapping that assigns every constant of the signature into the corresponding value in a carrier of A.

This result is, however, not that important for our purposes, since we are interested not in all Σ-algebras but in the class described by an equational specification. Thus we ask, what is the initial algebra $\Im_E$ in the class of all (Σ, E)-algebras?

The following formal notions and a related result give a formal foundation for the construction used in the introductory example.

Definition 7.12
A Σ-*congruence* $\cong$ on a Σ-algebra A, is a family $\langle \cong_s \rangle_{s \in S}$ of equivalence relations $\cong_s$ on A_s for $s \in S$, such that if $f \in \Sigma_{s_1 \ldots s_n, s}$, if $a_i, a'_i \in A_{s_i}$ and if $a_i \cong_{s_i} a'_i$, then $f_A(a_1, \ldots, a_n) \cong_s f_A(a'_1, \ldots, a'_n)$. □

If A is a Σ-algebra, then $A_s / \cong_s$ denotes the set of $\cong_s$-equivalence classes of A_s. For $a \in A_s$ we denote by $[a]_s$ the class containing a. We define the Σ-algebra $A/\cong$ by

1. if $f \in \Sigma_{\epsilon, s}$ then $f_{A/\cong} = [f_A]$
2. if $f \in \Sigma_{s_1 \ldots s_n, s}$ and $[a_i]_{s_i} \in A_{s_i} / \cong_{s_i}$, then $f_{A/\cong}([a_1]_{s_1}, \ldots, [a_n]_{s_n}) = [f_A(a_1, \ldots, a_n)]_s$

Theorem 7.13 If A is a Σ-algebra and $\cong$ is a Σ-congruence on A, then $A/\cong$ as defined above is a Σ-algebra, called the *quotient* of A by $\cong$.

The set E of Σ-equations generates a congruence relation $\cong_E$ on any Σ-algebra A by

- if $s = t \in E$ where $s, t \in Ter_s(\Sigma, V)$, then $\sigma(s) \cong_E \sigma(t)$ for all assignments $\sigma : V \to A$
- if $f \in \Sigma_{s_1 \ldots s_n, s}$ and $a_i \cong_E a'_i$ where $a_i, a'_i \in A_{s_i}$ then $f_A(a_1, \ldots, a_n) \cong_E f_A(a'_1, \ldots, a'_n)$
- if $a \cong_E a'$ then $a' \cong_E a$
- if $a \cong_E a'$ and $a' \cong_E a''$ then $a \cong_E a''$
- $a \cong_{E,s} a$ for all $s \in S$ and $a \in A_s$
- nothing else is in $\cong_E$

Example 7.14 For the algebra T_Σ the equations of Example 7.6 define the equivalence classes $\{[0], [s[0], [s(s(0))], \ldots\}$. Notice first that every ground instance of a rule includes the same number of s's in the lhs and in the rhs. It follows by the definition of $\cong_E$ that two terms including different number of s's cannot be equivalent. It remains to show that any term t including $+$'s is equivalent to a term not including $+$. We can prove this by induction on the number of occurrences of $+$ in t. □

The following result generalizes theorem 7.11.

Theorem 7.15
Let E be a set of Σ-equations and let $\cong_E$ be the congruence relation on $T_\Sigma(\emptyset)$ generated by E. Then $\Im_E = T_\Sigma(\emptyset)/\cong_E$ is initial in the class of all algebras satisfying E.

Thus, the semantics of a data type specification is given by the initial algebra in the variety of E.

Discussion

The initial algebra characterizes the set of ground equations $s = t$ (equation of ground terms s, t) such that $E \models s = t$. Intuitively this is what we are interested in when computing with the abstract datatype. As we have seen, term rewriting systems give convenient means to answer questions of the form $E \models s = t$. Consider again the equations

$$x + 0 = x$$
$$x + s(y) = s(x + y)$$

Applying these axioms as rewrite rules one can reduce both sides of a given ground equation to normal forms and in that way to decide the validity of the equation. However, this technique does not allow to prove that $E \models 0 + t = t$ for *every* ground term t (which is actually the case) since it is not true that $E \models 0 + x = x$. (Cf. Exercise 3 of Chapter 5.) On the other hand, it *is* true that $\Im_E \models 0 + x = x$, since in general $\Im_E \models s = t$ iff $E \models \sigma(s) = \sigma(t)$ for every ground substitution σ. This leads to two conclusions: (1) In the context of data types, we are primarily interested in the properties of the initial model. (2) Ordinary equational reasoning is generally not strong enough for proving such properties. In our example, $\Im_E \models 0 + x = x$ can be proved by induction on the structure of terms of the sort $\underline{nat}$. Thus verification of data type specifications calls for more powerful proof methods than equational reasoning based on rewriting. In particular, inductive proofs are typically used for that purpose. Recall that the equational logic is complete (Theorem 5.6). That is, any equation which is a logical consequence of a given specification can be proved by the inference rules of the equational logic. In contrast to that, there are fundamental reasons for incompleteness of inductive reasoning. For discussion and literature see e.g. [56].

Bibliography

[1] W. Ackermann. Zum Hilbertschen Aufbau der reelen Zahlen. *Math. Annalen*, 99, pp. 118–133, 1928.

[2] H.P. Barendregt. *The Lambda calculus: its syntax and semantics*, 2nd edition. North-Holland, 1984.

[3] G. Birkhoff. On the structure of abstract algebras. In *Proceedings of the Cambridge Philosophical Society*, pp. 433–454, 1935.

[4] G. Boole. *The mathematical analysis of logic.* Macmillan 1847.

[5] G. Boolos and R. Jeffrey. *Computability and Logic.* Cambridge University Press, 1980.

[6] B. Buchberger and R. Loos. Algebraic simplification. In *Computer algebra: Symbolic and algebraic computation*, pp. 11–42, Springer-Verlag, 1982.

[7] B. Caviness and R. Fateman. Simplification of radical expressions. *SYMSAC*, pp. 329–338, 1976.

[8] C.L. Chang and R. Lee. *Symbolic logic and mechanical theorem proving.* Academic Press, 1973.

[9] N. Chomsky. Three models for the description of language. In *IRE transactions on information theory*, 2:3, pp. 113–124, 1956.

[10] A. Church. *The calculi of Lambda-conversion.* Annals of mathematical studies 6, Princeton University Press, 1941.

[11] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society* 39, pp. 472–482, 1936.

[12] P. Cohn. *Universal algebra.* Harper and Row, 1965.

[13] H. Curry and R. Feys. *Combinatory logic*, Vol. I. North-Holland, 1958.

[14] H. Curry, J.R. Hindley and J.P. Seldin. *Combinatory logic*, Vol II. North-Holland, 1972.

[15] D. van Dalen, H.C. Doets and H. de Swart. *Sets: naïve, axiomatic and applied.* Pergamon, 1978.

[16] M. Dauchet. Simulation of Turing machines by a regular rewrite rule. *Theoretical computer science* 103, pp. 409–420, 1992.

[17] D. De Groot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations.* Prentice Hall 1986.

[18] P. Deransart and J. Małuszynski. *A Grammatical View of Logic Programming.* The MIT Press 1993.

[19] N. Dershowitz. Orderings for term rewriting systems. *Theoretical computer science* 17, pp. 279–301, 1982.

[20] N. Dershowitz. Termination of rewriting. *Journal of symbolic computation* (3), pp. 69–115, 1987.

[21] N. Dershowitz. Completion and its applications. In *Resolution of equation in algebraic structures, Vol II: Rewriting techniques*, pp. 31–86. Academic Press, 1989.

[22] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications I* EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag, 1985.

[23] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical computer science* 43, pp. 189–200, 1986.

[24] M. Fay. First-order unification in equational theories. In *Proc. of the 4th international workshop on automated deduction*, pp. 161–167, 1979.

[25] M. Hanus. The integration of functions into logic programming. *Journal of logic programming* 19/20, pp. 583–628, 1994.

[26] M. Hanus. A unified computation model for functional and logic programming. *Proc POPL'97*, pp. 80–93, ACM 1997.

[27] J. Herbrand. *Recherches de la théorie de la demonstration.* Thèse d'état, Université de Paris, 1930. (In Goldfarb (ed.) *Logical writings*, Harvard University Press, 1971.)

[28] R. Hindley and J. Seldin. *Introduction to combinators and λ-calculus.* Cambridge University Press, 1986.

[29] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*, 2nd edition. Addison-Wesley, 1979.

[30] G. Huet. *Resolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse d'état, Université de Paris VII, 1976.

[31] G. Huet. An algorithm to generate the basis of solutions to homogeneous Diophantine equations. *Information Processing Letters*, 7, 3, 1978.

[32] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM* 27 (4), pp. 797–821, 1980.

[33] G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences* 23, pp. 11–21, 1981.

[34] G. Huet and D. Lankford. *On the uniform halting problem for term rewriting systems*. Research report 283, INRIA Rocquencourt, France, 1978.

[35] J.-M. Hullot, Canonical forms and unification. In *Proceedings of the 5th conference on automated deduction*, Lecture notes in computer science **87**, pp. 318-334, 1980.

[36] G. Kahn. Natural Semantics. In *Proc. of STACS 1987*, volume 247 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[37] J. W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, Vol II, Oxford University Press.

[38] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, vol. **21**. no. 1, March 1989.

[39] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), pp. 127–145, June 1968.

[40] D. Knuth and P. Bendix. Simple word problems in universal algebras. *Computational problems in abstract algebra*, pp. 263–297, Pergamon Press, 1970.

[41] J. B. Kruskal. Well-quasi ordering, the Tree theorem, and Vazsonyi's conjecture. In *Transactions of the American mathematical society* 95, pp. 210–225, 1960.

[42] Peter Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[43] D. Lankford. *Canonical algebraic simplification in computational logic*. Memo ATP-25, Automatic Theorem Proving Project, University of Texas at Austin, 1975.

[44] D. Lankford. *Some approaches to equality for equational logic: A survey and assessment*. Memo ATP-36, Automatic Theorem Proving Project, University of Texas at Austin, 1977.

[45] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited, In J. Minker, ed., *Foundations of deductive databases and logic programming*, pp. 587-626, Morgan Kaufmann, Los Altos, 1988.

[46] Makanin, G.S. The problem of solvability of equations on a free semigroup. *Math. USSR Sbornik 32*, 2, 1977. English translation, AMS, 1979.

[47] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Proceedings of the 3rd Hawaii International Conference on System Science*, pp. 789–792, 1968.

[48] K. Marriott and P.J. Stuckey. *Programming with Constraints*, The MIT Press, 1999.

[49] A. Martelli and U. Montanari. Theorem proving with structure sharing and efficient unification. In *Proc. of the international joint conference on AI (IJCAI)*, 1977.

[50] A. Martelli, U. Montanari, An efficient unification algorithm. *ACM TOPLAS*, vol. **4**, no. 2, April 1982.

[51] Y. Matiyasevitch. Diophantine representation of recursively enumerable predicates. In *Second Scandinavian logic symposium*, North Holland, 1970.

[52] Y. Métivier. About the rewriting systems produced by the Knuth-Bendix completion algorithm. *Information Processing Letters* **16** pp 31-34, 1983.

[53] A. Middeldorp and E. Hamoen. *Counterexamples to completeness results for basic narrowing*. CWI report CS-R9154. Extended abstract in *Proc. of 3rd international conference on Algebraic and logic programming*, pp. 244–258. Springer-Verlag, 1992.

[54] M. Newman. On theories with a combinatorial definition of "equivalence". *Ann. Math.* 43, pp. 223–243, 1942.

[55] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*, Second Edition, Wiley and Sons 1995.¸´

[56] H. Olsén. *Collage Induction: Proving Properties of Logic Programs by Program Synthesis.* Linköping studies in Science and Technology, Lic. Thesis No.464, Linköping 1994.

[57] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences* 16, pp. 158–167, 1978.

[58] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13, pp. 231–278.

[59] D. A. Plaisted. The undecidability of self-embedding for term rewriting systems. *Information processing letters* 20, pp. 61–64, 1985.

[60] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[61] J. Robinson. A machine-oriented logic based on the resolution principle. In *Journal of the ACM* 12, pp. 23–41, 1965.

[62] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[63] B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM* 20(1), pp. 160–187, 1973.

[64] M. Schönfinkel. Über die Bausteine der matematischen Logik. *Math. Ann.* 92, pp. 305–316, 1924.

[65] J. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7, 3, 1989.

[66] J. Slagle. Automated theorem proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM* 21:4, pp. 622–642, 1974.

[67] M. Stickel. A complete unification algorithm for associative-commutative functions. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*, Tblisi, pp. 71–76, 1975. Extended version in *Journal of the ACM*, 28,3, pp. 423–434, 1981.

[68] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2:42, pp. 230–265. Corrected in *ibid.* 43, pp. 544–546, 1936.

[69] E. Villemonte de la Clergerie and B. Lang. LPDA: Another look at tabulation in logic programming. *Proceedings of the Eleventh International Conference on Logic Programming*, pp. 470–486, The MIT Press 1994.

[70] M. Wirsing. Algebraic Specifications. In: *Handbook of Theoretical Computer Science.* J. van Leeuwen (ed.), Elsevier, 1990.