

Chapter 10

Many-Sorted First-Order Logic

10.1 Introduction

There are situations in which it is desirable to express properties of structures of different types (or sorts). For instance, this is the case if one is interested in axiomatizing data structures found in computer science (integers, reals, characters, stacks, queues, lists, etc). By adding to the formalism of first-order logic the notion of *type* (also called *sort*), one obtains a flexible and convenient logic called *many-sorted first-order logic*, which enjoys the same properties as first-order logic.

In this chapter, we shall define and give the basic properties of many-sorted first-order logic. It turns out that the semantics of first-order logic can be given conveniently using the notion of a many-sorted algebra defined in Section 2.5, given in the appendix. Hence, the reader is advised to review the appendix before reading this chapter.

At the end of this chapter, we give an algorithm for deciding whether a quantifier-free formula is valid, using the method of congruence closure due to Kozen (Kozen, 1976, 1977).

Due to the lack of space, we can only give a brief presentation of many-sorted first-order logic, and most proofs are left as exercises. Fortunately, they are all simple variations of proofs given in the first-order case.

10.2 Syntax

First, we define alphabets for many-sorted first-order languages.

10.2.1 Many-Sorted First-Order Languages

In contrast to standard first-order languages, in many-sorted first-order languages, the arguments of function and predicate symbols may have different types (or sorts), and constant and function symbols also have some type (or sort). Technically, this means that a many-sorted alphabet is a many-sorted ranked alphabet as defined in Subsection 2.5.1.

Definition 10.2.1 The *alphabet of a many-sorted first-order language* consists of the following sets of symbols:

A countable set $S \cup \{bool\}$ of *sorts* (or *types*) containing the special sort $bool$, and such that S is nonempty and does not contain $bool$.

Logical connectives: \wedge (and), \vee (or), \supset (implication), \equiv (equivalence), all of rank $(bool, bool, bool)$, \neg (not) of rank $(bool, bool)$, \perp (of rank $(e, bool)$);

Quantifiers: For every sort $s \in S$, \forall_s (for all), \exists_s (there exists), each of rank $(bool, bool)$;

For every sort s in S , the *equality symbol* \doteq_s , of rank $(ss, bool)$.

Variables: For every sort $s \in S$, a countably infinite set $\mathbf{V}_s = \{x_0, x_1, x_2, \dots\}$, each variable x_i being of rank (e, s) . The family of sets \mathbf{V}_s is denoted by \mathbf{V} .

Auxiliary symbols: “(” and “)”.

An $(S \cup \{bool\})$ -ranked alphabet \mathbf{L} of *nonlogical symbols* consisting of:

(i) *Function symbols:* A (countable, possibly empty) set \mathbf{FS} of symbols f_0, f_1, \dots , and a *rank function* $r : \mathbf{FS} \rightarrow S^+ \times S$, assigning a pair $r(f) = (u, s)$ called *rank* to every function symbol f . The string u is called the *arity* of f , and the symbol s the *sort* (or *type*) of f .

(ii) *Constants:* For every sort $s \in S$, a (countable, possibly empty) set \mathbf{CS}_s of symbols c_0, c_1, \dots , each of rank (e, s) . The family of sets \mathbf{CS}_s is denoted by \mathbf{CS} .

(iii) *Predicate symbols:* A (countable, possibly empty) set \mathbf{PS} of symbols P_0, P_1, \dots , and a *rank function* $r : \mathbf{PS} \rightarrow S^* \times \{bool\}$, assigning a pair $r(P) = (u, bool)$ called *rank* to each predicate symbol P . The string u is called the *arity* of P . If $u = e$, P is a propositional letter.

It is assumed that the sets \mathbf{V}_s , \mathbf{FS} , \mathbf{CS}_s , and \mathbf{PS} are disjoint for all $s \in S$. We will refer to a many-sorted first-order language with set of nonlogical symbols \mathbf{L} as the language \mathbf{L} . Many-sorted first-order languages obtained

by omitting the equality symbol are referred to as *many-sorted first-order languages without equality*.

Observe that a standard (one sorted) first-order language corresponds to the special case of a many-sorted first-order language for which the set S of sorts contains a single element.

We now give inductive definitions for the sets of many-sorted terms and formulae.

Definition 10.2.2 Given a many-sorted first-order language \mathbf{L} , let Γ be the union of the sets \mathbf{V} , \mathbf{CS} , \mathbf{FS} , \mathbf{PS} , and $\{\perp\}$, and let Γ_s be the subset of Γ^+ consisting of all strings whose leftmost symbol is of sort $s \in S \cup \{bool\}$.

For every function symbol f of rank $(u_1 \dots u_n, s)$, let C_f be the function $C_f : \Gamma_{u_1} \times \dots \times \Gamma_{u_n} \rightarrow \Gamma_s$ such that, for all strings t_1, \dots, t_n , with each t_i of sort u_i ,

$$C_f(t_1, \dots, t_n) = ft_1 \dots t_n.$$

For every predicate symbol P of arity $u_1 \dots u_n$, let C_P be the function $C_P : \Gamma_{u_1} \times \dots \times \Gamma_{u_n} \rightarrow \Gamma_{bool}$ such that, for all strings t_1, \dots, t_n , each t_i of sort u_i ,

$$C_P(t_1, \dots, t_n) = Pt_1 \dots t_n.$$

Also, let C_{\equiv}^s be the function $C_{\equiv}^s : (\Gamma_s)^2 \rightarrow \Gamma_{bool}$ (of rank $(ss, bool)$) such that, for all strings t_1, t_2 of sort s ,

$$C_{\equiv}^s(t_1, t_2) = \dot{=}^s t_1 t_2.$$

Finally, the functions C_{\wedge} , C_{\vee} , C_{\supset} , C_{\equiv} , C_{\neg} are defined as in definition 3.2.2, with C_{\wedge} , C_{\vee} , C_{\supset} , C_{\equiv} of rank $(bool.bool, bool)$, C_{\neg} of rank $(bool, bool)$, and the functions $A_i^s, E_i^s : \Gamma_{bool} \rightarrow \Gamma_{bool}$ (of rank $(bool, bool)$) are defined such that, for any string A in Γ_{bool} ,

$$A_i^s(A) = \forall_s x_i A, \text{ and } E_i^s(A) = \exists_s x_i A.$$

The $(S \cup \{bool\})$ -indexed family $(\Gamma_s)_{s \in (S \cup \{bool\})}$ is made into a many-sorted algebra also denoted by Γ as follows:

Each carrier of sort $s \in (S \cup \{bool\})$ is the set of strings Γ_s ;

Each constant c of sort s in \mathbf{CS} is interpreted as the string c ;

Each predicate symbol P of rank $(e, bool)$ in \mathbf{PS} (propositional symbol) is interpreted as the string P ;

The constant \perp is interpreted as the string \perp .

The operations are the functions $C_f, C_P, C_{\wedge}, C_{\vee}, C_{\supset}, C_{\equiv}, C_{\neg}, C_{\equiv}^s, A_i^s$ and E_i^s .

From Subsection 2.5.5, we know that there is a least subalgebra $T(\mathbf{L}, \mathbf{V})$ containing the $(S \cup \{bool\})$ -indexed family of sets \mathbf{V} (with the component of sort $bool$ being empty).

The set of terms $TERM_{\mathbf{L}}^s$ of \mathbf{L} -terms of sort s (for short, terms of sort s) is the carrier of sort s of $T(\mathbf{L}, \mathbf{V})$, and the set $FORM_{\mathbf{L}}$ of \mathbf{L} -formulae (for short, formulae) is the carrier of sort *bool* of $T(\mathbf{L}, \mathbf{V})$.

A less formal way of stating definition 10.2.2 is the following. Terms and atomic formulae are defined as follows:

- (i) Every constant and every variable of sort s is a term of sort s .
- (ii) If t_1, \dots, t_n are terms, each t_i of sort u_i , and f is a function symbol of rank $(u_1 \dots u_n, s)$, then $ft_1 \dots t_n$ is a term of sort s .
- (iii) Every propositional letter is an atomic formula, and so is \perp .
- (iv) If t_1, \dots, t_n are terms, each t_i of sort u_i , and P is a predicate symbol of arity $u_1 \dots u_n$, then $Pt_1 \dots t_n$ is an atomic formula; If t_1 and t_2 are terms of sort s , then $\dot{=}^s t_1 t_2$ is an atomic formula;

Formulae are defined as follows:

- (i) Every atomic formula is a formula.
- (ii) For any two formulae A and B , $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$ and $\neg A$ are also formulae.
- (iii) For any variable x_i of sort s and any formula A , $\forall_s x_i A$ and $\exists_s x_i A$ are also formulae.

EXAMPLE 10.2.1

Let \mathbf{L} be following many-sorted first-order language for stacks, where $S = \{stack, integer\}$, $\mathbf{CS}_{integer} = \{0\}$, $\mathbf{CS}_{stack} = \{\Lambda\}$, $\mathbf{FS} = \{Succ, +, *, push, pop, top\}$, and $\mathbf{PS} = \{<\}$.

The rank functions are given by:

$$r(Succ) = (integer, integer);$$

$$r(+)=r(*)=(integer.integer, integer);$$

$$r(push)=(stack.integer, stack);$$

$$r(pop)=(stack, stack);$$

$$r(top)=(stack, integer);$$

$$r(<)=(integer.integer, bool).$$

Then, the following are terms:

$$Succ\ 0$$

$$top\ push\ \Lambda\ Succ\ 0$$

EXAMPLE 10.2.2

Using the first-order language of example 10.2.1, the following are formulae:

$< 0 \text{ Succ } 0$

$\forall_{integer} x \forall_{stack} y \dot{=}_{stack} \text{ pop push } y \ x \ y .$

10.2.2 Free Generation of Terms and Formulae

As in Subsections 5.2.2 and 5.2.3, it is possible to show that terms and formulae are freely generated. This is left as an exercise for the reader.

Theorem 10.2.1 The many-sorted algebra $T(\mathbf{L}, \mathbf{V})$ is freely generated by the family \mathbf{V} . \square

As a consequence, the family $TERM_{\mathbf{L}}$ of many-sorted terms is freely generated by the set of constants and variables as atoms and the functions C_f , and the set of \mathbf{L} -formulae is freely generated by the atomic formulae as atoms and the functions C_X (X a logical connective), A_i^s and E_i^s .

Remarks:

(1) Instead of defining terms and atomic formulae in prefix notation, one can define them as follows (using parentheses):

The second clause of definition 10.2.2 is changed to: For every function symbol f of rank $(u_1 \dots u_n, s)$ and any terms t_1, \dots, t_n , with each t_i of sort u_i , $f(t_1, \dots, t_n)$ is a term of sort s . Also, atomic formulae are defined as follows:

For every predicate symbol P of arity $u_1 \dots u_n$, for any terms t_1, \dots, t_n , with each t_i of sort u_i , $P(t_1, \dots, t_n)$ is an atomic formula; For any terms t_1 and t_2 of sort s , $(t_1 \dot{=}_s t_2)$ is an atomic formula.

We will also use the notation $\forall x : sA$ and $\exists x : sA$, instead of $\forall_s xA$ and $\exists_s xA$.

One can still show that the terms and formulae are freely generated. In the sequel, we shall use either notation. For simplicity, we shall also frequently use $=$ instead of $\dot{=}_s$ and omit parentheses whenever possible, using the conventions adopted in Chapter 5.

(2) The many-sorted algebra $T(\mathbf{L}, \mathbf{V})$ is free on the set of variables \mathbf{V} (as defined in Section 2.5), and is isomorphic to the tree algebra $T_{\mathbf{L}}(\mathbf{V})$ (in $T_{\mathbf{L}}(\mathbf{V})$, the term $A_i^s(A)$ is used instead of $\forall x_i : sA$, and $E_i^s(A)$ instead of $\exists x_i : sA$).

10.2.3 Free and Bound Variables, Substitutions

The definitions of *free and bound variables*, *substitution*, and *term free for a variable in a formula* given in Chapter 5 extend immediately to the many-sorted case. The only difference is that in a substitution $s[t/x]$ or $A[t/x]$ of a term t for a variable x , the sort of the term t must be equal to the sort of the variable x .

PROBLEMS

10.2.1. Prove theorem 10.2.1.

10.2.2. Examine carefully how the definitions of Section 5.2 generalize to the many-sorted case (free and bound variables, substitutions, etc.).

10.2.3. Give a context-free grammar describing many-sorted terms and many-sorted formulae, as in problem 5.2.8.

10.2.4. Generalize the results of problems 5.2.3 to 5.2.7 to the many-sorted case.

10.3 Semantics of Many-Sorted First-Order Languages

First, we need to define many-sorted first-order structures.

10.3.1 Many-Sorted First-Order Structures

Given a many-sorted first-order language \mathbf{L} , the semantics of formulae is defined as in Section 5.3, but using the concept of a many-sorted algebra rather than the concept of a (one-sorted) structure.

Recall from definition 10.2.1 that the nonlogical part \mathbf{L} of a many-sorted first-order language is an $(S \cup \{bool\})$ -sorted ranked alphabet.

Definition 10.3.1 Given a many-sorted first-order language \mathbf{L} , a *many-sorted \mathbf{L} -structure* \mathbf{M} , (for short, a *structure*) is a many-sorted \mathbf{L} -algebra as defined in Section 2.5.2, such that the carrier of sort *bool* is the set $BOOL = \{\mathbf{T}, \mathbf{F}\}$.

Recall that the carrier of sort s is nonempty and is denoted by M_s . Every function symbol f of rank $(u_1 \dots u_n, s)$ is interpreted as a function $f_{\mathbf{M}} : M^u \rightarrow M_s$, with $M^u = M^{u_1} \times \dots \times M^{u_n}$, $u = u_1 \dots u_n$, each constant c of sort s is interpreted as an element $c_{\mathbf{M}}$ in M_s , and each predicate P of arity $u_1 \dots u_n$ is interpreted as a function $P_{\mathbf{M}} : M^u \rightarrow BOOL$.

10.3.2 Semantics of Formulae

We now wish to define the semantics of formulae by generalizing the definition given in Section 5.3.

Definition 10.3.2 Given a first-order many-sorted language \mathbf{L} and an \mathbf{L} -structure \mathbf{M} , an *assignment of sort s* is any function $v_s : \mathbf{V}_s \rightarrow M_s$ from the set of variables \mathbf{V}_s to the domain M_s . The family of all such functions is denoted as $[\mathbf{V}_s \rightarrow M_s]$. An assignment v is any S -indexed family of assignments of sort s . The set of all assignments is denoted by $[\mathbf{V} \rightarrow M]$.

As in Chapter 5, the meaning of a formula is defined recursively using theorem 2.5.1. In order to apply theorem 2.5.1, it is necessary to define a many-sorted algebra \mathcal{M} . For this, the next two definitions are needed.

Definition 10.3.3 For all $i \geq 0$, for every sort s , the functions $(A_i^s)_{\mathcal{M}}$ and $(E_i^s)_{\mathcal{M}}$ (of rank $(bool, bool)$) from $[[\mathbf{V} \rightarrow M] \rightarrow BOOL]$ to $[[\mathbf{V} \rightarrow M] \rightarrow BOOL]$ are defined as follows: For every function f in $[[\mathbf{V} \rightarrow M] \rightarrow BOOL]$, $(A_i^s)_{\mathcal{M}}(f)$ is the function such that: For every assignment v in $[\mathbf{V} \rightarrow M]$,

$$(A_i^s)_{\mathcal{M}}(f)(v) = \mathbf{F} \quad \text{iff} \quad f(v[x_i := a]) = \mathbf{F} \text{ for some } a \in M_s;$$

The function $(E_i^s)_{\mathcal{M}}(f)$ is the function such that: For every assignment v in $[\mathbf{V} \rightarrow M]$,

$$(E_i^s)_{\mathcal{M}}(f)(v) = \mathbf{T} \quad \text{iff} \quad f(v[x_i := a]) = \mathbf{T} \text{ for some } a \in M_s.$$

Note that $(A_i^s)_{\mathcal{M}}(f)(v) = \mathbf{T}$ iff the function $g_v : M_s \rightarrow BOOL$ such that $g_v(a) = f(v[x_i := a])$ for all $a \in M_s$, is the constant function whose value is \mathbf{T} , and that $(E_i^s)_{\mathcal{M}}(f)(v) = \mathbf{F}$ iff the function $g_v : M_s \rightarrow BOOL$ defined above is the constant function whose value is \mathbf{F} .

Definition 10.3.4 Given any \mathbf{L} -structure \mathbf{M} , the many-sorted algebra \mathcal{M} is defined as follows. For each sort $s \in S$, the carrier \mathcal{M}_s is the set $[[\mathbf{V} \rightarrow M] \rightarrow M_s]$, and the carrier \mathcal{M}_{bool} is the set $[[\mathbf{V} \rightarrow M] \rightarrow BOOL]$. The functions $\wedge_{\mathcal{M}}$, $\vee_{\mathcal{M}}$, $\supset_{\mathcal{M}}$ and $\equiv_{\mathcal{M}}$ of rank $(bool, bool, bool)$ from $\mathcal{M}_{bool} \times \mathcal{M}_{bool}$ to \mathcal{M}_{bool} , and the function $\neg_{\mathcal{M}}$ of rank $(bool, bool)$ from \mathcal{M}_{bool} to \mathcal{M}_{bool} are defined as follows: For every two functions f, g in \mathcal{M}_{bool} , for every assignment v in $[\mathbf{V} \rightarrow M]$,

$$\wedge_{\mathcal{M}}(f, g)(v) = H_{\wedge}(f(v), g(v));$$

$$\vee_{\mathcal{M}}(f, g)(v) = H_{\vee}(f(v), g(v));$$

$$\supset_{\mathcal{M}}(f, g)(v) = H_{\supset}(f(v), g(v));$$

$$\equiv_{\mathcal{M}}(f, g)(v) = H_{\equiv}(f(v), g(v));$$

$$\neg_{\mathcal{M}}(f)(v) = H_{\neg}(f(v)).$$

For every function symbol f of rank (u, s) , every predicate symbol P of rank $(u, bool)$, and every constant symbol c , the functions $f_{\mathcal{M}} : \mathcal{M}^u \rightarrow \mathcal{M}_s$, $P_{\mathcal{M}} : \mathcal{M}^u \rightarrow \mathcal{M}_{bool}$, and the function $c_{\mathcal{M}} \in \mathcal{M}_s$ are defined as follows. For any $(g_1, \dots, g_n) \in \mathcal{M}^u$, ($n = |u|$), for any assignment v in $[\mathbf{V} \rightarrow M]$,

$$f_{\mathcal{M}}(g_1, \dots, g_n)(v) = f_{\mathbf{M}}(g_1(v), \dots, g_n(v));$$

$$P_{\mathcal{M}}(g_1, \dots, g_n)(v) = P_{\mathbf{M}}(g_1(v), \dots, g_n(v));$$

$$c_{\mathcal{M}}(v) = c_{\mathbf{M}}.$$

For every variable x of sort s , we also define $x_{\mathcal{M}} \in \mathcal{M}_s$ as the function such that for every assignment v , $x_{\mathcal{M}}(v) = v_s(x)$. Let $\varphi : \mathbf{V} \rightarrow \mathcal{M}$ be the function defined such that, $\varphi(x) = x_{\mathcal{M}}$. Since $T(\mathbf{L}, \mathbf{V})$ is freely generated by \mathbf{V} , by theorem 2.5.1, there is a unique homomorphism $\widehat{\varphi} : T(\mathbf{L}, \mathbf{V}) \rightarrow \mathcal{M}$ extending φ . We define the meaning $t_{\mathbf{M}}$ of a term t as $\widehat{\varphi}(t)$, and the meaning $A_{\mathbf{M}}$ of a formula A as $\widehat{\varphi}(A)$. The explicit recursive definitions follow.

Definition 10.3.5 Given a many-sorted \mathbf{L} -structure \mathbf{M} and an assignment $v : \mathbf{V} \rightarrow M$, the function $t_{\mathbf{M}} : [\mathbf{V} \rightarrow M] \rightarrow M_s$ defined by a term t of sort s is the function such that for every assignments v in $[\mathbf{V} \rightarrow M]$, the value $t_{\mathbf{M}}[v]$ is defined recursively as follows:

- (i) $x_{\mathbf{M}}[v] = v_s(x)$, for a variable x of sort s ;
- (ii) $c_{\mathbf{M}}[v] = c_{\mathbf{M}}$, for a constant c ;
- (iii) $(ft_1 \dots t_n)_{\mathbf{M}}[v] = f_{\mathbf{M}}((t_1)_{\mathbf{M}}[v], \dots, (t_n)_{\mathbf{M}}[v])$.

The recursive definition of the function $A_{\mathbf{M}} : [\mathbf{V} \rightarrow M] \rightarrow \text{BOOL}$ is now given.

Definition 10.3.6 The function $A_{\mathbf{M}} : [\mathbf{V} \rightarrow M] \rightarrow \text{BOOL}$ is defined recursively by the following clauses:

(1) For atomic formulae: $A_{\mathbf{M}}$ is the function such that, for every assignment v ,

- $(Pt_1 \dots t_n)_{\mathbf{M}}[v] = P_{\mathbf{M}}((t_1)_{\mathbf{M}}[v], \dots, (t_n)_{\mathbf{M}}[v])$;
- $(\doteq_s t_1 t_2)_{\mathbf{M}}[v] = \text{if } (t_1)_{\mathbf{M}}[v] = (t_2)_{\mathbf{M}}[v] \text{ then } \mathbf{T} \text{ else } \mathbf{F}$;
- $(\perp)_{\mathbf{M}}[v] = \mathbf{F}$;

(2) For nonatomic formulae:

$(A * B)_{\mathbf{M}} = *_{\mathcal{M}}(A_{\mathbf{M}}, B_{\mathbf{M}})$, where $*$ is in $\{\wedge, \vee, \supset, \equiv\}$, and $*_{\mathcal{M}}$ is the corresponding function defined in definition 10.3.4;

- $(\neg A)_{\mathbf{M}} = \neg_{\mathcal{M}}(A_{\mathbf{M}})$;
- $(\forall x_i : sA)_{\mathbf{M}} = (A_i^s)_{\mathcal{M}}(A_{\mathbf{M}})$;
- $(\exists x_i : sA)_{\mathbf{M}} = (E_i^s)_{\mathcal{M}}(A_{\mathbf{M}})$.

Note that by definitions 10.3.3, 10.3.4, 10.3.5, and 10.3.6, for every assignment v ,

- $(\forall x_i : sA)_{\mathbf{M}}[v] = \mathbf{T}$ iff $A_{\mathbf{M}}[v[x_i := m]] = \mathbf{T}$, for all $m \in M_s$, and
- $(\exists x_i : sA)_{\mathbf{M}}[v] = \mathbf{T}$ iff $A_{\mathbf{M}}[v[x_i := m]] = \mathbf{T}$, for some $m \in M_s$.

The notions of *satisfaction*, *validity*, and *model* are defined as in Subsection 5.3.3. As in Subsection 5.3.4, it is also possible to define the semantics of formulae using modified formulae obtained by substitution.

10.3.3 An Alternate Semantics

The following result analogous to lemma 5.3.2 can be shown.

Lemma 10.3.1 For any formula B , for any assignment v in $[\mathbf{V} \rightarrow M]$, and any variable x_i of sort s , the following hold:

- (1) $(\forall x_i : sB)_{\mathbf{M}}[v] = \mathbf{T}$ iff $(B[\mathbf{m}/x_i])_{\mathbf{M}}[v] = \mathbf{T}$ for all $m \in M_s$;
- (2) $(\exists x_i : sB)_{\mathbf{M}}[v] = \mathbf{T}$ iff $(B[\mathbf{m}/x_i])_{\mathbf{M}}[v] = \mathbf{T}$ for some $m \in M_s$. \square

In view of lemma 10.3.1, the recursive clauses of the definition of satisfaction can also be stated more informally as follows:

$$\begin{aligned}
\mathbf{M} \models (\neg A)[v] &\text{ iff } \textit{not } \mathbf{M} \models A[v], \\
\mathbf{M} \models (A \wedge B)[v] &\text{ iff } \mathbf{M} \models A[v] \text{ and } \mathbf{M} \models B[v], \\
\mathbf{M} \models (A \vee B)[v] &\text{ iff } \mathbf{M} \models A[v] \text{ or } \mathbf{M} \models B[v], \\
\mathbf{M} \models (A \supset B)[v] &\text{ iff } \textit{not } \mathbf{M} \models A[v] \text{ or } \mathbf{M} \models B[v], \\
\mathbf{M} \models (A \equiv B)[v] &\text{ iff } (\mathbf{M} \models A[v] \text{ iff } \mathbf{M} \models B[v]), \\
\mathbf{M} \models (\forall x_i : sA)[v] &\text{ iff } \mathbf{M} \models (A[\mathbf{a}/x_i])[v] \text{ for every } a \in M_s, \\
\mathbf{M} \models (\exists x_i : sA)[v] &\text{ iff } \mathbf{M} \models (A[\mathbf{a}/x_i])[v] \text{ for some } a \in M_s.
\end{aligned}$$

It is also easy to show that the semantics of a formula A only depends on the set $FV(A)$ of variables free in A .

10.3.4 Semantics and Free Variables

The following lemma holds.

Lemma 10.3.2 Given a formula A with set of free variables $\{y_1, \dots, y_m\}$, for any two assignments s_1, s_2 such that $s_1(y_i) = s_2(y_i)$, for $1 \leq i \leq m$, $A_{\mathbf{M}}[s_1] = A_{\mathbf{M}}[s_2]$. \square

10.3.5 Subformulae and Rectified Formulae

Subformulae and *rectified formulae* are defined as in Subsection 5.3.6, and lemma 5.3.4 can be generalized easily. Similarly, the results of the rest of Section 5.3 can be generalized to many-sorted logic. The details are left as exercises.

PROBLEMS

10.3.1. Prove lemma 10.3.1.

10.3.2. Prove lemma 10.3.2.

10.3.3. Generalize lemma 5.3.4 to the many-sorted case.

10.3.4. Examine the generalization of the other results of Section 5.3 to the many-sorted case.

10.4 Proof Theory of Many-Sorted Languages

The system G defined in Section 5.4 is extended to the many-sorted case as follows.

10.4.1 Gentzen System G for Many-Sorted Languages Without Equality

We first consider the case of a many-sorted first-order language \mathbf{L} *without equality*.

Definition 10.4.1 (Gentzen system G for languages without equality) The symbols Γ, Δ, Λ will be used to denote arbitrary sequences of formulae and A, B to denote formulae. The rules of the sequent calculus G are the following:

$$\frac{\Gamma, A, B, \Delta \rightarrow \Lambda}{\Gamma, A \wedge B, \Delta \rightarrow \Lambda} (\wedge : left) \quad \frac{\Gamma \rightarrow \Delta, A, \Lambda \quad \Gamma \rightarrow \Delta, B, \Lambda}{\Gamma \rightarrow \Delta, A \wedge B, \Lambda} (\wedge : right)$$

$$\frac{\Gamma, A, \Delta \rightarrow \Lambda \quad \Gamma, B, \Delta \rightarrow \Lambda}{\Gamma, A \vee B, \Delta \rightarrow \Lambda} (\vee : left) \quad \frac{\Gamma \rightarrow \Delta, A, B, \Lambda}{\Gamma \rightarrow \Delta, A \vee B, \Lambda} (\vee : right)$$

$$\frac{\Gamma, \Delta \rightarrow A, \Lambda \quad B, \Gamma, \Delta \rightarrow \Lambda}{\Gamma, A \supset B, \Delta \rightarrow \Lambda} (\supset : left) \quad \frac{A, \Gamma \rightarrow B, \Delta, \Lambda}{\Gamma \rightarrow \Delta, A \supset B, \Lambda} (\supset : right)$$

$$\frac{\Gamma, \Delta \rightarrow A, \Lambda}{\Gamma, \neg A, \Delta \rightarrow \Lambda} (\neg : left) \quad \frac{A, \Gamma \rightarrow \Delta, \Lambda}{\Gamma \rightarrow \Delta, \neg A, \Lambda} (\neg : right)$$

In the quantifier rules below, x is any variable of sort s and y is any variable of sort s free for x in A and not free in A , unless $y = x$ ($y \notin FV(A) - \{x\}$). The term t is any term of sort s free for x in A .

$$\frac{\Gamma, A[t/x], \forall x : sA, \Delta \rightarrow \Lambda}{\Gamma, \forall x : sA, \Delta \rightarrow \Lambda} (\forall : left) \quad \frac{\Gamma \rightarrow \Delta, A[y/x], \Lambda}{\Gamma \rightarrow \Delta, \forall x : sA, \Lambda} (\forall : right)$$

$$\frac{\Gamma, A[y/x], \Delta \rightarrow \Lambda}{\Gamma, \exists x : sA, \Delta \rightarrow \Lambda} (\exists : left) \quad \frac{\Gamma \rightarrow \Delta, A[t/x], \exists x : sA, \Lambda}{\Gamma \rightarrow \Delta, \exists x : sA, \Lambda} (\exists : right)$$

Note that in both the $(\forall : right)$ -rule and the $(\exists : left)$ -rule, the variable y does *not* occur free in the lower sequent. In these rules, the variable y is called the *eigenvariable* of the inference. The condition that the eigenvariable does not occur free in the conclusion of the rule is called the *eigenvariable condition*. The formula $\forall x : sA$ (or $\exists x : sA$) is called the *principal formula* of the inference, and the formula $A[t/x]$ (or $A[y/x]$) the *side formula* of the inference.

The *axioms* of G are all sequents $\Gamma \rightarrow \Delta$ such that Γ and Δ contain a common formula.

10.4.2 Deduction Trees for the System G

Deduction trees and *proof trees* are defined as in Subsection 5.4.2.

10.4.3 Soundness of the System G

The soundness of the system G is obtained easily from the proofs given in Subsection 5.4.3.

Lemma 10.4.1 (Soundness of G, many-sorted case) Every sequent provable in G is valid. \square

10.4.4 Completeness of G

It is also possible to prove the completeness of the system G by adapting the definitions and proofs given in Sections 5.4 and 5.5 to the many-sorted case. For this it is necessary to modify the definition of a Hintikka set so that it applies to a many-sorted algebra. We only state the result, leaving the proof as a sequence of problems.

Theorem 10.4.1 (Gödel's extended completeness theorem for G) A sequent (even infinite) is valid iff it is G-provable. \square

10.5 Many-Sorted First-Order Logic With Equality

The equality rules for many-sorted languages with equality are defined as follows.

10.5.1 Gentzen System $G_{=}$ for Languages With Equality

Let $G_{=}$ be the Gentzen system obtained from the Gentzen system G (defined in definition 10.4.1) by adding the following rules.

Definition 10.5.1 (Equality rules for $G_{=}$) Let Γ, Δ, Λ denote arbitrary sequences of formulae (possibly empty) and let $t, s_1, \dots, s_n, t_1, \dots, t_n$ denote arbitrary **L**-terms. For every sort s , for every term t of sort s :

$$\frac{\Gamma, t \doteq_s t \rightarrow \Delta}{\Gamma \rightarrow \Delta}$$

For each function symbol f of rank $(u_1 \dots u_n, s)$ and any terms $s_1, \dots, s_n, t_1, \dots, t_n$ such that s_i and t_i are of sort u_i :

$$\frac{\Gamma, (s_1 \doteq_{u_1} t_1) \wedge \dots \wedge (s_n \doteq_{u_n} t_n) \supset (f s_1 \dots s_n \doteq_s f t_1 \dots t_n) \rightarrow \Delta}{\Gamma \rightarrow \Delta}$$

For each predicate symbol P (including $\dot{=}_s$) of arity $u_1 \dots u_n$ and any terms $s_1, \dots, s_n, t_1, \dots, t_n$ such that s_i and t_i are of sort u_i :

$$\frac{\Gamma, ((s_1 \dot{=}_{u_1} t_1) \wedge \dots \wedge (s_n \dot{=}_{u_n} t_n) \wedge P s_1 \dots s_n) \supset P t_1 \dots t_n \rightarrow \Delta}{\Gamma \rightarrow \Delta}$$

10.5.2 Soundness of the System $G_=$

The following lemma is easily shown.

Lemma 10.5.1 If a sequent is $G_=$ -provable then it is valid. \square

10.5.3 Completeness of the System $G_=$

It is not difficult to adapt the proofs of Section 5.6 to obtain the following completeness theorem.

Theorem 10.5.1 (Gödel's extended completeness theorem for $G_=$) Let \mathbf{L} be a many-sorted first-order language with equality. A sequent (even infinite) is valid iff it is $G_=$ -provable. \square

10.5.4 Reduction of Many-Sorted Logic to One-Sorted Logic

Although many-sorted first-order logic is very convenient, it is not an essential extension of standard one-sorted first-order logic, in the sense that there is a translation of many-sorted logic into one-sorted logic. Such a translation is described in Enderton, 1972, and the reader is referred to it for details. The essential idea to convert a many-sorted language \mathbf{L} into a one-sorted language \mathbf{L}' is to add domain predicate symbols D_s , one for each sort, and to modify quantified formulae recursively as follows:

Every formula A of the form $\forall x : s B$ (or $\exists x : s B$) is converted to the formula $A' = \forall x (D_s(x) \supset B')$ ($\exists x : s B$ is converted to $\exists x (D_s(x) \wedge B')$), where B' is the result of converting B .

We also define the set MS to be the set of all one-sorted formulae of the form:

- (1) $\exists x D_s(x)$, for every sort s , and
- (2) $\forall x_1 \dots \forall x_n (D_{s_1}(x_1) \wedge \dots \wedge D_{s_n}(x_n) \supset D_s(f(x_1, \dots, x_n)))$, for every function symbol f of rank $(s_1 \dots s_n, s)$.

Then, the following lemma can be shown.

Lemma 10.5.2 Given a many-sorted first-order language \mathbf{L} , for every set T of many-sorted formulae in \mathbf{L} and for every many-sorted formula A in \mathbf{L} ,

$$T \vdash A \quad \text{iff} \\ T' \cup MS \vdash A', \quad \text{in the translated one-sorted language } \mathbf{L}',$$

where T' is the set of formulae in T translated into one-sorted formulae, and A' is the one-sorted translation of A . \square

Lemma 10.5.2 can be used to transfer results from one-sorted logic to many-sorted logic. In particular, the compactness theorem, model existence theorem, and Löwenheim-Skolem theorem hold in many-sorted first-order logic.

PROBLEMS

10.4.1. Prove lemma 10.4.1.

10.4.2. Define many-sorted Hintikka sets for languages without equality, and prove lemma 5.4.5 for the many-sorted case.

10.4.3. Prove the completeness theorem (theorem 10.4.1) for many-sorted logic without equality.

10.5.1. Prove lemma 10.5.1.

10.5.2. Generalize the other theorems of Section 5.5 to the many-sorted case (compactness, model existence, Löwenheim-Skolem).

10.5.3. Define many-sorted Hintikka sets for languages with equality, and prove lemma 5.6.1 for the many-sorted case.

10.5.3. Prove the completeness theorem (theorem 10.5.1) for many-sorted logic with equality.

10.5.4. Generalize the other theorems of Section 5.6 to the many-sorted case (compactness, model existence, Löwenheim-Skolem).

10.5.5. Prove lemma 10.5.2.

10.6 Decision Procedures Based on Congruence Closure

In this section, we show that there is an algorithm for deciding whether a quantifier-free formula without predicate symbols in a many-sorted language with equality is valid, using a method due to Kozen (Kozen, 1976, 1977) and Nelson and Oppen (Nelson and Oppen, 1980). Then, we show how this algorithm can easily be extended to deal with arbitrary quantifier-free formulae in a many-sorted language with equality.

10.6.1 Decision Procedure for Quantifier-free Formulae Without Predicate Symbols

First, we state the following lemma whose proof is left as an exercise.

Lemma 10.6.1 Given any quantifier-free formula A in a many-sorted first-order language \mathbf{L} , a formula B in disjunctive normal form such that $A \equiv B$ is valid can be constructed. \square

Using lemma 10.6.1, observe that a quantifier-free formula A is valid iff the disjunctive normal form B of $\neg A$ is unsatisfiable. But a formula $B = C_1 \vee \dots \vee C_n$ in disjunctive normal form is unsatisfiable iff every disjunct C_i is unsatisfiable. Hence, in order to have an algorithm for deciding validity of quantifier-free formulae, it is enough to have an algorithm for deciding whether a conjunction of literals is unsatisfiable.

If the language \mathbf{L} has no equality symbols, the problem is trivial since a conjunction of literals is unsatisfiable iff it contains some atomic formula $Pt_1 \dots t_n$ and its negation. Otherwise, we follow a method due to Kozen (Kozen, 1976, 1977) and Nelson and Oppen (Nelson and Oppen, 1980). First, we assume that the language \mathbf{L} does not have predicate symbols. Then, every conjunct C consists of equations and of negations of equations:

$$t_1 \doteq_{s_1} u_1 \wedge \dots \wedge t_m \doteq_{s_m} u_m \wedge \neg r_1 \doteq_{s'_1} v_1 \wedge \dots \wedge \neg r_n \doteq_{s'_n} v_n$$

We give a method inspired from Kozen and Nelson and Oppen for deciding whether a conjunction C as above is unsatisfiable. For this, we define the concept of a congruence on a graph.

10.6.2 Congruence Closure on a Graph

First, we define the notion of a labeled graph. We are considering graphs in which for every node u , the set of immediate successors is ordered. Also, every node is labeled with a function symbol from a many-sorted ranked alphabet, and the labeling satisfies a condition similar to the condition imposed on many-sorted terms.

Definition 10.6.1 A *finite labeled graph* G is a quadruple $(V, \Sigma, \Lambda, \delta)$, where:

V is an S -indexed family of finite sets V_s of *nodes* (or *vertices*);

Σ is an S -sorted ranked alphabet;

$\Lambda : V \rightarrow \Sigma$ is a *labeling function* assigning a symbol $\Lambda(v)$ in Σ to every node v in V ;

$$\delta : \{(v, i) \mid v \in V, 1 \leq i \leq n, r(\Lambda(v)) = (u_1 \dots u_n, s)\} \rightarrow V,$$

is a function called the *successor function*.

The functions Λ and δ also satisfy the following properties: For every node v in V_s , the rank of the symbol $\Lambda(v)$ labeling v is of the form $(u_1 \dots u_n, s)$, and for every node $\delta(v, i)$, the sort of the symbol $\Lambda(\delta(v, i))$ labeling $\delta(v, i)$ is equal to u_i .

For every node v , $\delta(v, i)$ is called the i -th successor of v , and is also denoted as $v[i]$. Note that for a node v such that $r(\Lambda(v)) = (e, s)$, δ is not defined. Such a node is called a *terminal node*, or *leaf*. Given a node u , the set P_u of predecessors of u is the set $\{v \in V \mid \delta(v, i) = u, \text{ for some } i\}$. A node u such that $P_u = \emptyset$ is called an *initial node*, or *root*. A pair (v, i) as in the definition of δ is also called an *edge*.

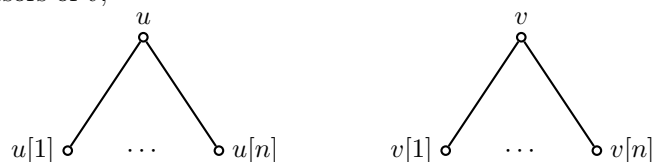
Next, we define a certain kind of equivalence relation on a graph called a congruence. A congruence is an equivalence relation closed under a certain form of implication.

Definition 10.6.2 Given a (finite) graph $G = (V, \Sigma, \Lambda, \delta)$, an S -indexed family R of relations R_s over V_s is a G -congruence (for short, a congruence) iff:

- (1) Each R_s is an equivalence relation;
- (2) For every pair (u, v) of nodes in V^2 , if $\Lambda(u) = \Lambda(v)$ and $r(\Lambda(u)) = (e, s)$ then $u R_s v$, else if $\Lambda(u) = \Lambda(v)$, $r(\Lambda(u)) = (s_1 \dots s_n, s)$, and for every i , $1 \leq i \leq n$, $u[i] R_{s_i} v[i]$, then $u R_s v$.

In particular, note that any two terminal nodes labeled with the same symbol of arity e are congruent.

Graphically, if u and v are two nodes labeled with the same symbol f of rank $(s_1 \dots s_n, s)$, if $u[1], \dots, u[n]$ are the successors of u and $v[1], \dots, v[n]$ are the successors of v ,



if $u[i]$ and $v[i]$ are equivalent for all i , $1 \leq i \leq n$, then u and v are also equivalent. Hence, we have a kind of backward closure.

We will prove shortly that given any finite graph and S -indexed family R of relations on G , there is a smallest congruence on G containing R , called the *congruence closure* of R . First, we show how the congruence closure concept can be used to give an algorithm for deciding unsatisfiability.

10.6.3 The Graph Associated With a Conjunction

The key to the algorithm for deciding unsatisfiability is the computation of a certain congruence over a finite graph induced by C and defined below.

Definition 10.6.3 Given a conjunction C of the form

$$t_1 \doteq_{s_1} u_1 \wedge \dots \wedge t_m \doteq_{s_m} u_m \wedge \neg r_1 \doteq_{s'_1} v_1 \wedge \dots \wedge \neg r_n \doteq_{s'_n} v_n,$$

let $TERM(C)$ be the set of all subterms of terms occurring in the conjunction C , including the terms t_i, u_i, r_j, v_j themselves. Let $S(C)$ be the set of sorts of all terms in $TERM(C)$. For every sort s in $S(C)$, let $TERM(C)_s$ be the set of all terms of sort s in $TERM(C)$. Note that by definition, each set $TERM(C)_s$ is nonempty. Let Σ be the $S(C)$ -ranked alphabet consisting of all constants, function symbols, and variables occurring in $TERM(C)$. The graph $G(C) = (TERM(C), \Sigma, \Lambda, \delta)$ is defined as follows:

For every node t in $TERM(C)$, if t is either a variable or a constant then $\Lambda(t) = t$, else t is of the form $fy_1\dots y_k$ and $\Lambda(t) = f$;

For every node t in $TERM(C)$, if t is of the form $fy_1\dots y_k$, then for every i , $1 \leq i \leq k$, $\delta(t, i) = y_i$, else t is a constant or a variable and it is a terminal node of $G(C)$.

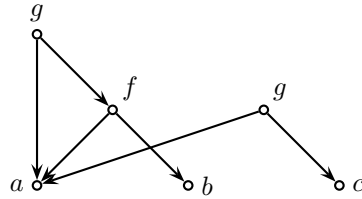
Finally, let $E = \{(t_1, u_1), \dots, (t_m, u_m)\}$ be the set of pairs of terms occurring in the positive (nonnegated) equations in the conjunction C .

EXAMPLE 10.6.1

Consider the alphabet in which $S = \{i, s\}$, $\Sigma = \{f, g, a, b, c\}$, with $r(f) = (is, s)$, $r(g) = (si, s)$, $r(a) = i$, $r(b) = r(c) = s$. Let C be the conjunction

$$f(a, b) \doteq c \wedge \neg g(f(a, b), a) \doteq g(c, a).$$

Then, $TERM(C)_i = \{a\}$, $TERM(C)_s = \{b, c, f(a, b), g(c, a), g(f(a, b), a)\}$, and $E = \{(f(a, b), c)\}$. The graph $G(C)$ is the following:



The key to the decision procedure is that the least congruence on $G(C)$ containing E exists, and that there is an algorithm for computing it. Indeed, assume that this least congruence $\xrightarrow{*}_E$ containing E (called the *congruence closure of E*) exists and has been computed. Then, we have the following result.

Lemma 10.6.2 Given a conjunction C of the form

$$t_1 \doteq_{s_1} u_1 \wedge \dots \wedge t_m \doteq_{s_m} u_m \wedge \neg r_1 \doteq_{s'_1} v_1 \wedge \dots \wedge \neg r_n \doteq_{s'_n} v_n$$

of equations and negations of equations, if $\xrightarrow{*}_E$ is the congruence closure on $G(C)$ of the relation $E = \{(t_1, u_1), \dots, (t_m, u_m)\}$, then

$$C \text{ is unsatisfiable iff for some } j, 1 \leq j \leq n, \quad r_j \xrightarrow{*}_E v_j.$$

Proof: Let $\mathcal{E} = \{t_1 \dot{=}_{s_1} u_1, \dots, t_m \dot{=}_{s_m} u_m\}$. First, we prove that the $S(C)$ -indexed family R of relations R_s on $TERM(C)$ defined such that

$$tR_s u \text{ iff } \mathcal{E} \models t \dot{=}_s u,$$

is a congruence on $G(C)$ containing E . It is obvious that

$$\mathcal{E} \models t_i \dot{=}_{s_i} u_i$$

for every (t_i, u_i) in E , $1 \leq i \leq m$, and so

$$t_i R_{s_i} u_i.$$

For every two subterms of the form $f y_1 \dots y_k$ and $f z_1 \dots z_k$ such that f is of rank $(w_1 \dots w_k, s)$, if for every i , $1 \leq i \leq k$,

$$\mathcal{E} \models y_i \dot{=}_{w_i} z_i$$

then by the definition of the semantics of equality,

$$\mathcal{E} \models f y_1 \dots y_k \dot{=}_s f z_1 \dots z_k.$$

Hence, R is a congruence on $G(C)$ containing E . Since $\xrightarrow{*}_E$ is the least congruence on $G(C)$ containing E ,

$$\text{if } r_j \xrightarrow{*}_E v_j, \text{ then } \mathcal{E} \models r_j \dot{=}_{s'_j} v_j.$$

But then, since any model satisfying C satisfies \mathcal{E} , both $r_j \dot{=}_{s'_j} v_j$ and $\neg r_j \dot{=}_{s'_j} v_j$ would be satisfied, a contradiction. We conclude that if for some j , $1 \leq j \leq n$,

$$r_j \xrightarrow{*}_E v_j,$$

then C is unsatisfiable. Conversely, assuming that there is no j , $1 \leq j \leq n$, such that

$$r_j \xrightarrow{*}_E v_j,$$

we shall construct a model \mathbf{M} of C . First, we make the $S(C)$ -indexed family $TERM(C)$ into a many-sorted Σ -algebra \mathbf{C} as follows:

For each sort s in $S(C)$, each constant or variable t of sort s is interpreted as the term t itself.

For every function symbol f in Σ of rank $(w_1 \dots w_k, s)$, for every k terms y_1, \dots, y_k in $TERM(C)$, each y_i being of sort w_i , $1 \leq i \leq k$,

$$f_{\mathbf{C}}(y_1, \dots, y_k) = \begin{cases} f y_1 \dots y_k & \text{if } f y_1 \dots y_k \in TERM(C)_s; \\ f z_1 \dots z_k & \text{if } f y_1 \dots y_k \notin TERM(C)_s \text{ and there are terms} \\ & z_1, \dots, z_k \text{ such that, } y_i \xrightarrow{*}_E z_i, \text{ and} \\ & f z_1 \dots z_k \in TERM(C)_s; \\ t_0 & \text{otherwise, where } t_0 \text{ is some arbitrary term} \\ & \text{chosen in } TERM(C)_s. \end{cases}$$

Next, we prove that $\xrightarrow{*}_E$ is a congruence on \mathbf{C} . Indeed, for every function symbol f in Σ of rank $(w_1 \dots w_k, s)$, for every k pairs of terms $(y_1, z_1), \dots, (y_k, z_k)$, with y_i, z_i of sort w_i , $1 \leq i \leq k$, if $y_i \xrightarrow{*}_E z_i$, then:

(1) If $f y_1 \dots y_k$ and $f z_1 \dots z_k$ are both in $TERM(C)$, then

$$f_{\mathbf{C}}(y_1, \dots, y_k) = f y_1 \dots y_k, \quad \text{and} \quad f_{\mathbf{C}}(z_1, \dots, z_k) = f z_1 \dots z_k,$$

and since $\xrightarrow{*}_E$ is a congruence on $G(C)$, we have $f y_1 \dots y_k \xrightarrow{*}_E f z_1 \dots z_k$. Hence, $f_{\mathbf{C}}(y_1, \dots, y_k) \xrightarrow{*}_E f_{\mathbf{C}}(z_1, \dots, z_k)$;

(2) $f y_1 \dots y_k \notin TERM(C)$, or $f z_1 \dots z_k \notin TERM(C)$, but there are some terms $z'_1, \dots, z'_k \in TERM(C)$, such that, $y_i \xrightarrow{*}_E z'_i$ and $f z'_1 \dots z'_k \in TERM(C)$. Since $y_i \xrightarrow{*}_E z_i$, there are also terms $z''_1, \dots, z''_k \in TERM(C)$ such that, $z_i \xrightarrow{*}_E z''_i$ and $f z''_1 \dots z''_k \in TERM(C)$. Then,

$$f_{\mathbf{C}}(y_1, \dots, y_k) = f z'_1 \dots z'_k \quad \text{and} \quad f_{\mathbf{C}}(z_1, \dots, z_k) = f z''_1 \dots z''_k.$$

Since $y_i \xrightarrow{*}_E z_i$, we have, $z'_i \xrightarrow{*}_E z''_i$, and so, $f z'_1 \dots z'_k \xrightarrow{*}_E f z''_1 \dots z''_k$, that is,

$$f_{\mathbf{C}}(y_1, \dots, y_k) \xrightarrow{*}_E f_{\mathbf{C}}(z_1, \dots, z_k).$$

(3) If neither $f y_1 \dots y_k$ nor $f z_1 \dots z_k$ is in $TERM(C)$ and (2) does not hold, then

$$f_{\mathbf{C}}(y_1, \dots, y_k) = f_{\mathbf{C}}(z_1, \dots, z_k) = t_0$$

for some chosen term t_0 in $TERM(C)$, and we conclude using the reflexivity of $\xrightarrow{*}_E$.

Let \mathbf{M} be the quotient of the algebra \mathbf{C} by the congruence $\xrightarrow{*}_E$, as defined in Subsection 2.5.8. Let v be the assignment defined as follows: For every variable x occurring in some term in $TERM(C)$, $v(x)$ is the congruence class $[x]$ of x .

We claim that for every term t in $TERM(C)$,

$$t_{\mathbf{M}}[v] = [t],$$

the congruence class of t . This is easily shown by induction and is left as an exercise. But then, C is satisfied by v in \mathbf{M} . Indeed, for every (t_i, u_i) in E ,

$$[t_i] = [u_i],$$

and so

$$\mathbf{M} \models (t_i \doteq_{s_i} u_i)[v],$$

and for every j , $1 \leq j \leq n$, since we have assumed that the congruence classes $[r_j]$ and $[v_j]$ are unequal,

$$\mathbf{M} \models (\neg r_j \doteq_{s'_j} v_j)[v].$$

□

The above lemma provides a decision procedure for quantifier-free formulae without predicate symbols. It only remains to prove that $\xleftrightarrow{*}_E$ exists and to give an algorithm for computing it. First, we give an example.

EXAMPLE 10.6.2

Recall the conjunction C of example 10.6.1:

$$f(a, b) \doteq c \wedge \neg g(f(a, b), a) \doteq g(c, a).$$

Then, $TERM(C)_i = \{a\}$, $TERM(C)_s = \{b, c, f(a, b), g(c, a), g(f(a, b), a)\}$, and it is not difficult to verify that the congruence closure of the relation $E = \{(f(a, b), c)\}$ has the following congruence classes: $\{a\}$, $\{b\}$, $\{f(a, b), c\}$ and $\{g(f(a, b), a), g(c, a)\}$. A possible candidate for the algebra \mathbf{C} is given by the following table:

| | f | g |
|----------------------|-----------|-----------------|
| (a, b) | $f(a, b)$ | |
| (b, a) | | b |
| (c, a) | | $g(c, a)$ |
| (a, c) | b | |
| $(f(a, b), a)$ | | $g(f(a, b), a)$ |
| $(a, f(a, b))$ | b | |
| $(a, g(c, a))$ | b | |
| $(g(c, a), a)$ | | b |
| $(a, g(f(a, b), a))$ | b | |
| $(g(f(a, b), a), a)$ | | b |

By lemma 10.6.2, C is unsatisfiable.

10.6.4 Existence of the Congruence Closure

We now prove that the congruence closure of a relation R on a finite graph G exists.

Lemma 10.6.3 (Existence of the congruence closure) Given a finite graph $G = (V, \Sigma, \Lambda, \delta)$ and a relation R on V , there is a smallest congruence \leftarrow^*_R on G containing R .

Proof: We define a sequence R^i of S -indexed families of relations inductively as follows: For every sort $s \in S$,

$$R_s^0 = R_s \cup \{(u, v) \in V^2 \mid \Lambda(u) = \Lambda(v), \text{ and } r(\Lambda(u)) = (e, s)\} \cup \{(u, u) \mid u \in V\};$$

For every sort $s \in S$,

$$\begin{aligned} R_s^{i+1} = & R_s^i \cup \{(v, u) \in V^2 \mid (u, v) \in R_s^i\} \\ & \cup \{(u, w) \in V^2 \mid \exists v \in V, (u, v) \in R_s^i \text{ and } (v, w) \in R_s^i\} \\ & \cup \{(u, v) \in V^2 \mid \Lambda(u) = \Lambda(v), r(\Lambda(u)) = (s_1 \dots s_n, s), \\ & \text{ and } u[j]R_{s_j}^i v[j], 1 \leq j \leq n\}. \end{aligned}$$

Let

$$\left(\leftarrow^*_R\right)_s = \bigcup R_s^i.$$

It is easily shown by induction that every congruence on G containing R contains every R^i , and that \leftarrow^*_R is a congruence on G . Hence, \leftarrow^*_R is the least congruence on G containing R . \square

Since the graph G is finite, there must exist some integer i such that $R^i = R^{i+1}$. Hence, the congruence closure of R is also computable. We shall give later a better algorithm due to Nelson and Oppen. But first, we show how the method of Subsection 10.6.3 can be used to give an algorithm for deciding the unsatisfiability of a conjunction of literals over a many-sorted language with equality (and function, constant, and predicate symbols).

10.6.5 Decision Procedure for Quantifier-free Formulae

The crucial observation that allows us to adapt the congruence closure method is the following:

For any atomic formula $Pt_1 \dots t_k$, if \top represents the logical constant always interpreted as \mathbf{T} , then $Pt_1 \dots t_k$ is logically equivalent to $(Pt_1 \dots t_k \equiv \top)$, in the sense that

$$Pt_1 \dots t_k \equiv (Pt_1 \dots t_k \equiv \top)$$

is valid.

But then, this means that \equiv behaves semantically exactly as the identity relation on **BOOL**. Hence, we can treat \equiv as the equality symbol \doteq_{bool} of sort *bool*, and interpret it as the identity on **BOOL**.

Hence, every conjunction C of literals is equivalent to a conjunction C' of equations and negations of equations, such that every atomic formula $Pt_1\dots t_k$ in C is replaced by the equation $Pt_1\dots t_k \equiv \top$, and every formula $\neg Pt_1\dots t_k$ in C is replaced by $\neg(Pt_1\dots t_k \equiv \top)$.

Then, as in Subsection 10.6.3, we can build the graph $G(C')$ associated with C' , treating \top as a constant of sort *bool*, and treating every predicate symbol P of arity $u_1\dots u_k$ as a function symbol of rank $(u_1\dots u_k, bool)$. We then have the following lemma generalizing lemma 10.6.2.

Lemma 10.6.4 Let C' be the conjunction obtained from a conjunction of literals obtained by changing literals of the form $Pt_1\dots t_k$ or $\neg Pt_1\dots t_k$ into equations as explained above. If C' is of the form

$$t_1 \doteq_{s_1} u_1 \wedge \dots \wedge t_m \doteq_{s_m} u_m \wedge \neg r_1 \doteq_{s'_1} v_1 \wedge \dots \wedge \neg r_n \doteq_{s'_n} v_n$$

and if $\xleftrightarrow{*}_E$ is the congruence closure on $G(C)$ of the relation $E = \{(t_1, u_1), \dots, (t_m, u_m)\}$, then

$$C \text{ is unsatisfiable iff for some } j, 1 \leq j \leq n, \quad r_j \xleftrightarrow{*}_E v_j.$$

Proof: We define $\mathcal{E} = \{t_1 \doteq_{s_1} u_1, \dots, t_m \doteq_{s_m} u_m\}$ and $TERM(C)$ as in Subsection 10.6.3, except that $S(C)$ also contains the sort *bool*, and we have a set of terms of sort *bool*. First, we prove that the $S(C)$ -indexed family R of relations R_s on $TERM(C)$ defined such that

$$tR_s u \text{ iff } \mathcal{E} \models t \doteq_s u,$$

is a congruence on $G(C)$ containing E . For function symbols of sort $s \neq bool$, the proof is identical to the proof of lemma 10.6.2. For every two subterms of the form $Py_1\dots y_k$ and $Pz_1\dots z_k$ such that P is of rank $(w_1\dots w_k, bool)$, if for every $i, 1 \leq i \leq k$,

$$\mathcal{E} \models y_i \doteq_{w_i} z_i,$$

then by the definition of the semantics of equality symbols,

$$\mathcal{E} \models Py_1\dots y_k \equiv Pz_1\dots z_k.$$

Hence, R is a congruence on $G(C)$ containing E . Now there are two cases.

(i) If $r_j \xleftrightarrow{*}_E v_j$ and the sort of r_j and v_j is not *bool*, we conclude as in the proof of lemma 10.6.2.

(ii) Otherwise, r_j is some atomic formula $Py_1\dots y_k$ and v_j is the constant \top . In that case,

$$\mathcal{E} \models r_j.$$

But then, since any model satisfying C satisfies \mathcal{E} , both r_j and $\neg r_j$ would be satisfied, a contradiction. We conclude that if for some j , $1 \leq j \leq n$,

$$r_j \xrightarrow{*}_E v_j,$$

then C is unsatisfiable. Conversely, assuming that there is no j , $1 \leq j \leq n$, such that

$$r_j \xrightarrow{*}_E v_j,$$

we shall construct a model \mathbf{M} of C . First, we make the $S(C)$ -indexed family $TERM(C)$ into a many-sorted Σ -algebra \mathbf{C} as in the proof of lemma 10.6.2. The new case is the case of symbols of sort *bool*. For every predicate symbol P of rank $(w_1 \dots w_k, \text{bool})$, for every k terms $y_1, \dots, y_k \in TERM(C)$, each y_i being of sort w_i , $1 \leq i \leq k$,

$$P_{\mathbf{C}}(y_1, \dots, y_k) = \begin{cases} \mathbf{T} & \text{if } Py_1 \dots y_k \in TERM(C) \text{ and } Py_1 \dots y_k \xrightarrow{*}_E \top; \\ \mathbf{T} & \text{if } Py_1 \dots y_k \notin TERM(C), \text{ there are terms } z_1, \dots, z_k \\ & \text{such that, } y_i \xrightarrow{*}_E z_i, Pz_1 \dots z_k \in TERM(C), \\ & \text{and } Pz_1 \dots z_k \xrightarrow{*}_E \top; \\ \mathbf{F} & \text{otherwise.} \end{cases}$$

Next, we prove that $\xrightarrow{*}_E$ is a congruence on \mathbf{C} . The case of symbols of sort $\neq \text{bool}$ has already been treated in the proof of lemma 10.6.2, and we only need to consider the case of predicate symbols. For every predicate symbol P of rank $(w_1 \dots w_k, \text{bool})$, for every k pairs of terms $(y_1, z_1), \dots, (y_k, z_k)$, with y_i, z_i of sort w_i , $1 \leq i \leq k$, if $y_i \xrightarrow{*}_E z_i$, then:

(1) $Py_1 \dots y_k \in TERM(C)$ and $Pz_1 \dots z_k \in TERM(C)$. Since $y_i \xrightarrow{*}_E z_i$ and $\xrightarrow{*}_E$ is a graph congruence, we have $Py_1 \dots y_k \xrightarrow{*}_E Pz_1 \dots z_k$. Hence, $Py_1 \dots y_k \xrightarrow{*}_E \top$ iff $Pz_1 \dots z_k \xrightarrow{*}_E \top$, that is, $P_{\mathbf{C}}(y_1, \dots, y_k) = \mathbf{T}$ iff $P_{\mathbf{C}}(z_1, \dots, z_k) = \mathbf{T}$.

(2) $Py_1 \dots y_k \notin TERM(C)$ or $Pz_1 \dots z_k \notin TERM(C)$. In this case, if $P_{\mathbf{C}}(y_1, \dots, y_k) = \mathbf{T}$, this means that there are terms $z'_1, \dots, z'_k \in TERM(C)$ such that, $y_i \xrightarrow{*}_E z'_i$, $Pz'_1 \dots z'_k \in TERM(C)$, and $Pz'_1 \dots z'_k \xrightarrow{*}_E \top$. Since $y_i \xrightarrow{*}_E z_i$, we also have $z_i \xrightarrow{*}_E z'_i$. Since $Pz'_1 \dots z'_k \in TERM(C)$, and $Pz'_1 \dots z'_k \xrightarrow{*}_E \top$, we have, $P_{\mathbf{C}}(z_1, \dots, z_k) = \mathbf{T}$. The same argument shows that if $P_{\mathbf{C}}(z_1, \dots, z_k) = \mathbf{T}$, then $P_{\mathbf{C}}(y_1, \dots, y_k) = \mathbf{T}$. Hence, we have shown that $P_{\mathbf{C}}(y_1, \dots, y_k) = \mathbf{T}$ iff $P_{\mathbf{C}}(z_1, \dots, z_k) = \mathbf{T}$.

This concludes the proof that $\xrightarrow{*}_E$ is a congruence. Let \mathbf{M} be the quotient of the algebra \mathbf{C} by the congruence $\xrightarrow{*}_E$, as defined in Subsection 2.5.8. Let v be the assignment defined as follows: For every variable x occurring in some term in $TERM(C)$, $v(x)$ is the congruence class $[x]$ of x .

As in the proof of lemma 10.6.2, it is easily shown that for every term t in $TERM(C)$,

$$t_{\mathbf{M}}[v] = [t],$$

the congruence class of t . But then, C is satisfied by v in \mathbf{M} . The case of equations not involving predicate symbols is treated as in the proof of lemma 10.6.2. Clearly, for every equation $Py_1\dots y_k \equiv \top$ in C' , by the definition of \mathbf{M} , $\mathbf{M} \models Py_1\dots y_k[v]$. Since we have assumed that for every negation $\neg(Py_1\dots y_k \equiv \top)$ in C' , it is not the case that $Py_1\dots y_k \xrightarrow{*}_E \top$, by the definition of \mathbf{M} , $\mathbf{M} \models \neg Py_1\dots y_k[v]$. This concludes the proof. \square

The above lemma provides a decision procedure for quantifier-free formulae.

EXAMPLE 10.6.3

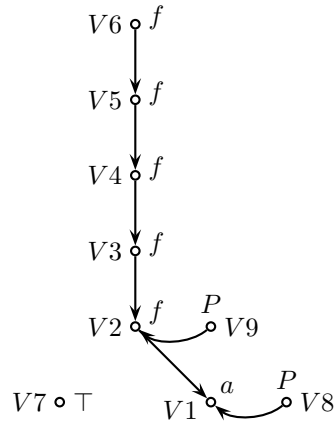
Consider the following conjunction C over a (one-sorted) first-order language:

$$f(f(f(a))) \doteq a \wedge f(f(f(f(f(a)))) \doteq a \wedge P(a) \wedge \neg P(f(a)),$$

where f is a unary function symbol and P a unary predicate. First, C is converted to C' :

$$f(f(f(a))) \doteq a \wedge f(f(f(f(f(a)))) \doteq a \wedge P(a) \equiv \top \wedge \neg P(f(a)) \equiv \top.$$

The graph $G(C)$ corresponding to C' is the following:



Initially, $R = \{(V1, V4), (V1, V6), (V8, V7)\}$, and let R' be its congruence closure. Since $(V1, V4)$ is in R , $(V2, V5)$ is in R' . Since $(V2, V5)$ is in R' , $(V3, V6)$ is in R' . Since both $(V1, V6)$ and $(V3, V6)$ are in R' , $(V1, V3)$ is in R' . Hence, $(V2, V4)$ is in R' . So the nodes $V1, V2, V3, V4, V5, V6$ are all equivalent under R' . But then, since $(V1, V2)$ is in R' , $(V8, V9)$ is in R' , and since $(V8, V7)$ is in R , $(V9, V7)$ is in R' . Consequently, $P(f(a))$ is equivalent to \top , and C' is unsatisfiable.

10.6.6 Computing the Congruence Closure

We conclude this section by giving an algorithm due to Nelson and Oppen (Nelson and Oppen, 1980) for computing the congruence closure R' of a relation R on a graph G . This algorithm uses a procedure *MERGE* that, given a graph G , a congruence R on G , and a pair (u, v) of nodes, computes the congruence closure of $R \cup \{(u, v)\}$. An equivalence relation is represented by its corresponding partition; that is, the set of its equivalence classes. Two procedures for operating on partitions are assumed to be available: *UNION* and *FIND*.

UNION(u, v) combines the equivalence classes of nodes u and v into a single class. *FIND*(u) returns a unique name associated with the equivalence class of node u .

The recursive procedure *MERGE*(R, u, v) makes use of the function *CONGRUENT*(R, u, v) that determines whether two nodes u, v are congruent.

Definition 10.6.4 (Oppen and Nelson's congruence closure algorithm)

Function *CONGRUENT*

```

function CONGRUENT( $R'$ : congruence;  $u, v$ : node): boolean;
  var  $flag$ : boolean;  $i, n$ : integer;
  begin
    if  $\Lambda(u) = \Lambda(v)$  then
      let  $n = |w|$  where  $r(\Lambda(u)) = (w, s)$ ;
       $flag := \mathbf{true}$ ;
      for  $i := 1$  to  $n$  do
        if  $FIND(u[i]) \neq FIND(v[i])$  then
           $flag := \mathbf{false}$ 
        endif
      endfor;
       $CONGRUENT := flag$ 
    else
       $CONGRUENT := \mathbf{false}$ 
    endif
  end

```

Procedure *MERGE*

```

procedure MERGE(var  $R'$ : congruence;  $u, v$ : node);
  var  $X, Y$ : set-of-nodes;  $x, y$ : node;
  begin
    if  $FIND(u) \neq FIND(v)$  then
       $X :=$  the union of the sets  $P_x$  of predecessors of all
        nodes  $x$  in  $[u]$ , the equivalence class of  $u$ ;
    endif
  end

```

```

Y := the union of the sets  $P_y$  of predecessors of all
    nodes  $y$  in  $[v]$ , the equivalence class of  $v$ ;
UNION( $u, v$ );
for each pair  $(x, y)$  such that  $x \in X$  and  $y \in Y$  do
    if  $FIND(x) \neq FIND(y)$  and  $CONGRUENT(R', x, y)$ 
    then
        MERGE( $R', x, y$ )
    endif
endfor
endif
end

```

In order to compute the congruence closure R' of a relation $R = \{(u_1, v_1), \dots, (u_n, v_n)\}$ on a graph G , we use the following algorithm, which computes R' incrementally. It is assumed that R' is a global variable.

```

program closure;
var  $R'$ : relation;
function CONGRUENT( $R'$ : congruence;  $u, v$ : node): boolean;
procedure MERGE(var  $R'$ : congruence;  $u, v$ : node);
begin
    input( $G, R$ );
     $R' := Id$ ; {the identity relation on the set  $V$  of nodes}
    for each  $(u_i, v_i)$  in  $R$  do
        MERGE( $R', u_i, v_i$ )
    endfor
end

```

To prove the correctness of the above algorithm, we need the following lemma.

Lemma 10.6.5 (Correctness of the congruence closure algorithm) Given a finite graph G , a congruence R on G , and any pair (u, v) of nodes in G , let R_1 be the least equivalence relation containing $R \cup \{(u, v)\}$, and R_3 be the congruence closure of $R_1 \cup R_2$, where

$$R_2 = \{(x, y) \in X \times Y \mid CONGRUENT(R_1, x, y) = \mathbf{true}\},$$

with

X = the union of the sets P_x of predecessors of all nodes x in $[u]$, the equivalence class of u (modulo R_1), and

Y = the union of the sets P_y of predecessors of all nodes y in $[v]$, the equivalence class of v (modulo R_1).

Then the relation R_3 is the congruence closure of $R \cup \{(u, v)\}$.

Proof: First, since R_3 is the congruence closure of $R_1 \cup R_2$ and R_1 contains $R \cup \{(u, v)\}$, R_3 is a congruence containing $R \cup \{(u, v)\}$. To show that it is the smallest one, observe that for any congruence R' containing $R \cup \{(u, v)\}$, by the definition of a congruence, R' has to contain all pairs in R_2 , as well as all pairs in R and (u, v) . Hence, any such congruence R' contains R_3 . \square

Using lemma 10.6.5, it is easy to justify that if $MERGE(R, u, v)$ terminates, then it computes the congruence closure of $R \cup \{(u, v)\}$. The only fact that remains to be checked is that the procedure terminates. But note that $MERGE(R', u, v)$ calls $UNION(u, v)$ iff u and v are not already equivalent, before calling $MERGE(R', x, y)$ recursively iff x and y are not equivalent. Hence, every time $MERGE$ is called recursively, the number of inequivalent nodes decreases, which guarantees termination. Then, the correctness of the algorithm closure is straightforward. \square

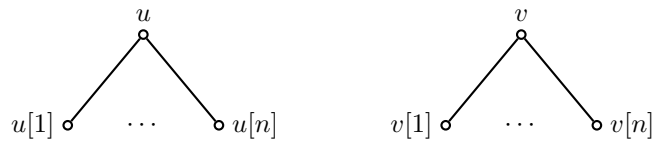
The complexity of the procedure $MERGE$ has been analyzed in Nelson and Oppen, 1980. Nelson and Oppen showed that if G has m edges and G has no isolated nodes, in which case the number of nodes n is $O(m)$, then the algorithm can be implemented to run in $O(m^2)$ -time. Downey, Sethi, and Tarjan (Downey, 1980) have also studied the problem of congruence closure, and have given a faster algorithm running in $O(m \log(m))$ -time.

It should also be noted that there is a dual version of the congruence closure problem, the *unification closure problem*, and also a symmetric version of the problem, which have been investigated in Oppen, 1980a, and Downey, 1980. Both have applications to decision problems for certain classes of formulae. For instance, in Oppen, 1980a, the symmetric congruence closure is used to give a decision procedure for the theory of recursively defined data structures, and in Nelson and Oppen, 1980, the congruence closure is used to give a decision problem for the theory of list structures. Other applications of the concept of congruence closure are also found in Oppen, 1980b; Nelson and Oppen, 1979; and Shostak, 1984b.

The dual concept of the congruence closure is defined as follows. An equivalence relation R on a graph G is a *unification closure* iff, for every pair (u, v) of nodes in V^2 , whenever $uR_s v$ then:

- (1) Either $\Lambda(u) = \Lambda(v)$, or one of $\Lambda(u)$, $\Lambda(v)$ is a variable;
- (2) If $\Lambda(u) = \Lambda(v)$ and $r(\Lambda(u)) = (s_1 \dots s_n, s)$, then for every i , $1 \leq i \leq n$, $u[i]R_{s_i} v[i]$.

Graphically, if u and v are two nodes labeled with the same symbol f of rank $(s_1 \dots s_n, s)$, if $u[1], \dots, u[n]$ are the successors of u and $v[1], \dots, v[n]$ are the successors of v ,



if u and v are equivalent then $u[i]$ and $v[i]$ are equivalent for all $i, 1 \leq i \leq n$. Hence, we have a kind of forward closure. In contrast with the congruence closure, the least unification closure containing a given relation R does not necessarily exist, because condition (1) may fail (see the problems).

The unification closure problem has applications to the unification of trees, and to the equivalence of deterministic finite automata. There is a linear-time unification closure algorithm due to Paterson and Wegman (Paterson and Wegman, 1978) when a certain acyclicity condition is satisfied, and in the general case, there is an $O(m\alpha(m))$ -time algorithm, where α is a functional inverse of Ackermann's function. For details, the reader is referred to Downey, 1980. An $O(m\alpha(m))$ -time unification algorithm is also given in Huet, 1976.

PROBLEMS

10.6.1. Prove lemma 10.6.1.

10.6.2. Give the details of the proof of lemma 10.6.3.

10.6.3. Use the method of Subsection 10.6.3 to show that the following formula is valid:

$$x \doteq y \supset f(x, y) \doteq f(y, x)$$

10.6.4. Use the method of Subsection 10.6.3 to show that the following formula is valid:

$$f(f(f(a))) \doteq a \wedge f(f(a)) \doteq a \supset f(a) \doteq a$$

10.6.5. Use the method of Subsection 10.6.5 to show that the following formula is valid:

$$x \doteq y \wedge g(f(x, y)) \doteq h(f(x, y)) \wedge P(g(f(x, y)) \supset P(h(f(y, x)))$$

10.6.6. Use the method of Subsection 10.6.5 to show that the following formula is not valid:

$$x \doteq y \wedge g(z) \doteq h(z) \wedge P(g(f(x, y))) \supset P(h(f(y, x)))$$

* **10.6.7.** An equivalence relation R on a graph G is a *unification closure* iff, for every pair (u, v) of nodes in V^2 , whenever $uR_s v$ then:

- (1) Either $\Lambda(u) = \Lambda(v)$, or one of $\Lambda(u)$, $\Lambda(v)$ is a variable;
- (2) If $\Lambda(u) = \Lambda(v)$ and $r(\Lambda(u)) = (s_1 \dots s_n, s)$, then for every i , $1 \leq i \leq n$, $u[i]R_{s_i}v[i]$.
- (a) Given an arbitrary relation R_0 on a graph G , give an example showing that the smallest unification closure containing R_0 does not necessarily exist, because condition (1) may fail.
- (b) Using the idea of Subsection 10.6.4, show that there is an algorithm for deciding whether the smallest unification closure containing a relation R_0 on G exists, and if so, for computing it.

* **10.6.8.** In order to test whether two trees t_1 and t_2 are unifiable, we can compute the unification closure of the relation $\{t_1, t_2\}$ on the graph $G(t_1, t_2)$ constructed from t_1 and t_2 as follows:

- (i) The set of nodes of $G(t_1, t_2)$ is the set of all subterms of t_1 and t_2 .
- (ii) Every subterm that is either a constant or a variable is a terminal node labeled with that symbol.
- (iii) For every subterm of the form $fy_1 \dots y_k$, the label is f , and there is an edge from $fy_1 \dots y_k$ to y_i , for each i , $1 \leq i \leq k$.

Let R be the least unification closure containing the relation $\{t_1, t_2\}$ on the graph $G(t_1, t_2)$, if it exists. A new graph $G(t_1, t_2)/R$ can be constructed as follows:

- (i) The nodes of $G(t_1, t_2)/R$ are the equivalence classes of R .
- (ii) There is an edge from a class C to a class C' iff there is an edge in $G(t_1, t_2)$ from some node y in class C to some node z in class C' .

Prove that t_1 and t_2 are unifiable iff the unification closure R exists and the graph $G(t_1, t_2)/R$ is acyclic. When t_1 and t_2 are unifiable, let $\langle C_1, \dots, C_n \rangle$ be the sequence of all equivalence classes containing some variable, ordered such that if there is a path from C_i to C_j , then $i < j$. For each i , $1 \leq i \leq n$, if C_i contains some nonvariable term, let t_i be any such term, else let t_i be any variable in C_i . Let $\sigma_i = [t_i/z_1, \dots, t_i/z_k]$, where $\{z_1, \dots, z_k\}$ is the set of variables in C_i , and let $\sigma = \sigma_1 \circ \dots \circ \sigma_n$. Show that σ is a most general unifier.

(A cycle in a graph is a sequence v_1, \dots, v_k , of nodes such that $v_1 = v_k$, and there is an edge from v_i to v_{i+1} , $1 \leq i \leq k-1$. A graph is acyclic iff it does not have any cycle.)

* **10.6.9.** Let C be a quantifier-free formula of the form

$$(s_1 \doteq t_1) \wedge \dots \wedge (s_n \doteq t_n) \supset (s \doteq t).$$

Let $COM(C)$ be the conjunction of all formulae of the form

$$\forall x \forall y (f(x, y) \doteq f(y, x)),$$

for every binary function symbol f in C .

Show that the decision procedure provided by the congruence closure algorithm can be adapted to decide whether formulae of the form $COM(C) \supset C$ are valid.

Hint: Make the following modification in building the graph $G(C)$: For every term ft_1t_2 , create a node labeled ft_1t_2 having two ordered successors:

A terminal node labeled f , and a node labeled with $v(ft_1t_2)$, such that $v(ft_1t_2)$ has two unordered successors labeled with t_1 and t_2 .

Also modify the definition of a congruence, so that for any two nodes u and v labeled with $v(ft_1t_2)$, where f is a binary function symbol, if either

- (i) $u[1]$ is congruent to $v[1]$ and $u[2]$ is congruent to $v[2]$, or
- (ii) $u[1]$ is congruent to $v[2]$ and $u[2]$ is congruent to $v[1]$,

then u and v are congruent.

Notes and Suggestions for Further Reading

Many-sorted first-order logic is now used quite extensively in computer science. Its main uses are to the definition of abstract data types and to programming with rewrite rules. Brief presentations of many-sorted first-order logic can be found in Enderton, 1972, and Monk, 1976. A pioneering paper appears to be Wang, 1952.

The literature on abstract data types and rewrite rules is now extensive. A good introduction to abstract data types can be found in the survey paper by Goguen, Thatcher, Wagner, and Wright, in Yeh, 1978. For an introduction to rewrite rules, the reader is referred to the survey paper by Huet and Oppen, in Book, 1980, and to Huet, 1980.

Congruence closure algorithms were first discovered by Kozen (Kozen, 1976, 1977), and independently a few years later by Nelson and Oppen (Nelson and Oppen, 1980, Oppen, 1980a), and Downey, Sethi, and Tarjan (Downey, 1980). The extension given in Subsection 10.6.4 appears to be new. Problems 10.6.7 and 10.6.8 are inspired from Paterson and Wegman, 1978, where a linear-time algorithm is given, and problem 10.6.9 is inspired from Downey, 1980, where fast algorithms are given.