SORTED UNIFICATION AND THE SOLUTION OF SEMI-LINEAR MEMBERSHIP CONSTRAINTS

BY

TOMÁS ESTEBAN URIBE

B.S., University of Illinois at Urbana-Champaign, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

Abstract

This thesis describes a new representation for sortal constraints and a unification algorithm for the corresponding constrained terms. Variables range over sets of terms described by *systems of set constraints*, which can express limited inter-variable dependencies. These sets of terms are more general than regular tree languages, but are still closed under intersection. The new unification algorithm shows sorted unification to be decidable for a broad class of sorted signatures, which we call *semi-linear*, and, more generally, for sort theories with a least Herbrand model that can be represented using the new constraints.

Even though the unification problem in question is $\mathcal{NP}$-hard, the generality of the algorithm may allow for particular efficient implementations for more restricted theories. This unification algorithm can be integrated into any of a variety of deductive systems, resulting in a hybrid substitutional reasoner. As an example, the soundness and completeness of a resolution-based deductive system that uses the new unification procedure is proved.

To my parents, Fernando and Cecilia.

Acknowledgements

Many people helped make this thesis possible.

I would like to thank my advisor, Professor Alan M. Frisch, whose ideas motivated most of the research presented in this thesis, and whose comments and suggestions were always helpful and insightful. I would also like to thank current and previous members of the Knowledge Representation and Reasoning Group, Peter Haddawy, Michael Mendelson, Mike Mitchell, David Page, Christian Prehofer, Richard Scherl and Herry Sutanto, with whom I had useful discussions about this work and many other subjects. Thanks in general to the Artificial Intelligence group at the Beckman Institute, and in particular everyone at 2414, who provided a friendly and stimulating research environment. I also had the opportunity to talk with Anthony Cohn, who also offered helpful comments on this research.

This work is dedicated to my family, and in particular my parents, Fernando and Cecilia, whose love and support has been unwavering throughout the years. My uncle, Rodrigo Restrepo, has also been an essential source of support and encouragement. My brother Daniel, his wife, Olga Forero, and their son Julián receive very special thanks for their invaluable assistance during most of my stay in the United States. The final corrections made to this thesis were possible only thanks to their generous assistance, since they provided shelter for the homeless author. Furthermore, their excellent culinary skills helped me recover from long weeks of malnutrition. Special thanks also go to the rest of my family back home in Colombia, (wonderfully extended through all those marriages I've missed): Luis Fernando, Ana Cecilia and Ernesto and Mariana and David, Miguel and Diana, Félix, and Juan Pablo and Carolina.

Finally, I want to acknowledge all the other friends, now spread across the world, who helped make my stay in Urbana-Champaign a very interesting one, as well as those old friends, both in the United States and Colombia, who have remained friends throughout. They know who they are, and most of them will probably never read this thesis anyway.

This research was partially supported by NASA under grant number NAG 1-613 and by the University of Illinois.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Specialized constraint-solving procedures can improve the efficiency and expressiveness of a general-purpose deductive system. The resulting systems are often called *hybrid reasoners*, since they combine two or more separate inference mechanisms that may use different representations for the knowledge that they reason about.

One of the preferred ways of integrating special-purpose reasoning into a wide range of systems is through *unification*, a fundamental operation in most deductive tasks. Specialized unification provides a way to cleanly separate the reasoning about the background knowledge from the activities of the main inference engine, so that an independent module can carry out the background deduction work. In this way, hybrid reasoners can be built by integrating new unifiers into existing systems. This approach has the advantage of preserving the higher-level procedures of the general-purpose reasoner (search strategies and refinements in the case of resolution-based theorem proving, for example) and simplifies ensuring that the new system retains the completeness and correctness properties of the original one.

A well-known instance of this approach is sorted logic, which has become increasingly popular in many Computer Science applications including many areas in Artificial Intelligence. It is quite natural, when reasoning about objects in a given domain, to want to classify them in some way; we usually make statements about *subsets* of the domain of discourse, rather than the entire universe. A sorted logic does just this: It classifies the elements of the domain by grouping them into *sorts*, which can then be described as background information. Specialized methods for representing and manipulating the taxonomic knowledge encoded in the sorts can improve the efficiency of deductive procedures.

Sorted unification is the process of solving equations between terms while respecting sortal restrictions imposed on their variables, given the background information that describes the different sorts. Specialized procedures can be used *within* the unification component to manipulate this information, again increasing efficiency. The complexity of the background information can vary greatly; as it increases, so does the complexity of unification.

The first part of this chapter is organized as follows: Section 1.1 is a summary of the research presented in this thesis. Section 1.2 is a brief description of sorted logic. Section 1.3 reviews related work on constraint deduction, sorted unification and tree languages. An overview of the rest of this thesis is presented in Section 1.4.

1.1 Objective

This thesis describes a new representation for monadic constraints and a unification algorithm for the corresponding constrained terms. Variables range over sets of terms described by a particular class of *systems of set constraints* (SSC's), which can express limited inter-variable dependencies. These sets of terms are more general than regular tree languages, but are still closed under intersection. The new unification algorithm shows sorted unification to be decidable for a broad class of sorted signatures, which we call *semi-linear*, and, more generally, for sort theories with a least Herbrand model that can be represented using the new constraints.

The constraints on variables are *membership constraints* in the sense that a variable restricted by an SSC can be seen as schematic for the set of ground terms described by the constraint, just as a regular variable is schematic for the set all ground terms. We will see that general SSC's are too powerful a representation, making unification undecidable, and so we focus on a particular subclass, that of *shallow* systems of set constraints. These shallow SSC's are similar to the set equations in [Heintze and Jaffar, 1990], on which they are based, which in turn are equivalent to term grammars (i.e. generate regular tree languages). Shallow SSC's are a generalization of these, as is made clear in Section 2.6.

The unification algorithm for terms constrained by shallow SSC's has two main components: The first, presented in Chapter 3, rewrites a shallow SSC into an equivalent one in a *solved form*, such that the satisfiability of membership constraints can be easily decided. The second component, presented in Chapter 4, finds a *normal form* for a set of equations and general membership constraints on terms.

In the context of sorted unification, a sort theory Σ is taken to be just that: A set of logical axioms that defines a number of unary predicates, the *sort predicates*, which are used to restrict the range of variables in the main logic. The new unification algorithm can be used to decide sorted unification problems for all sort theories Σ that have a least Herbrand model, M_Σ, such that M_Σ can be expressed as an SSC that satisfies a number of simple conditions. Given a representation for M_Σ as a shallow SSC, such as that obtained in [Heintze and Jaffar, 1990] for a certain class of logic programs, sorted unification under the theory Σ can be carried out.

All specifications of order-sorted logics need to impose restrictions on the sortal information (i.e. the sort theory or signature) in order to ensure the decidability of sorted unification. Implicitly, these restrictions are such that the sets of ground terms of each sort are at most as general as *regular* tree languages. The new unification algorithm shows unification to be decidable when the ground terms of each sort are a more general class of tree languages, that representable using shallow SSC's. In Chapter 6 we show how a large class of sorted signatures, which we call *semi-linear*, can be "translated" into shallow SSC's, proving that sorted unification is decidable for a broader class of sorted signatures than any previously known.

Shallow SSC's provide a way of representing complex, structured sets of terms. When used to represent sortal constraints, SSC's can reduce the number of unifiers even in the case of simple monomorphic sort theories. In the general polymorphic case, a finite number of unifiers is always obtained using the new constraints, including those cases where a minimal set of ordinary most general sorted unifiers has to be infinite. Thus, we can collapse into a single unification step what would correspond to an infinite sequence of deductions in the traditional sorted or unsorted setting. The unification algorithm possibly adds new sorts (expressed within SSC's) as the need arises, resulting in an *incremental* procedure.

This unification algorithm can be integrated into any of a variety of deductive systems, resulting in a hybrid substitutional reasoner. As an example, the soundness and completeness of a resolution-based deductive system that uses the new unification procedure is proved (Chapter 7).

The new approach is not without potential problems. As the expressiveness of the constraints increases, so does the complexity of unification,[1] and in fact, the unification problem for the new constraints is shown to be $\mathcal{NP}$-hard in general. However, the generality of the unification algorithm may allow for particular efficient implementations in restricted cases. Furthermore, a number of practical systems use unification procedures that solve intractable problems (so that they can take exponential time, or worse, in some cases) but that seem to work well for most applications. As Nebel and Smolka point out, even though computational complexity results can help understand the nature of problems, "*...one should not be scared by intractability in the first place. It may well be the case that it is possible to find algorithms that are well-behaved in all 'normal cases'*" [1990, page 124].

Furthermore, even when the overhead of sorted unification is comparable to that of the corresponding set of unsorted deductions, a deductive system that uses sorted logic can still be more efficient than its unsorted counterpart [Abrams and Frisch, 1989]: The use of sorts can add, implicitly, a new measure of control to the search [Cohn, 1986].[2]

1.2 An Overview of Sorted Logic

A thorough survey of sorted logic and unification could make up a thesis by itself, so we are purposely general here. Just as there are many uses for sorted logics, their specification can vary considerably from case to case, even though the underlying ideas are usually quite similar. For a comprehensive description of different sorted logics and their use, see [Cohn, 1989b].

Sort theories can be broadly classified along various dimensions. Most sorted logics specify constraints on terms through the use of *sorted variables*, which are each assigned to a unique sort. For example, to indicate that the variable x is restricted to range over the integers we can write x:INTEGER, and the sum of an odd and an even number can be expressed as the term $sum(x{:}\text{EVEN}, y{:}\text{ODD})$. We call these **atomic** constraints; as we will see, more general methods for specifying sortal information can be used.

There is the question of whether sorts are syntactically specified (so that *terms* are assigned sorts) or have a model-theoretic (semantic) interpretation (so that it is the *objects in the domain* that are assigned sorts). In the first case, one can speak of "well-sorted" and "ill-sorted" *terms*, indicating whether they satisfy the syntactic restrictions imposed by the sort theory (usually called the sorted signature, in this case); one can also restrict the arguments taken on by different predicates to be of certain sorts. In the second case, all terms and predicates are syntactically correct.[3] The motivation for defining ill-sorted terms and formulas is that functions and predicates are often *partial*, that is, they are only defined for arguments that belong to particular subsets of the domain.

[1] This is one of the main reasons why much of the work carried out in type specification for logic programming is geared towards eliminating the need for *dynamic* type checking. This requires imposing severe restrictions on the sort structures that are allowed.

[2] This, of course, assumes that unification is decidable in the first place.

[3] This problem is similar to the view of Prolog programs as being *prescriptively* or *descriptively* typed [Reddy and Lakshman, 1991].

If equational theories are not considered, the distinction between sort theories and sorted signatures does not have any practical relevance; sorted signatures can be thought of as a particular way of specifying a sort theory. This is proved in Chapter 5, which presents the precise definitions of sort theories and sorted signatures.

The simplest sort theories divide the domain into *disjoint* sorts. The resulting sorted logics are sometimes called **many-sorted** — somewhat misleadingly, since the term is often also used to refer to sorted logic in general. The use of these many-sorted logics can be traced back to Herbrand [1930].

In contrast to the many-sorted case, **order-sorted** logics allow for non-empty intersections of sorts, and usually present a **sort hierarchy** that defines a partial order on the sorts, representing the subset relation. Thus, whereas in the many-sorted case a term is assigned a unique sort, in an order-sorted logic a term can belong to a number of different sorts.

For example, Figure 1.1 shows a simple (tree-structured) sort hierarchy similar to that used for solving the "Schubert steamroller" deduction problem using an order-sorted logic [Cohn, 1985; Walther, 1985a; Stickel, 1986]. Under this hierarchy, a variable of sort ANIMAL can be unified with any term of sort BIRD, FOX or SNAIL, but not with a term of sort PLANT.

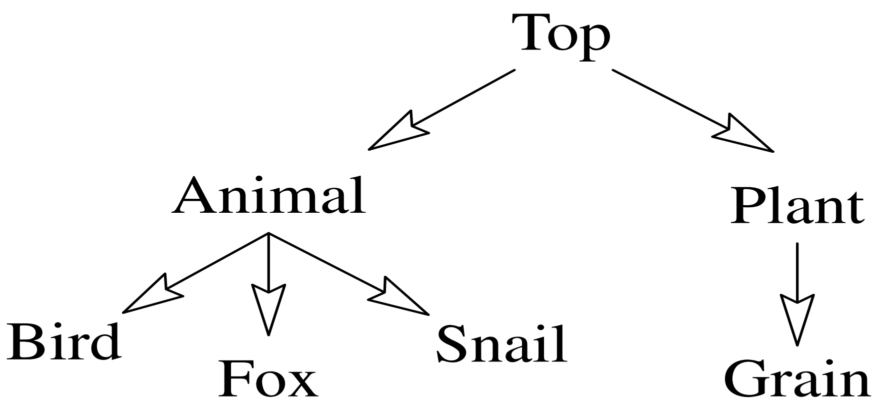

Figure 1.1: A simple sort hierarchy

Order-sorted logics can be roughly classified as **monomorphic** or **polymorphic**, though the actual definitions for these terms varies depending on the author. Here we use a generic one: A sort theory is *monomorphic* if the sorts assigned to a well-sorted term are uniquely determined by the outermost function symbol; that is, for a given f, all terms of the form $f(t_1, \ldots, t_n)$ belong to the same set of sorts, provided they are well-sorted.

In a *polymorphic* sort theory, the sorts to which a term belongs can depend on the sort of its subterms. In the more general case, this allows for the *overloading* of functions, in what is sometimes called *ad hoc* polymorphism. For example, one could specify that $plus(x, y)$ is of sort INTEGER if x and y are integers, of sort VECTOR if x and y are vectors, and of sort EVEN if either x and y are equal or both x and y are even.

Finally, we should note that what are often called "order-sorted signatures" in the literature are a special case of both sorted signatures with term declarations and the general sort theories that are considered in this thesis (see Section 5.5).

1.3 Related Work

Constraint Deduction

The general principles involved in the construction of hybrid deductive systems through the use of specialized unification are studied by Frisch [1989; 1991], who presents the *substitutional framework* for sorted deduction. This framework provides the theoretical basis for building hybrid deductive systems where monadic constraints are imposed on variables, and where a constrained variable is treated as schematic for all the ground instances that are known to satisfy the constraint. Proving the correctness and completeness of the resulting hybrid deductive systems can be done by systematically transforming the corresponding proofs for the original system. We follow this general procedure to construct the soundness and completeness proofs of Chapter 7. As an instance of the substitutional approach, this author modified FRAPPS[4] [Mitchell, 1989; Mitchell *et al.*, 1991], a system for constructing resolution-based theorem provers, so as to enable the introduction of an arbitrary unification procedure [Uribe, 1991].

The use of constraints in logic programming languages is a very active research area, in particular what is known as the *constraint logic programming* (CLP) paradigm [Jaffar and Lassez, 1987]. This includes systems such as Prolog III [Colmerauer, 1990], CLP($\Re$) [Jaffar *et al.*, 1990] and CHIP [Van Hentenryk, 1989]. Colmerauer [1982; 1984] considered extending the definition of terms in ordinary Prolog to include infinite *rational trees* (those that have a finite number of different subtrees), for which both equations and inequations are solved, as the foundation for Prolog II. Cohen, who presents an overview of constraint logic programming in [Cohen, 1990], points out that this was one of the first instances of a constraint language over a particular domain being embedded into a logic programming system.

A more general framework for constraints in logic programming is presented by Höhfeld and Smolka [1988] and used by Smolka [1989] in conjunction with *parametrically polymorphic types*.[5] Burckert [Burckert, 1990] presents a general theory of resolution with constraints.

See [Frisch and Cohn, 1991] for a survey of research on hybrid reasoning, as well as a discussion of the main issues associated with this area.

Sorted Unification

The idea of unification of logical terms is usually attributed to Herbrand [1930], but it was with the introduction of the resolution principle by Robinson [1965] that substitutions and unifiers as are now commonly known in automated deduction were introduced. Regular (unsorted) unification is discussed in [Martelli and Montanari, 1982], [Eder, 1985], and [Lassez *et al.*, 1988], to name a few. Knight [1989] presents a broad survey of different unification procedures and their applications.

[4] The Framework for Resolution-based Proof Procedure Systems.

[5] "Parametric" specifications of sorts include "meta-level" variables that stand for sets of sorts. The sort domains themselves in [Smolka, 1989] are still instances of regular tree languages.

Sorted logic has been used in general theorem proving [Cohn, 1983; Walther, 1987], knowledge retrieval and deductive databases [Frisch, 1986a], logic programming [Smolka, 1989], type specification for programming languages [Smolka, 1989; Reddy and Lakshman, 1991], rewrite rules [Schmidt-Schauß, 1989; Chen and Hsiang, 1991a; Comon, 1991], generalization [Frisch and Page, 1990], parsing logical languages [Frisch, 1986b], and natural language processing [Veronis, 1987], among other areas. Sorts have been shown to improve the efficiency of deductive systems (given the appropriate formulation of the problem at hand) in [Cohn, 1985], [Walther, 1985a], [Ohlbach and Schmidt-Schauß, 1985], [Stickel, 1986], [Cohn, 1989a], and [Abrams and Frisch, 1989].

Walther [1985b] examines sorted unification in the monomorphic case, allowing an infinite number of sorts, and shows how the existence and cardinality of a set of most general unifiers depend on properties of the lattice of sorts. Schmidt-Schauß [1989] presents a detailed analysis of the general problem of sorted unification, including polymorphism and equational theories, mostly from an algebraic point of view. The sort theory is presented as a **signature**, comprising a sort hierarchy and a set of **declarations**. We show in Chapter 5 how (again, in the absence of equality) the sorted signatures used by these two authors can be thought of as instances of the model-theoretic approach to sorts that we use.

Both Walther [1987] and Schmidt-Schauß [1989] present resolution-based deductive systems that build in taxonomic reasoning through sorts, and that can be classified as "substitutional" [Frisch, 1991].[6] These and most other sorted deductive systems use only atomic constraints on variables (as in x:INTEGER). However, Frisch [1991] proposes a framework that allows for more general monadic constraints, and uses as the main example a sorted logic in which restrictions on sorted variables are extended to include boolean expressions over sorts (as in x:INTEGER $\vee \neg$RATIONAL). The constraints defined in this thesis can be thought of as a variant of this scheme.

Cohn and Frisch [1990] have described a general procedure for sorted unification, given an oracle that answers certain kinds of queries about the sort theory. This procedure is revisited in [Cohn and Frisch, 1991], where the oracle is used as a source of appropriate well-sorted substitutions. In some sense, the unification algorithm presented in this thesis is an attempt to provide an effective implementation of such a procedure for as large a class of theories as possible.

With the exception of Walther's [1987] and Schmidt-Schauß's [1989] work on unification under equational theories (E-unification), most of the above work is closely related to the problems investigated in this thesis. E-unification is not considered in this thesis, and is also a broad and active research area (see [Jouannaud and Kirchner, 1990], or [Siekmann, 1990], for example).

Sorts and Tree Languages

Besides this thesis, other recent work on deduction with constraints and specialized unification also favors the view of sortal constraints as specifying tree languages of various kinds. Chen and Hsiang [1991b] propose a method for finitely representing, and unifying, structured infinite sets of terms, and show how these *recurrence domains* can be incorporated into the CLP framework.

[6] Other systems, like Cohn's LLAMA, [Cohn, 1983], cannot be classified as substitutional — see Chapter 8.

These domains are neither more nor less general than the class of tree languages generated by shallow SSC's.[7]

Chomicki and Imielinski [1989] restrict the possible occurrences of function symbols in a Horn clause database so that *congruence closure* can be used to finitely represent possibly infinite sets of answers; the least fixpoints of sets of rules are explicitly represented as classes in a congruence. Our constraints can achieve similar results in the more general context of resolution (see Chapter 7), though we do not use the congruence closure approach.[8]

Shallow SSC's are the "grammar-like" counterpart of a subclass of *tree automata with equality tests* presented by Bogaert and Tison [1991] and used by Comon and Delor [1991] to generalize a simplification algorithm initially presented in terms of regular tree languages [Comon, 1990]. Comon and Delor [1991] present a set of rules that normalizes equational formulae with membership constraints, where elementary sorted signatures are represented as tree automata.[9] Our unification algorithm can be seen as a instance of this procedure, using a different representation for constraints. The languages generated by SSC's correspond to Bogaert and Tison's [1991] $Rec_=$ class (i.e. are recognized by tree automata whose tests do not contain inequalities — see Section 2.6.3 for more details). Comon [1991] has also developed a procedure for second-order sorted unification for restricted kinds of terms[10] using membership constraints.

Heintze and Jaffar [1990] present an algorithm that approximates the least model of a logic program by ignoring inter-variable dependencies. The approximation itself is represented as a term grammar, and the algorithm is presented using set constraints. In [Heintze and Jaffar, 1991], a decision procedure to solve sets of subset relations between general set expressions is presented. This work was the source of the notation used in Chapter 2, where set constraints are allowed to express inter-variable dependencies.

As may already be clear from the above discussion, the following references have been particularly essential sources: [Schmidt-Schauß, 1989], [Heintze and Jaffar, 1990], [Cohn and Frisch, 1990], and [Frisch, 1991]. The origin of this thesis was an attempt to integrate some of the ideas and results in these works. While the algorithms presented here are independent of those in [Comon and Delor, 1991], that paper reinforced the approach towards sorted unification adopted here and helped decide on the title of this thesis. I would like to thank Hubert Comon and Bruno Bogaert, who made their papers available in a very timely fashion, and Nevin Heintze, Jieh Hsiang and Sophie Tison, who through electronic mail helped put this work in some perspective.

1.4 Overview

We begin by establishing some basic notation in Section 1.5. These are standard definitions, so the reader can skip this section and use it only for later reference, if necessary.

Sections 1.6 and 1.7 present the essential definitions concerning (unsorted) substitutions and unifiers, and some basic lemmas that are referred to later on. Unfortunately, a diverse array of

[7] In fact, they are neither more nor less general than regular tree languages.

[8] McAllester [1983] shows that the equivalence classes generated by the congruence closure of a finite set of equations is a regular tree language.

[9] An earlier paper, [Comon, 1989], is apparently the first one to do so, in an application of sorted logic to inductive proofs.

[10] The restriction on the second-order terms to be unified is the same one we apply to the term declarations in a sorted signature so as to make unification is decidable, that is, they should be *semi-linear* — see Chapter 6.

notations and definitions regarding substitutions and unifiers can be found in the literature. At best, the notation is different and the definitions are equivalent; but it is not unusual that the same notation is used to represent unequivalent, and sometimes opposite, definitions. Therefore, the reader is encouraged to take a look at these two sections before proceeding.

Chapter 2 introduces *systems of set constraints* (SSC's), including the particular subclass that we call *shallow* for which unification of constrained terms will be shown to be decidable. Section 2.6 describes the relationship between SSC's of various kinds and term grammars (which generate languages recognizable by finite tree automata).

Chapter 3 presents an procedure that rewrites a shallow SSC into an equivalent one in a *solved form*, together with procedures that answer some simple questions about these solved form SSC's.

Chapter 4 presents an procedure that rewrites a set of equations and membership constraints into a normal form, and then combines it with the procedures of Chapter 3 to obtain, finally, a unification algorithm for terms constrained by shallow SSC's.

The second part of this thesis describes how the new unification algorithm can be used to solve sorted unification problems for a wide range of sort theories and sorted signatures. Chapter 5 presents two of the main approaches to sorted logic: Unification under sorted signatures (without equality) is shown to be a special case of unification under sort theories. This chapter also includes previous results for some classes of sorted unification problems.

Chapter 6 describes the relationship between SSC's and logic programs, and presents a procedure for translating sort theories into equivalent shallow SSC's for a broad class of sorted signatures, which we call *semi-linear*.

Chapter 7 shows how the new unification algorithm can be integrated into a resolution-based deductive system. The soundness and completeness of such a system is then proved following the substitutional framework approach.

This thesis probably raises more questions than it answers; Chapter 8 presents a summary of the results, open problems and concluding remarks.

A Note on the Algorithms and Proofs

The design and description of sorted unification algorithms is often based on unsorted ones. The preferred method is to describe a set of transformations that can be applied to a given set of equations with little or no control necessary, resulting in a "solved form" equation set. The algorithms presented here retain this "transformational" approach which, as pointed out by Gallier and Snyder [1990], has the following advantages:

- Logic and control are cleanly separated, which allows for a large number of different implementations.

- Implementing the algorithm efficiently can be thought of as optimizing the application of the transformations.

- The algorithm often brings a "deeper understanding" of the problem being solved.

- The proofs of correctness and termination of the algorithm are usually clearer and easier.

I have not worried about making the algorithms efficient; in some instances, I have decided to omit some simple optimizations for the sake of clarity. Therefore, the algorithms presented in this thesis are not meant to be efficient if implemented literally. The problem of finding adequate data structures and efficient implementations is thus left open.

I have tried to present detailed and correct proofs for all of the new results in this thesis. I also tried to provide clear and formal definitions of all the new concepts (hoping that figuring out the notation is not harder than understanding the results); not many details are left to the reader, so the reader will have to bear with the details. The underlying intuitions are often quite simple; unfortunately, formalizing intuition is not always easy.

1.5 Preliminaries

Sets and Orderings

Given a set S, $|S|$ denotes the number of elements in S, otherwise known as the **cardinality** of S. We write $|S| = \infty$ if S is infinite.[11] We write $S \subseteq T$ or $T \supseteq S$ if S is a subset of T. If $S \subseteq T$ and $S \neq T$, we write $S \subset T$ and say that S is a **proper subset** of T. The "$\cup$" and "$\cap$" symbols denote the union and intersection operations, as usual. **Multisets** extend sets to allow repeated occurrences of their elements, and can be formalized as a set together with a function defined on each element that indicates the number of times the element appears in the associated multiset.

Given a set $\mathcal{S}$ and a function f defined on the elements of $\mathcal{S}$, $\bigcup_{x \in \mathcal{S}} f(S)$ denotes the union of $f(x)$ over all the elements x in S, and it might sometimes be abbreviated to $\bigcup f(x)$ if the context makes $\mathcal{S}$ clearly understood. Similarly, if P is a property that is true or false over the elements of some domain, $\{f(x) \mid P(x)\}$ denotes the set of all the values $f(x)$ for all those elements x in the domain for which P is true.[12]

A **multiset** is a collection that can have repeated occurrences of the same element. An **n-tuple** is an ordered multiset containing n elements, written as $\langle e_1, \ldots, e_n \rangle$. $\langle s_1, \ldots, s_n \rangle$, where $s_i \in S$. A **ranked** set S is one for which an integer $n \geq 0$ is assigned to each element $s \in S$; n is said to be the **arity** of s. Multisets can be formally specified as ranked sets. The sets of function and predicate symbols of first order logic are ranked sets.

A relation $\leq$ defined over the elements of a set S is a **quasi-ordering** (also known as **pre-ordering**) if it is reflexive and transitive. We may write $a \geq b$ when $b \leq a$, and $a < b$ means that $a \leq b$ but not $a \geq b$. The symbols $>$, $\not\leq$, and $\not\geq$ are also used in the usual way. A quasi-ordering $\leq$ generates an equivalence relation $\cong$ over the elements of S, where $a \cong b$ if $a \leq b$ and $b \leq a$. A **chain** is a sequence of elements such that each is strictly greater than the previous one. A quasi-ordering $\leq$ on S is **well-founded** if there is no infinite strictly descending chain, that is, an infinite sequence of elements $s_1, s_2, \ldots$ such that $s_i > s_{i+1}$ for all i.

A relation $\leq$ is a **partial order** if it is reflexive, transitive and anti-symmetric. A set S together with a partial ordering $\leq$ is said to be a partially ordered set, or **poset**.

Given a quasi-ordering $\leq$ defined on a set S, l is a **lower bound** of $S_o \subseteq S$ if $l \in S$ and $l \leq t$ for all $t \in S_o$; u is an **upper bound** of S_o if $u \in S$ and $t \leq u$ for all $t \in S_o$. Similarly, l is a **greatest lower bound** (g.l.b.) of S_o if l is a lower bound of S_o and for all lower bounds b of S_o, $b \leq l$; and u is a **least upper bound** (l.u.b.) of S_o if u is a upper bound of S_o and for all upper bounds b of S_o, $u \leq b$.

[11] Among infinite ordinals, we will only distinguish between the cardinality of countably infinite and uncountably infinite sets, if at all.

[12] We'll try our best to avoid Russell's paradox when using this notation.

A poset $\langle S, \leq \rangle$ is a **lattice** if every finite $S_o \subseteq S$ has both a l.u.b. and a g.l.b.; it is a **complete lattice** if this is true for all (possibly infinite) subsets of S. If S is finite, all lattices are complete lattices. It is a **semi-lattice** if only the existence of an l.u.b. is guaranteed. A g.l.b. (resp. l.u.b.) of $\{a, b\}$ exists for all $a, b \in S$ if, and only if, a g.l.b. (resp. l.u.b.) exists for all finite subsets of S.

Terms and Variables

Terms are built from a set of variables $\mathcal{V}$ and a ranked set of function symbols $\mathcal{F}$ in the usual way: function symbols of arity 0 are **constants**, and a **term** is either a variable, a constant, or $f(t_1, \ldots, t_n)$ for terms $t_1, \ldots, t_n$ and an n-ary function f, $n \geq 1$. Terms can be thought of as directed **trees**, where leaves are labeled with variable or constant symbols and the outgoing edges from each internal node (which is labeled with a function symbol) are ordered. We may sometimes write $f(\cdots)$ to denote any term whose outermost function symbol is f.

For any syntactic object P, $Var(P)$ denotes the set of variables occurring in P. A formula or a term t is **ground** if $Var(t) = \emptyset$. A term t is **linear** if no variable occurs twice in t.

For any term t, the **nesting depth** of t, denoted by $d(t)$, is defined as follows: $d(t) \stackrel{\text{def}}{=} 0$ if t is a variable or a constant; else, $d(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} 1 + \max\{d(t_1), \ldots, d(t_n)\}$. Similarly, the set of **subterms** of a term t, $Sub(t)$, can be defined as follows: For any variable or constant t, $Sub(t) \stackrel{\text{def}}{=} \{t\}$; and if $t = f(t_1, \ldots, t_n)$, then $Sub(t) \stackrel{\text{def}}{=} \{t\} \cup Sub(t_1) \cup \ldots \cup Sub(t_n)$. The terms $t_1, \ldots, t_n$ are the **immediate subterms** of t.By "induction on terms" we mean induction on the structure of terms, or, equivalently, induction on the nesting depth of terms.

Clausal Form

Given a ranked set of predicate symbols $\mathcal{P}$, an **atomic formula** (or atom) is $P(t_1, \ldots, t_n)$, where P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. A **literal** is one of ϕ, $\neg\phi$, for an atomic formula ϕ. A literal is **positive** if it is an atomic formula, and negative otherwise. A **clause** is a disjunction of literals $\phi_1 \vee \ldots \vee \phi_n$, and can be identified with the set that contains all the literals appearing in it. The **empty clause** is denoted by $\Box$. A **Horn** clause contains at most one positive literal. A **definite** clause contains exactly one positive literal. A Horn clause c can be written as $\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi$, where $\neg\phi_1, \ldots, \neg\phi_n$ are the negative literals in c and ψ is the one positive literal in c, if one exists, $\emptyset$ otherwise. The set of literals $\{\phi_1, \ldots, \phi_n\}$ is the **body** of the clause, while ϕ is the **head**. A **logic program** is a finite conjunction of Horn clauses.

In general, a set of literals can be identified with their disjunction (a clause), while a set of clauses is usually identified with their conjunction (a logical theory in clausal form).

The **universal closure** of a formula ϕ is the sentence $\forall x_1 \ldots \forall x_n(\psi)$, where $\{x_1, \ldots, x_n\}$ are the free variables in ψ; this sentence is denoted by $\overline{\forall}\psi$. Clauses are implicitly universally quantified over all their variables; that is, a clause $c = \{l_1, \ldots, l_n\}$ stands for $\overline{\forall}(l_1 \vee \ldots \vee l_n)$. A set of clauses S is **satisfiable** if there exists a model that makes the universal closure of each of the clauses in S true.

1.6 Substitutions

A **substitution** is a mapping from variables to terms that is the identity for all but a finite set of variables. A substitution θ will usually be written as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where each variable x_i is mapped to t_i, $x_i \neq x_j$ for $i \neq j$, and $x_i \neq t_i$ for all i.[13] The term t_i is denoted by $t_i\theta$.[14] The identity substitution, $\{\}$, will be denoted by ϵ. The set of all variables X such that $X \neq X\theta$ is the **domain** of θ, denoted by $Dom(\theta)$. The set of all terms t such that $t = X\theta$ for some $X \in Dom(\theta)$ is the **range** of θ, denoted by $Rng(\theta)$.[15]

A substitution induces a mapping from terms into terms defined as follows: If t is a variable, then $t\theta$ is given by the original definition; if t is a term $f(t_1, \ldots, t_n)$, then $t\theta \stackrel{\text{def}}{=} f(t_1\theta, \ldots, t_n\theta)$. The term $t\theta$ is said to be the result of **applying** θ to t, and is an **instance** of t. Thus, $t\theta$ is obtained by "simultaneously" replacing all the variables V in t by $V\theta$.

A substitution θ is a **ground substitution** over a set of variables S if $Var(X\theta) = \emptyset$ for all $X \in S$. If $Var(Rng(\theta)) = \emptyset$, then θ is a ground substitution for $Dom(\theta)$, and we will often simply say that θ is ground.

Substitutions, as functions from terms into terms, can be composed. The composition of two substitutions is always a substitution. Given substitutions θ and Ω, the substitution obtained by **composing** Ω onto θ is denoted by $\theta \cdot \Omega$ [16] and is the function that maps all variables $X \in V$ into $(X\theta)\Omega$. (That is, first apply θ to X, and *then* apply Ω to the result.) The composition of substitutions is associative, but not commutative. However, the empty substitution is the left and right identity; i.e. $\epsilon \cdot \theta = \theta \cdot \epsilon = \theta$ for all substitutions θ.

A substitution θ that maps variables into variables in a one-to-one fashion is called a **renaming** substitution; that is, θ is bijective over $Dom(\theta)$ — or, to say it in yet another way, θ is permutation of the variables in its (finite) domain. A renaming substitution π, being bijective, has a unique **inverse**, denoted by π^{-1}, such that $\pi \cdot \pi^{-1} = \pi^{-1} \cdot \pi = \epsilon$. Two substitutions θ and ϕ are **renamings** of each other if there is a renaming substitution π such that $\theta = \phi \cdot \pi$ (so equivalently $\theta \cdot \pi^{-1} = \phi$). Similarly, two terms t_1 and t_2 are renamings of each other if $t_1 = t_2\pi$ for some renaming substitution π.

A substitution θ is **more general** than a substitution ϕ, written as $\theta \geq \phi$,[17] if there exists a substitution ρ such that $\phi = \theta \cdot \rho$. If $\theta \geq \phi$ and $\phi \geq \theta$ we say that θ and ϕ are **equivalent**. Note that the $\leq$ relation for substitutions as defined above is a quasi-ordering, but is *not* a partial ordering. It is, however, a partial ordering on the equivalence sets generated by $\leq$ (as is the case for all quasi-orderings).

A substitution θ is **idempotent** if $\theta = \theta \cdot \theta$. That is, for all terms t, $t\theta = (t\theta)\theta$. We will make use of the following lemma, as found in [Eder, 1985].[18]

Lemma 1.1 *(i) A substitution Ω is idempotent if, and only if, $Dom(\Omega) \cap Var(Rng(\Omega)) = \emptyset$. (ii) If a substitution Ω is idempotent, then for any substitution θ, $\Omega \geq \theta$ if, and only if, $\Omega \cdot \theta = \theta$. That is, $\Omega \geq \theta$ if, and only if, for all terms t, $t\theta = (t\Omega)\theta$.*

---

[13] Other notations found in the literature include $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ and $\{t_1/x_1, \ldots, t_n/x_n\}$.

[14] Some authors use θt or $\theta(t)$ instead of $t\theta$.

[15] Some authors define the range of θ to be the set of *variables* appearing on all such t's. This set will be denoted here by $Var(Rng(\theta))$.

[16] Some authors use $\theta\Omega$, $\Omega\theta$ or $\Omega \cdot \theta$.

[17] Some authors, including Eder and Schmidt-Schauß, write $\theta \leq \phi$ instead.

[18] Under a different notation, that is.

1.7 Constraint-free Unification

Given a set of terms S, we say that a substitution θ is a **unifier** of S if $t_1\theta = t_2\theta$ for all $t_1, t_2 \in S$. A **most general unifier** (mgu) θ of a set S is a unifier for S such that for all other unifiers ρ of S, $\theta \geq \rho$. According to this definition, there can be many different mgu's for even a single pair of terms, which are *not* renamings of each other. As warned in [Lassez *et al.*, 1988, page 589]:

> *All mgus are equal, but some mgus are more equal than others.*

Some authors (most notably [Robinson, 1965]) use a stronger definition of generality, where most general unifiers are restricted to be *idempotent*. Idempotent mgu's never introduce any new variables, and all idempotent mgu's are renamings of each other. We will explicitly require the idempotency of an mgu when it essential that this be the case. It can be shown that if θ is an idempotent mgu of S, then ρ unifies S if, and only if, $\rho = \rho \cdot \theta$ for all substitutions ρ.

Given a set of sets of terms $\mathcal{S}$, θ is a unifier of $\mathcal{S}$ iff it is a unifier of each set $S \in \mathcal{S}$. The distinctions between unifying pairs, sets and sets of sets of terms can be avoided by simply considering *sets of equations* between terms: Given a set of equations $E = \{t_1 = t'_1, \ldots, t_n = t'_n\}$, we say that a substitution θ **solves**, or is a **solution** of, the set E if $t_i\theta = t'_i\theta$ for all i. We may sometimes also say that θ is a unifier of E. An idempotent most general solution for a set of equations can be obtained from a well-known algorithm that applies the transformations shown in Figure 1.2 to the equation set until no transformation applies.

1: $S_o \cup \{f(t_1, \ldots, t_n) = x\} \Longrightarrow S_o \cup \{x = f(t_1, \ldots, t_n)\}$.

2: $S_o \cup \{x = x\} \Longrightarrow S_o$.

3: $S_o \cup \{f_1(s_1, \ldots, s_n) = f_2(t_1, \ldots, t_m)\} \Longrightarrow S_o \cup \{s_1 = t_1, \ldots, s_n = t_n\}$,
given that $f_1 = f_2$ (and $n = m$); otherwise, return FAIL.

4: $S_o \cup \{x = t\} \Longrightarrow S_o\{x \mapsto t\} \cup \{x = t\}$,
where $S_o\{x \mapsto t\}$ is the result of replacing all the occurrences of x in S_o by t, provided $x \notin Var(t)$ and $x \in Var(S_o)$.
If $x \in Var(t)$ and $t \neq x$, return FAIL.

Figure 1.2: Transformations for unsorted unification

A concise proof of the termination and correctness of the non-deterministic application of these transformations can be found in [Lassez *et al.*, 1988].[19] The algorithm either outputs FAIL or terminates with an equation set in *solved form*, that is, $S = \{x_1 = t_1, \ldots, x_n = t_n\}$, for variables t_i and terms t_n, where $x_i \neq x_j$ if $i \neq j$. The output is FAIL iff the original set

[19] This is also a special case of the termination and correctness of the unification algorithm for constrained terms in Chapter 4.

of equations has no solution. Otherwise, the substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an *idempotent* most general solution for the input equation set.

Note that the transformations do not introduce any new variables, so that $Var(Rng(\theta)) \subseteq Var(S)$, where S is the original equation set.

Martelli and Montanari [1982] present a variant of this algorithm that runs in time $O(n + v\log(v))$, where n is the number of function symbols and v the number of variables in the original equation set. Other *quasi-linear* algorithms exist, usually of order $O(n + m * f(m))$, where n is the size of the input, m is the number of variable occurrences, and f is a "very slow growing" function, like the inverse of the Ackermann function. Truly linear unification algorithms exist, but for practical purposes the quasi-linear ones are preferred (see [Siekmann, 1990]).

Note that unification can be seen as the process of *solving equations over the Herbrand Universe*. In fact, the equation-based approach to unification algorithms is also traced back to Herbrand's thesis [1930]. For efficiency purposes, some implementations of unification (including that for Prolog) do not include the **occurs check** (in transformation 4 above), which detects the unsolvability of an equation $x = t$ where $t \neq x$ and x appears in t. A stricter control on the transformations is necessary in this case to ensure termination. The absence of the occurs check endangers the correctness of the system in which the unification is embedded; for example, it can lead to incorrect results in Prolog if the programs are not carefully written (see [Knight, 1989]).

Chapter 2

Set Constraints

The **systems of set constraints** (SSC's) defined in this chapter are a tool for describing and manipulating tree languages, and can also be used to represent sorted signatures and sortal constraints. We will see in Chapter 5 that systems of set constraints are very similar to the general sorted signatures with term declarations presented by Schmidt-Schauß [1989]. In particular, we will focus on *shallow* SSC's; some important questions are decidable for the languages they represent, so that unification is also decidable when they are used in membership constraints, as will be shown in Chapter 4. Shallow SSC's can be used to represent a broad class of sorted signatures, that properly includes all linear ones (see Chapter 6). The reason for this greater generality is that these SSC's can express certain inter-variable dependencies, so that the resulting tree languages are not always regular.

Our set constraints correspond to the solved form set constraints investigated by Heintze and Jaffar [1990; 1991], with added inter-variable dependencies. This means that we need new definitions and new algorithms; for example, the simplification algorithm in Chapter 3 needs to be more general than the corresponding algorithm in [Heintze and Jaffar, 1990] (even though they are very similar). On the other hand, the shallow SSC's we use are not so general that they can be considered as a mere way of rewriting a sort theory using a different syntax; since only limited inter-variable dependencies can be expressed, unification under these constraints is decidable.

Heintze and Jaffar [1990] present an algorithm that computes an approximation of the least model of a logic program, based on the idea of ignoring inter-variable dependencies. This approximation is represented as a term grammar (see Section 2.6) and is exact for a class of logic programs that includes the relativization of any linear sorted signature (see Chapter 5): For each unary predicate appearing in the logic program, a non-terminal in the grammar generates the set of ground terms for which the predicate is true in the program's least Herbrand model; this set is always a regular tree language when the approximation is exact. Much of the notation in this chapter is in fact borrowed from Heintze and Jaffar's [1990] paper; we will be careful to point out those cases where definitions are different. Note that we do *not* present an alternate algorithm for representing logic programs in general; the algorithm in Chapter 6 works only for a particular class of logic programs, for which the translation is rather straightforward. In the case of arbitrary sort *theories*, we refer to Heintze and Jaffar's [1990] approximation algorithm, if the theory can be expressed as a logic program; when the approximation is exact, sorted unification under the given theory can be carried out using the algorithms presented in this thesis.[1]

This chapter is organized as follows: Section 2.1 contains all the basic syntactic definitions. Section 2.2 defines *interpretations* and *models* of systems of set constraints. Section 2.3 defines *cycles* and *derivations* in SSC's, which are then used in Section 2.4 to prove that under some simple conditions, SSC's always have a unique model. Section 2.5 defines the class of shallow constraints that are used throughout the rest of this thesis, and a solved form for them. Finally, Section 2.6 describes the relationship between SSC's and *term grammars* (also known as *regular tree grammars*), showing that the latter are equivalent to a subclass of the shallow SSC's we use. Bogaert and Tison's [1991] *tree automata with equality tests*, which are also closely related to SSC's, are briefly discussed in this section.

2.1 Definitions

We are given a finite ranked set $\mathcal{F}$ of function symbols and a countably infinite set of variables $\mathcal{V}$. We assume that $\mathcal{F}$ includes at least one constant, so that the Herbrand Universe is not empty. The terms constructed over $\mathcal{F} \cup \mathcal{V}$ make up the main language, and SSC's will be used to define and express relationships between sets of terms in this language. For this, we use a countably infinite set of variables $\mathcal{V}_\mathcal{S}$, disjoint from $\mathcal{V}$, and introduce four new special symbols: "$\top$", "$\bot$", "$\sqcup$" and "$\sqcap$". The building blocks for the new constraints are *set expressions*, constructed from these four symbols and terms over $\mathcal{F}$ and $\mathcal{V}_\mathcal{S}$, as follows:

Definition 2.1 (Set Expressions) *A* set expression *is*
(i) a term over $\mathcal{F} \cup \mathcal{V}_\mathcal{S}$*,*
(ii) one of the special symbols "$\top$*" and "*$\bot$*", or*
(iii) $(t \sqcup s)$ *or* $(t \sqcap s)$*, where* s *and* t *are set expressions.*

As we will see, "$\top$" stands for the set of all terms over $\mathcal{F} \cup \mathcal{V}$, and "$\bot$" stands for the empty set.[2] The variables in $\mathcal{V}_\mathcal{S}$ will denote *sets* of terms over $\mathcal{F} \cup \mathcal{V}$, so we call them *set variables*. We will usually write set variables using uppercase letters, as in X, Y and Z. The symbols "$\sqcup$" and "$\sqcap$" will be used as the idempotent, commutative and associative operators they represent ($\cup$ and $\cap$, respectively). For example, "$(e_1 \sqcap (e_2 \sqcap e_3))$" also stands for "$((e_3 \sqcap e_1) \sqcap e_2)$". Accordingly, parenthesis will be dropped whenever possible, so that a set expression of the form $(e_1 \sqcap (e_2 \sqcap \ldots (e_{n-1} \sqcap e_n) \ldots))$ can be written as $e_1 \sqcap \ldots \sqcap e_n$.

The program approximation algorithm in [Heintze and Jaffar, 1990] is described using a more general syntax for set expressions; [Heintze and Jaffar, 1991] presents a decision procedure that finds solutions to collections of subset relationships between these general set expressions. We will not need this more flexible syntax, so the above definition covers only a subset of the set expressions studied by Heintze and Jaffar [1990; 1991]. We do *not* allow "$\top$", "$\bot$", or any set expression that contains "$\sqcap$" or "$\sqcup$" to be nested inside a function symbol. For example,

[1] Actually, simpler versions of the algorithms presented here could be used, since the resulting representation would not express any inter-variable dependencies.

[2] In the literature, these symbols are often used to denote the top and bottom sort, respectively. They have the corresponding meaning here.

$f((x \sqcap y), (a \sqcup b))$ and $g(\top)$ are *not* considered to be valid set expressions, and we will not provide semantics for them.[3]

Definition 2.2 (Set Equations and Set Constraints) *A* set equation *is a pair of set expressions, $\{E_1, E_2\}$, written as $E_1 \equiv E_2$. A* set constraint *is an* ordered *pair $\langle X, E\rangle$, where X is a variable in $\mathcal{V}_S$ and E is a set expression. The variable X is said to be* constrained *by E.*

Set constraints will be written as set equations between brackets, as in $\langle X \equiv (Y \sqcap Z) \sqcup f(a)\rangle$.[4] The *systems of set constraints* (or SSC's, for short) that are the main subject of the rest of this chapter are collections of set constraints that define a function on the constrained variables, as follows:

Definition 2.3 (Systems of Set Constraints (SSC's)) *A* system of set constraints *is a finite set C of set constraints, $\{\langle Y_1 \equiv E_1\rangle, \ldots, \langle Y_n \equiv E_n\rangle\}$, such that $Y_i \neq Y_j$ for all $i \neq j$. The set of variables $\{Y_1, \ldots, Y_n\}$ is the* domain *of C, denoted by $Dom(C)$, and the set of set expressions $\{E_1, \ldots, E_n\}$ is the* range *of C, denoted by $Rng(C)$. The set expression E_i is therefore uniquely determined for every $Y \in Dom(C)$, and is denoted by $C(Y)$.*

Not all the variables appearing in $Rng(C)$ have to appear in $Dom(C)$ (and vice-versa):

Definition 2.4 (Parameters) *Given an* SSC *C, $Var(C) - Dom(C)$ is the set of* parameters *of C, denoted by $Par(C)$. The parameters of C are also referred to as the* unconstrained *variables in C, and we say that the variables in $Dom(C)$ are* constrained *by C.*

An interesting subclass of SSC's are the *linear* ones:

Definition 2.5 (Linear Set Expressions and SSC's) *A set expression E is* linear *if every term t appearing in E is linear (that is, t contains no repeated variables). A set constraint $\langle X \equiv E\rangle$ is* linear *if E is a linear set expression. An* SSC *C is* linear *if $C(X)$ is linear for all $X \in Dom(C)$.*

Note that the same variable can appear more than once in a linear set expression — it is just that it cannot appear twice in any *term*. For example, $X \sqcup (f(X, Y) \sqcap Y)$ is linear, but $f(X, X) \sqcup Y$ is not.

Example 2.6 *Consider the following system of set constraints:*

$$C = \left\{ \begin{array}{ll} \langle W \equiv X \sqcup Y \sqcup Z\rangle, & \langle X \equiv f(g(X), Y) \sqcup a \sqcup b\rangle, \\ \langle Y \equiv g(X) \sqcup P\rangle, & \langle Z \equiv (h(X, Y) \sqcap h(Z, Z)) \sqcup a\rangle \end{array} \right\}$$

$Dom(C) = \{W, X, Y, Z\}$, and $Par(C) = \{P\}$. This is not a linear SSC*, since $C(X)$ contains the non-linear term $h(Z, Z)$.*

[3] However, as is made clear in Section 2.6, the *systems of set constraints* considered in this thesis can be used to represent a larger class of sets than the *definite set constraints* used in the above mentioned works.

[4] Again, this definition is syntactically less general than the corresponding one in [Heintze and Jaffar, 1990]; it corresponds to the *normal form* set constraints defined there.

We now present some more definitions that will be useful when referring to set expressions and SSC's: A set expression is an *atom*, or *atomic*, if it is "$\top$", "$\bot$", or a term (over $\mathcal{F} \cup \mathcal{V_S}$). That is, an atom is a set expression that does not contain "$\sqcap$" or "$\sqcup$". A set expression of the form $e_1 \sqcup \ldots \sqcup e_n$ is a *disjunction*, and the set expressions e_i are its *disjuncts*. A set expressions of the form $e_1 \sqcap \ldots \sqcap e_n$ is a *conjunction*, and the set expressions e_i are its *conjuncts*. A set expression is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of atomic set expressions.[5] We say that a system of set constraints C is in *disjunctive normal form* if all the elements of $Rng(C)$ are set expressions in disjunctive normal form.

An occurrence of a term t is said to be *nested* in a set expression if it is encapsuled by a function symbol (in $\mathcal{F}$) (that is, it is a proper subterm of another term), and is said to be *non-nested* otherwise. For example, if $exp = f(g(X), Z) \sqcup (Y \sqcap W)$, then X and Z are nested in exp, while Y and W are not.

The function $DNF(exp)$, which rewrites a set expression into an equivalent one in disjunctive normal form, is recursively defined in the usual way:

1. $DNF(e) = e$ if e is an atomic set expression.

2. $DNF(e_1 \sqcup e_2) = DNF(e_1) \sqcup DNF(e_2)$.

3. $DNF((e_1 \sqcup \ldots \sqcup e_n) \sqcap (s_1 \sqcup \ldots \sqcup s_m))$
   $= DNF(e_1 \sqcap s_1) \sqcup \ldots \sqcup DNF(e_1 \sqcap s_m) \sqcup \ldots \sqcup DNF(e_n \sqcap s_1) \sqcup \ldots \sqcup DNF(e_n \sqcap s_m)$.

Note that $DNF(exp)$ does not change the atoms in exp; therefore, an atom has a nested occurrence in exp iff it has a nested occurrence in $DNF(exp)$, and exp is linear iff $DNF(exp)$ is linear too.

2.2 Interpretations and Models

Systems of set constraints are used to represent sets of terms in the main language (i.e. terms over $\mathcal{F} \cup \mathcal{V}$). These sets are defined through *interpretations*, which are solutions to set constraints.

Definition 2.7 (Interpretations) *An* interpretation I *defined on a finite set of variables* $V \subset \mathcal{V_S}$ *is a mapping from each variable in* V *into a set of terms over* $\mathcal{F} \cup \mathcal{V}$.[6] *This mapping is extended to set expressions as follows:*

1. $I(\top)$ *is equal to the set of all ground terms over* $\mathcal{F}$, *that is, the Herbrand Universe.*
   $I(\bot)$ *is the empty set,* $\emptyset$.

2. *If* X *is a variable in* V, *then* $I(X)$ *is given by the original interpretation.*
   If $X \in \mathcal{V_S} - V$, *then* $I(X) \stackrel{\text{def}}{=} \emptyset$.

3. $I(e_1 \sqcap e_2) \stackrel{\text{def}}{=} I(e_1) \cap I(e_2)$;

4. $I(e_1 \sqcup e_2) \stackrel{\text{def}}{=} I(e_1) \cup I(e_2)$;

[5] That is, if it is of the form $(exp_1 \sqcup \ldots \sqcup exp_n)$, and each exp_i is of the form $(a_1 \sqcap \ldots \sqcap a_m)$ for atomic set expressions $a_1, \ldots, a_m$.

[6] These are sometimes called *set-interpretations*, to emphasize the fact that $I(X)$ is a set.

5. *If t is a non-variable term over $\mathcal{F} \cup \mathcal{V}_\mathcal{S}$, then*

$$I(t) \stackrel{\text{def}}{=} \{t\theta \mid X\theta \in I(X) \text{ for all } X \in Var(t)\}\,.$$

That is, $I(t)$ contains all terms that can be obtained by uniformly replacing all the occurrences of each variable X in t by the same element of $I(X)$.

The last condition ensures that equal variables within $f(t_1, \ldots, t_n)$ correspond to the same term, and is the main difference between these constraints and those in [Heintze and Jaffar, 1990]. Thus, all the elements of $I(t)$ are instances of t. If t is a ground term, then $I(t) = \{t\}$ for all interpretations I. Note that we allow $I(X) = \emptyset$, and that $I(X) = \emptyset$ for all variables $X \in \mathcal{V}_\mathcal{S}$ for which I is not initially defined. It is simple to show that for any set expression exp and any interpretation I, $I(exp) = I(DNF(exp))$.

- Two interpretations I and I' are **equivalent** over a set of variables $V \subseteq \mathcal{V}_\mathcal{S}$, written as $I =_V I'$, if $I(X) = I'(X)$ for all $X \in V$.
- We define a partial order $\leq$ on interpretations as follows:
  $I_1 \leq I_2$ if for all $X \in \mathcal{V}_\mathcal{S}$, $I_1(X) \subseteq I_2(X)$.

Definition 2.8 (Models) *An interpretation I* satisfies *a set equation $E_1 \equiv E_2$ if $I(E_1) = I(E_2)$. Given an* SSC *C, an interpretation I is a* model *of C if I satisfies $X \equiv C(X)$ for all $X \in Dom(C)$.*

- Two SSC's C and C' are **equivalent** over a set of variables V if for any model I of C, there exists a model I' of C' such that $I =_V I'$, and for any model I' of C', there exists a model I of C such that $I' =_V I$. In this case, we sometimes use the notation $C \stackrel{V}{\Longleftrightarrow} C'$.
- Given an SSC C and a set equation $E_1 \equiv E_2$, we say that C **entails** $E_1 \equiv E_2$, written as $C \models E_1 \equiv E_2$, if $I(E_1) = I(E_2)$ for any model I of C.

All the SSC's we will consider are *satisfiable* in the sense that they always have a model, as we show in Section 2.4. We will usually assume that a model I of C is only defined on $Var(C)$, so that $I(V) = \emptyset$ if $V \in \mathcal{V}_\mathcal{S} - Var(C)$.

Example 2.9 *In general, parameter-free* SSC*'s may have many models that are not equivalent over the constrained variables: Consider:*

$$C = \{\langle X \equiv X \sqcap (a \sqcup b)\rangle\}$$

There are 4 different models of C: $I(X) = \emptyset$, $I(X) = \{a\}$, $I(X) = \{b\}$, and $I(X) = \{a, b\}$.

2.3 Cycles and Derivations

We now present a sufficient condition under which an SSC C has a unique model, given a fixed interpretation for the parameters of C. For this, we need the following definitions:

Definition 2.10 (Cycles) *A* cycle *in a system of set constraints C is a sequence of variables $\langle X_1, \ldots, X_n\rangle$, $n \geq 1$, such that $X_1 = X_n$ and for each i, $1 \leq i < n$, $C(X_i)$ contains an un-nested occurrence of X_{i+1}.*

Intuitively, a cycle in C means that a variable is defined directly in terms of itself, with no mediating function symbol; the SSC $\{\langle X \equiv X \sqcap (a \sqcup b)\rangle\}$ used in Example 2.9 has the simplest possible kind of cycle. Note that we do *not* define $\langle X \equiv f(X)\rangle$ to form a cycle. If C' is obtained from C by rewriting any set expression in $Rng(C)$ into DNF, then C has a cycle iff C' has a cycle.

Definition 2.11 (Parameter Assignments) *Given an* SSC *C, a* parameter assignment *is an interpretation A that maps each parameter $P \in Par(C)$ into a set of terms over $\mathcal{F} \cup \mathcal{V}$. The parameter assignment A such that $A(P) = \emptyset$ for all $P \in Par(C)$ is always denoted by $A_\emptyset$.*

We formalize the sets of terms represented by the variables in an SSC using a sequent-like calculus and a set of inference rules, as follows: A *sequent* is an ordered pair of set expressions, E_1 and E_2, written as $E_1 \Rightarrow E_2$. The following are the inference rules for sequents, where $E, E_1, \ldots, E_n, F$ and G are arbitrary set expressions:

- **Rule 1**: From $E \Rightarrow F$ infer $(E \sqcup G) \Rightarrow F$.
- **Rule 2**: From $E_1 \Rightarrow F, \ldots, E_n \Rightarrow F$, infer $(E_1 \sqcap \ldots \sqcap E_n) \Rightarrow F$.
- **Rule 3**: From $X_1 \Rightarrow t_1, \ldots, X_n \Rightarrow t_n$ and $E \Rightarrow t$ for a non-variable term t over $\mathcal{F} \cup \mathcal{V}_\mathcal{S}$, where $Var(t) = \{X_1, \ldots, X_n\}$ and $t_1, \ldots, t_n$ are terms over $\mathcal{F} \cup \mathcal{V}$, infer $E \Rightarrow t\theta$, where $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$.
- **Rule 4**: From $E \Rightarrow F$ and $F \Rightarrow G$, infer $E \Rightarrow G$, where $E \neq F$ and $F \neq G$.

Given an SSC C, $\mathcal{A}(C)$ denotes the set of axioms $\{X \Rightarrow C(X) \mid X \in Dom(C)\}$ together with the infinite set of axioms of the form $\top \Rightarrow t$ where t is any term over $\mathcal{F}$. Given a parameter assignment A, $\mathcal{A}(A)$ denotes the (possibly infinite) set of axioms $\{X \Rightarrow t | A(X) = t\}$.

Definition 2.12 (Derivations) *Given a set of axioms $\mathcal{A}$ and a sequent S_0, a* derivation tree *for S from $\mathcal{A}$ is a finite directed tree whose nodes are sequents, whose root is S, whose leaves are all elements of $\mathcal{A}$, and where the children of a node S are the set of sequents from which S can be derived using one of the above rules.*

Given a parameter assignment A defined on $Par(C)$, we say that there is a derivation from E of F given C and A, *written as $E \overset{C,A}{\vdash\!\leadsto} F$, if there is a derivation tree for $E \Rightarrow F$ from $\mathcal{A}(C) \cup \mathcal{A}(A)$.*

Clearly, $E \overset{C,A_\emptyset}{\vdash\!\leadsto} F$ iff there is a derivation tree for $E \Rightarrow F$ from $\mathcal{A}(C)$. In this case, we write $E \overset{C}{\vdash\!\leadsto} F$. The *depth* of a derivation is the depth of the tree, defined as usual: If the tree has only one node, the depth is 0; otherwise, the depth of the tree is equal to one more than the maximum depth of its subtrees. When we refer to derivation trees, we will mean those of *minimum* depth. Note that if $E_1 \Rightarrow E_2$ is a node in a derivation tree from $\mathcal{A}(C) \cup \mathcal{A}(A)$, then $E_1 \overset{C,A}{\vdash\!\leadsto} E_2$, and that the $\overset{C,A}{\vdash\!\leadsto}$ relation is transitive (due to rule 4).

The following lemma provides a recursive definition of $\overset{C,A}{\vdash\!\leadsto}$:

Lemma 2.13 *Let C be an* SSC, *let A be a parameter assignment for C, and let E and F be set expressions. Then $E \overset{C,A}{\vdash\!\leadsto} F$ if, and only if, one of the following holds:*

1. *F is a term over $\mathcal{F} \cup \mathcal{V}$ and $E = F$, $E = \top$, or E is a parameter Y such that $t \in A(Y)$.*

2. *$E \in Dom(C)$ and $C(E) \overset{C,A}{\rightsquigarrow} F$.*

3. *$E = e_1 \sqcup \ldots \sqcup e_m$, $m > 1$, and for some e_i, $1 \leq i \leq m$, $e_i \overset{C,A}{\rightsquigarrow} F$.*

4. *$E = e_1 \sqcap \ldots \sqcap e_m$, $m > 1$, and for each e_i, $1 \leq i \leq m$, $e_i \overset{C,A}{\rightsquigarrow} F$.*

5. *$E = f(s_1, \ldots, s_m)$ is a term over $\mathcal{F} \cup \mathcal{V}_\mathcal{S}$, $F = f(t_1, \ldots, t_m)$ is a term over $\mathcal{F} \cup \mathcal{V}$, and there is a substitution θ such that $E\theta = t$ and for all $X_i \in Var(E)$, $X_i \overset{C,A}{\rightsquigarrow} X_i\theta$.*

Proof: "If:" Consider each case: If case 1 holds, then $E \Rightarrow F$ is an axiom in $\mathcal{A}(C) \cup \mathcal{A}(A)$. If case 2 holds, then $E \Rightarrow C(E)$ is an axiom in $\mathcal{A}(C)$, so inference rule 4 can be applied. Cases 3, 4 and 5 correspond to inference rules 1, 2 and 3, respectively.

"Only if:" We show that if $E \overset{C,A}{\rightsquigarrow} F$, then one of the above cases holds, by induction on the depth of a derivation tree for $E \Rightarrow F$:
<u>Basis</u>: The only node in the tree is an axiom in $\mathcal{A}(C) \cup \mathcal{A}(A)$. Then case 1 or case 2 must hold.
<u>Induction Step</u>: Consider the inference rule used to obtain the root node of the derivation tree for $E \Rightarrow F$. It is clear that if one of inference rules 1, 2 or 3 was applied, then one of cases 3, 4 or 5 would hold, respectively. Consider now rule 4: $E \Rightarrow E_1$ and $E_1 \Rightarrow F$ for some $E_1 \neq t$. But then, by the induction hypothesis, one of cases 1–4 has to apply for $E \overset{C,A}{\rightsquigarrow} E_1$ and $E_1 \overset{C,A}{\rightsquigarrow} F$. Cases 1 and 5 would mean that E_1 is a term over $\mathcal{F} \cup \mathcal{V}$, and then the only way that $E_1 \Rightarrow F$ could be derived would be if $E_1 = F$. Since $E_1 \neq F$, then the case that applies to $E \overset{C,A}{\rightsquigarrow} E_1$ has to be one of 2, 3 or 4. But then, by the transitivity of $\overset{C,A}{\rightsquigarrow}$, the same case will also apply for E and F. ∎

Example 2.14 *In the* SSC *C from Example 2.6, if $A(P) = \{c\}$, then $Y \overset{C,A}{\rightsquigarrow} c$, since $Y \Rightarrow (g(X) \sqcup P)$ is an axiom in $\mathcal{A}(C)$, so that $Y \overset{C}{\rightsquigarrow} P$, and $\mathcal{A}(A) = \{P \Rightarrow c\}$. Since $X \overset{C}{\rightsquigarrow} a$ and $X \overset{C}{\rightsquigarrow} f(g(a), Y)$, then $X \overset{C,A}{\rightsquigarrow} f(g(a), c)$.*

2.4 The Least Model

The definition of an interpretation allows $I(Y)$ to be an arbitrary set of terms over $\mathcal{F} \cup \mathcal{V}$ if Y is a parameter in C; this is why the above definition of derivations is dependent on a parameter assignment A. We now show that if we fix the interpretation of all the parameters of C, then an SSC with no cycles has a unique model defined over $Var(C)$.

Definition 2.15 (Extensions) *Given an interpretation A defined on the parameters of an* SSC *C, we say that I is a model of C that* extends *A, if I is a model of C, $I =_{Par(C)} A$, and $I(X) = \emptyset$ for all variables $X \in \mathcal{V}_\mathcal{S} - Var(C)$. When C is understood, we say that I is an* extension *of A.*

The condition that I map all the set variables not in C into $\emptyset$ is included simply because we are not interested in how the models of C treat these variables.

The following lemma is the first step in showing that the least model of an SSC C exists and is characterized by the derivations defined in the previous section:

Lemma 2.16 *Let C be an SSC and let A be an interpretation defined on $Par(C)$. Assume that I is a model of C that extends A. If $E \overset{C,A}{\mapsto} F$ for set expressions E and F, then $I(F) \subseteq I(E)$.*

Proof: By induction on the depth d of a derivation tree for $E \Rightarrow F$ from $\mathcal{A}(C) \cup \mathcal{A}(A)$.
Basis: $d = 0$. Then $E \Rightarrow F$ has to be an axiom in $\mathcal{A}(C) \cup \mathcal{A}(A)$.

If $E \Rightarrow F \in \mathcal{A}(C)$, then either $E = F$, $E = \top$, or E is a variable such that $C(E) = F$. In the first two cases, clearly $I(F) \subseteq I(E)$ for all I. In the third case, $I(E) = I(F)$ since I is a model of C.

If $E \Rightarrow F \in \mathcal{A}(A)$, then E is a parameter in C and F is a term t such that $t \in A(E)$. Therefore, $I(F) = \{F\} \subseteq A(E)$ (note that $Var(F) \cap \mathcal{V}_{\mathcal{S}} = \emptyset$). Since $I(E) = A(E)$, $I(F) \subseteq I(E)$.
Induction Step: Assume that the theorem is true for all derivations of depth less than d and $E \overset{C,A}{\mapsto} T$ where the derivation tree for $E \Rightarrow t$ has depth d, for $d \geq 1$.

Consider now each of the inference rules 1–4 that can be used to obtain $E \Rightarrow F$:

- Rule 1: The single children node is $G \Rightarrow F$, where $E = G \sqcup exp$; then $G \overset{C,A}{\mapsto} F$ with a derivation of depth less than d, so by induction hypothesis, $I(F) \subseteq I(G)$; therefore, $I(F) \subseteq I(G) \cup I(exp) = I(G \sqcup exp) = I(E)$.

- Rule 2: $E = E_1 \sqcap \ldots \sqcap E_n$, where $E_1 \Rightarrow F, \ldots, E_i \Rightarrow F$ are the n children nodes, for $1 \leq i \leq n$. Then $E_i \overset{C,A}{\mapsto} F$ for each i, with a depth less than d, so by induction hypothesis, $I(F) \subseteq I(E_i)$ for each i, $1 \leq i \leq n$. Therefore, $I(F) \subseteq I(E_1) \cap \ldots \cap I(E_n) = I(E_1 \sqcap \ldots \sqcap E_n) = I(E)$.

- Rule 3: $F = t\theta$, where $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ and the children nodes are $X_1 \Rightarrow t_1, \ldots, X_n \Rightarrow t_n$ and $E \Rightarrow t$. Then $E \overset{C,A}{\mapsto} t$ and for all $X_i \in Dom(\theta)$ $X_i \overset{C,A}{\mapsto} X_i\theta$ with derivations of depth less than d, so by induction hypothesis, $I(t) \subseteq I(E)$ and $X_i\theta = I(X_i\theta) \subseteq I(X_i)$.

  $I(t) = \{t\Omega \mid X\Omega \in I(X) \text{ for all } X \in Dom(\Omega)\}$. Therefore, $t\theta \in I(t)$, so $I(F) = I(t\theta) = \{t\theta\} \subseteq I(t)$. Since $I(t) \subseteq I(E)$, then $I(F) \subseteq I(E)$.

- Rule 4: The children nodes are $E \Rightarrow G$ and $G \Rightarrow F$. Therefore, $E \overset{C,A}{\mapsto} G$ and $G \overset{C,A}{\mapsto} F$, each with a derivation of depth less than d. Then, by induction hypothesis, $I(F) \subseteq I(G)$ and $I(G) \subseteq I(E)$, so $I(F) \subseteq I(E)$. ■

Lemma 2.16 essentially establishes the "soundness" of derivations: It implies that if $X \overset{C,A}{\mapsto} t$ for a term t over $\mathcal{F} \cup \mathcal{V}$, then for any model I of C that extends A, $I(t) = \{t\} \subseteq I(X)$. The following lemma shows how derivations provide a *complete* characterization of models, once the parameter assignment A is fixed:

Lemma 2.17 *Let C be an* SSC *with no cycles, let X be in the domain of C, and let t be a term over $\mathcal{F} \cup \mathcal{V}$. If $t \in I(X)$ for a model I of C that extends A, then $X \overset{C,A}{\vdash\!\leadsto} t$.*

Proof: Let I be a model of C that extends A. Without loss of generality, we can assume that $C(V)$ is in DNF. We now show that if $t \in I(X)$, then $X \overset{C,A}{\vdash\!\leadsto} t$, for all $V \in Dom(C)$, by induction on the nesting depth of t.

I is a model of C, so $I(X) = I(C(X))$. Now, $C(X)$ is $e_1 \sqcup \ldots \sqcup e_m$, so $t \in I(e_j)$ for some $e = e_j$, $1 \leq j \leq m$. We now show that $e \overset{C,A}{\vdash\!\leadsto} t$, and thus $X \overset{C,A}{\vdash\!\leadsto} t$:

<u>Basis</u>: t is a constant or a variable not in $Dom(C)$. Consider the following cases:

1. Some conjunct in e is "$\perp$", a parameter Y where $t \notin A(Y)$, or of the form $f(s_1, \ldots, s_k)$ where $f \neq t$, $k \geq 0$. But then t cannot be in $I(e)$, a contradiction.

2. All conjuncts $c_1, \ldots, c_m$ in e_i are either "$\top$", a parameter Y such that $t \in A(Y)$, or variables in $Dom(C)$. In the first two cases, $c_j \Rightarrow t$ is an axiom in $\mathcal{A}(C) \cup \mathcal{A}(A)$, so $c_j \overset{C,A}{\vdash\!\leadsto} t$.

   Consider then the case where $c_j = V \in Dom(C)$: $V \neq X$, since C has no cycles, and $t \in I(V)$. Therefore, we can draw the same conclusions about $C(V)$ as were drawn about $C(X)$. We now can iterate this argument to obtain a sequence of variables $\langle V_0, V_1, \ldots \rangle$ such that $X = V_0$ and V_{i+1} occurs un-nested in $C(V_i)$. The sequence ends if we reach a V_n such that $V_n \overset{C,A}{\vdash\!\leadsto} t$, in which case $X \overset{C,A}{\vdash\!\leadsto} t$. Otherwise, eventually $V_i = V_j$ for some $i \neq j$, because $Dom(C)$ is finite. But then we have a cycle in C, a contradiction. Therefore, $X \overset{C,A}{\vdash\!\leadsto} t$.

   Since $C_i \overset{C,A}{\vdash\!\leadsto} t$ for all i, $1 \leq i \leq n$, then $X \overset{C,A}{\vdash\!\leadsto} T$.

<u>Induction Step</u>: Let $t = f(t_1, \ldots, t_n)$, and assume that for all i, $1 \leq i \leq n$, and any $Y \in Dom(C)$, if $t_i \in I(Y)$ then $Y \overset{C,A}{\vdash\!\leadsto} t_i$. We have the following cases:

1. If some conjunct in e_i is "$\perp$", a parameter Y where $t \notin A(Y)$, or of the form $g(s_1, \ldots, s_k)$ for $g \neq f$, $k \geq 0$, then t cannot be in $I(e_i)$, a contradiction.

2. All the conjuncts c_j in e_i are "$\top$", a parameter Y such that $t \in A(Y)$, of the form $f(s_1, \ldots, s_n)$, or variables $V \in Dom(C)$:

   In the first two cases, as before, $c_j \overset{C,A}{\vdash\!\leadsto} t$, since $c_j \Rightarrow t$ is an axiom in $\mathcal{A}(C) \cup \mathcal{A}(A)$. If $c_j = f(s_1, \ldots, s_n)$, then since $t \in I(e_i) = I(c_1) \cap \ldots \cap I(c_j)$, $t \in I(f(s_1, \ldots, s_n))$. This means that there is a substitution θ such that $t = f(s_1, \ldots, s_n)\theta$ and $X\theta \in I(X)$ for all $X \in Dom(\theta) = Var(f(s_1, \ldots, s_n))$. Since $X\theta$ is a subterm of t, then by induction hypothesis, $X \overset{C,A}{\vdash\!\leadsto} X\theta$, so $c_j \overset{C,A}{\vdash\!\leadsto} t$ (by the definition of derivations).

   The only remaining case is where $c_j = V \in Dom(C)$, $t \in I(V)$, and $V \neq X$. But now we can iterate the argument as in the basis step and conclude that either $X \overset{C,A}{\vdash\!\leadsto} t$ or there is a cycle in C. ∎

We are now ready to prove the following:

Theorem 2.18 *Let C be an* SSC *that contains no cycles. Let A be an interpretation defined on the parameters of C. Then C has a unique model that extends A.*

Proof: The proof is constructive. Again, w.l.o.g. we can assume that $C(X)$ is in DNF for all $X \in \mathit{Dom}(C)$. Define an interpretation M of C as follows:

$$M(X) = \left\{ t \mid X \overset{C,A}{\mapsto} t \text{ and } Var(t) \cap \mathcal{V}_{\mathcal{S}} = \emptyset \right\}$$

For all variables $V \in \mathcal{V}_{\mathcal{S}}$, $\mathit{Var}(A(Y)) \cap \mathit{Var}(C) = \emptyset$, since $\mathit{Var}(C) \subset \mathcal{V}_{\mathcal{S}}$, $I(Y)$ is a set of terms over $\mathcal{F} \cup \mathcal{V}$ and $\mathcal{V} \cap \mathcal{V}_{\mathcal{S}} = \emptyset$. Therefore, $M(Y) = A(Y)$ for all parameters Y of C, so M is an extension of A.

We now show that M is a model of C; that is, $M(X) = M(C(X))$ for all $X \in \mathit{Dom}(C)$. This is the case if for any term t, $t \in M(X)$ iff $t \in M(C(X))$. Note that if $t \in M(\mathit{exp})$ for any set expression exp, then t cannot contain variables in $\mathcal{V}_{\mathcal{S}}$.

(i): Assume that $t \in M(X)$. Then $X \overset{C,A}{\mapsto} t$. Since C is in DNF, $C(X) = e_1 \sqcup \ldots \sqcup e_m$, and for some e_j, $e_j \overset{C,A}{\mapsto} t$, $1 \leq j \leq m$. Then $e_j = c_1 \sqcap \ldots \sqcap c_k$, and for each c_i, $c_i \overset{C,A}{\mapsto} t$. From Lemma 2.13, each c_i has to be t, a parameter Y such that $t \in A(Y)$, "$\top$", a variable $Y \in \mathit{Dom}(C)$ or of the form $f(s_1, \ldots, s_n)$, where $t = f(t_1, \ldots, t_n)$, $n \geq 1$. Consider each case:

- If $c_i = t$, $c_i = \top$, or $c_i = Y$ where $t \in A(Y)$, then, by definition, $t \in M(c_i)$.
- If $c_i = Y \in \mathit{Dom}(C)$, then by the definition of M, $t \in M(Y) = M(c_i)$.
- In the last case, from Lemma 2.13, there has to be a substitution θ such that $t = f(s_1, \ldots, s_n)\theta$ and $X \overset{C,A}{\mapsto} X\theta$ for all $X \in \mathit{Dom}(\theta)$. Therefore, $X\theta \in M(X)$, by the definition of M. This means that $t \in M(f(s_1, \ldots, s_n))$.

Therefore, $t \in M(c_i)$ for all i, so $t \in M(e_j)$ implies $t \in M(C(X))$.

(ii): Assume that $t \in M(C(X))$. To show that $t \in M(X)$, we have to show that $X \overset{C,A}{\mapsto} t$. Again, $C(X) = e_1 \sqcup \ldots \sqcup e_m$ and $t \in M(e_j)$ for some j, where $e_j = c_1 \sqcap \ldots \sqcap c_k$ and $t \in M(c_i)$ for all i, $i \leq i \leq k$. The same cases hold:

- If $c_i = \top$, $c_i = t$, or c_i is a parameter Y such that $t \in A(Y)$. Then by definition $c_i \overset{C,A}{\mapsto} t$.
- If c_i is a variable, then if $t \in M(c_i)$, $c_i \overset{C,A}{\mapsto} t$, from the definition of M.
- Finally, if $c_i = f(s_1, \ldots, s_n)$, then $t \in M(c_i)$ means that there is a substitution θ such that $c_i\theta = t$ and $X\theta \in M(X)$ for all $X \in \mathit{Dom}(\theta)$. From the definition of M, $X \overset{C,A}{\mapsto} X\theta$, so $c_i \overset{C,A}{\mapsto} t$.

Therefore, $c_i \overset{C,A}{\mapsto} t$ for each c_i, so $e_j \overset{C,A}{\mapsto} t$, and thus $X \overset{C,A}{\mapsto} t$. From the definition of M, $t \in M(X)$.

Now that we have established that M is a model of C, we show that it is the *only* model of C that extends A: Let M' be a model of C that extends A. Let X be a variable in $\mathcal{V}_{\mathcal{S}}$. If $t \in M'(X)$, then, by Lemma 2.17, $X \overset{C,A}{\leadsto} t$, so $t \in M(X)$ by the definition of M. Thus, $M'(X) \subseteq M(X)$.

Now assume that there is a term t such that $t \in M(X)$. Then $X \overset{C,A}{\leadsto} t$. By Lemma 2.16, $M'(t) \subseteq M'(X)$. Since t cannot contain variables in $\mathcal{V}_{\mathcal{S}}$, then $M'(t) = \{t\}$, so $t \in M'(X)$. Thus, $M'(X) \supseteq M(X)$.

Therefore, $M(X) = M'(X)$ for all variables X, that is, M and M' are same model. ■

This means that an SSC C with no cycles has a unique model once a parameter assignment has been fixed. I is the **least model** of C iff I is a model of C and for every other model I' of C, $I(X) \subseteq I'(X)$ for all $X \in \mathcal{V}_{\mathcal{S}}$. It then follows that C has a unique least model:

Corollary 2.19 *An* SSC *C with no cycles has a unique least model, and if C has no parameters and no cycles, then the least model of C is the only model of C.*

Proof: By Theorem 2.18, the least model of C is the model that extends $A_{\emptyset}$, where $A_{\emptyset}$ maps each parameter in C into $\emptyset$. Since all models of C are extensions of some parameter assignment A, then if C has no parameters there can only be one model of C. ■

Definition 2.20 (Least Models, Constrained Terms, Instances) *Given an* SSC *C with no cycles, the least model of C is denoted by M_C. A term t over $\mathcal{F} \cup \mathcal{V}_{\mathcal{S}}$* constrained *by C, written as t/C, is said to stand for the set of ground terms $M_C(t)$. We sometimes write $M_C(t)$ as $[t/C]$. If $s \in [t/C]$, we say that s is an* instance *of t/C.*

All the SSC's we consider will have no cycles, so we do not have to worry about M_C being ill-defined. Note that if C_1 and C_2 are equivalent SSC's over V and t is a term containing only variables in V, then $[t/C_1] = [t/C_2]$.

Example 2.21 *Consider the following* SSC*:*

$$C = \left\{ \begin{array}{ll} \langle W \equiv a \sqcup b \sqcup P \rangle, & \langle X \equiv W \rangle, \\ \langle Y \equiv f(W, X) \rangle, & \langle Z \equiv f(X, X) \rangle \end{array} \right\}$$

P is a parameter in C, so that $M_C(P) = \emptyset$.

$$\begin{array}{l} M_C(W) = M_C(X) = \{a,\ b\} \\ M_C(Y) = \{\ f(a,a),\ f(a,b),\ f(b,a),\ f(b,b)\ \} \\ M_C(Z) = \{\ f(a,a),\ f(b,b)\ \} \end{array}$$

Note that derivations provide an effective procedure for enumerating the set $M_C(X)$ for $X \in Dom(C)$, by listing all the ground terms t such that there is a derivation of t from X of depth $1, 2, \ldots, n, n+1, \ldots$; the partial set at each stage is used to generate the one at the next.

Finally, we should note that the variables in the main language, $\mathcal{V}$, do not play any role in the above definition on M_C; $M_C(E)$ is always a set of ground terms for any set expression E. An interpretation I has to be the extension of a parameter assignment other than $A_{\emptyset}$ if a non-ground term t is to appear in $I(X)$ for $X \in \mathcal{V}_{\mathcal{S}}$. We will make use of non-empty parameter assignments in Chapter 6.

2.5 Shallow SSC's and Solved Form

We now focus on a particular class of systems of set constraints, the **shallow** ones, and define a **solved** form for them:

Definition 2.22 (Shallow) *A set expression exp is* shallow *iff every term appearing in exp is a variable, a constant, or of the form $f(V_1, \ldots, V_n)$ for an n-ary function f and variables $V_1, \ldots, V_n$. A system of set constraints C is* shallow *iff $C(X)$ is shallow for all $X \in Dom(C)$.*

That is, an SSC C is shallow if the only nested terms appearing in $Rng(C)$ are variables.[7] For instance, the SSC C in Example 2.6 is not shallow, since the term $f(g(X))$ appears in $Rng(C)$. Note that "$\sqcap$", "$\sqcup$", "$\top$" and "$\perp$" are all still allowed to appear in $Rng(C)$, though, as before, not within ordinary terms.

Unrestricted SSC's are a *very* general way of representing sets of terms; they are basically equivalent to arbitrary sorted signatures, as we will see in Chapter 6, and do not have the desirable computational properties that shallow ones do. The undecidability of sorted unification for general sorted signatures as proved by Schmidt-Schauß [1989] implies that the question of whether $[t_1/C] \cap [t_2/C]$ is empty is undecidable if C is allowed to be an arbitrary SSC's (with no cycles). This is why we will usually consider only *shallow* SSC's, and rarely any others; when we do consider other kinds of SSC's, they will turn out to be equivalent to shallow ones, as in Chapter 6.

We will usually assume that all the set expressions in $Rng(C)$ are in DNF for all SSC's C. In practice, conversion into DNF is an expensive process, and can be delayed in the simplification algorithms of Chapter 3 until strictly necessary.

Definition 2.23 (Solved Form) *A set constraint $\langle Y \equiv E \rangle$ is in* solved form *if E is "$\perp$", "$\top$", or $e_1 \sqcup \ldots \sqcup e_m$, $m \geq 1$, where $e_i \neq e_j$ for all $i \neq j$ and each e_i is a constant, a variable, or $f(X_1, \ldots, X_n)$, for variables $X_1, \ldots, X_n$, $n \geq 1$. A system of set constraints is in* solved form *if all its elements are solved.*

In other words, an SSC C is in solved form iff for all $X \in Dom(C)$, (i) C is shallow, that is, the only nested terms in $Rng(C)$ are variables; (ii) $C(X)$ contains no occurrences of "$\sqcap$", and (iii) if "$\top$" or "$\perp$" appear in $C(X)$, then either $C(X) = \top$ or $C(X) = \perp$.

This definition of solved form is similar to the "explicit form" for the collections of set equations in [Heintze and Jaffar, 1990]. We will see in the following section that, unlike that case, we cannot always construct a term grammar equivalent to an SSC C in solved form, because of the inter-variable dependencies that can be expressed in SSC's.

2.6 Term Grammars

Heintze and Jaffar's [1990] approximation for the least model of a logic program is represented as a **term grammar**, also known as a *regular tree grammar*. Term grammars are used by Mishra and Reddy [1985] and Aiken and Murphy [1991], among others, in the area of type specification for functional languages, and by McAllester [1983] to represent the equivalence

[7] Note that a constant a is considered to be nested in $f(a)$.

classes induced by sets of equations between ground terms. Heintze and Jaffar [1991] also use them to represent the solutions to collections of generalized set constraints.

See [Gècseg and Steinby, 1984] for a formal and comprehensive introduction to this area. This section only reviews some basic definitions and results that are useful when discussing systems of set constraints.

As Gècseg and Steinby point out, "*....being relatively new and very manyfaceted, the field [of tree automata] still lacks a uniform widely accepted formalism*" [1984, page 7]. This still appears to be true; for compatibility with [Heintze and Jaffar, 1990], we refer to regular tree grammars as "term grammars."

2.6.1 Definitions

Given a ranked set $\mathcal{F}$ of function symbols and a set of **nonterminals** $\mathcal{N}$, a **term grammar** is a finite set of productions of the form $N \rightarrow t$, where $N \in \mathcal{N}$ and t is a term built over $\mathcal{F} \cup \mathcal{N}$ (where nonterminals are 0-arity symbols). Given a grammar G, we write $t_1 \Rightarrow_G t_2$ if $N \rightarrow s$ is a production in G and t_2 can be obtained by replacing an occurrence of N in t_1 by s. We write $s \Rightarrow_{G*} t$ if there is a finite sequence of terms $\langle s = t_1, \ldots, t_n = t\rangle$ such that $t_i \Rightarrow_G t_{i+1}$ for all i, $1 \leq i < n$.

A **forest** or **language** is simply a set of terms, which are also called "trees." Given a distinguished non-terminal a_o in G, the **start symbol**, the *forest generated by* G is defined as the set of terms $\{t \mid a_o \Rightarrow_{G*} t\}$ where t contains no non-terminals, and will be denoted by $T(G)$. The class of languages generated by term grammars in this way is the same class that can be recognized by *frontier-to-root tree automata*, and is denoted by *Rec*.[8] The languages in *Rec* are called **recognizable forests**.

Two term grammars are **equivalent** if they define the same language. Clearly, a grammar defines a recognizable forest for each of its non-terminals when we select it as the start symbol. Thus, we say that two grammars are equivalent over a set of non-terminals S if for each non-terminal $N \in S$, the languages generated when using N as the start symbol in each grammar are the same.

Any term grammar can be rewritten to an equivalent one in **normal form**, where all the productions are of the form $N \rightarrow f(V_1, \ldots, V_n)$, where f is an n-ary function symbol and $V_i \in \mathcal{N}$ for all i, $1 \leq i \leq n$. Furthermore, a grammar G can be rewritten into a normal form grammar that is equivalent to G over all of the non-terminals in G.

There are many established results and algorithms concerning term grammars and recognizable forests (see [Gècseg and Steinby, 1984]). Some of these are:

- *Rec* is closed under union, intersection and set difference (and thus is also closed under complementation), and there are algorithms for computing the corresponding normal form grammars.

- If G_1 and G_2 are term grammars, the following questions are decidable:

[8] Frontier-to-root automata are sometimes called "bottom-up" automata. This can be misleading, since trees are not always drawn upside down. In any case, these automata start at the *leaves* and make their way to the *root* of the tree, and both the deterministic and non-deterministic versions have the power to describe the languages in *Rec*. In contrast, only *non-deterministic* root-to-frontier ("top-down") tree automaton can describe all the languages in *Rec*; the deterministic ones define a proper subset of *Rec*.

- The **finiteness problem**: Is $T(G_1)$ finite?
- The **membership problem**: Is a given t in $T(G_1)$?
- The **emptyness problem**: Is $T(G_1)$ empty?
- The **equivalence problem**: Is $T(G_1) = T(G_2)$?
- The **inclusion problem**: Is $T(G_1) \subseteq T(G_2)$?

Clearly, the decidability of the inclusion problem implies the decidability of the membership, emptyness and equivalence problems.

2.6.2 Term Grammars vs. Systems of Set Constraints

It appears that Comon [1989] was the first to point out that what are usually called "*order-sorted signatures*" are equivalent to finite (frontier-to-root) tree automata that define a regular tree language corresponding to the ground terms of each sort. In these signatures, all declarations are either subsort declarations or of the form $S(f(x_1{:}S_1, \ldots, x_n{:}S_n))$, where $x_i \neq x_j$ for $i \neq j$ (these declarations are usually written as $f(S_1 \times \ldots \times S_n) : S$).

Systems of set constraints also define sets of terms; given an SSC C with no cycles and a term t, $M_C(t)$ is a forest. The variables $X \in Dom(C)$ serve a purpose similar to that of the non-terminals in a term grammar G. In the following discussion, the set of non-terminals $\mathcal{N}$ will be a subset of $\mathcal{V}_\mathcal{S} \cup \{\top\}$.

Given the set of function symbols $\mathcal{F}$, we say that an SSC C and a term grammar G with non-terminal set $\mathcal{N}$ are *equivalent* over a set $S \subseteq Dom(C) \cap \mathcal{N}$ if for all $X \in S$, $t \in M_C(X)$ iff $X \Rightarrow_{G*} t$ for a ground term t over $\mathcal{F}$.

Every *linear* shallow SSC C in solved form has an equivalent term grammar G over $Dom(C) \cup \{\top\}$; G can be obtained from C as follows: For each constraint $\langle X \equiv e_1 \sqcup \ldots \sqcup e_n \rangle$, add to G the productions $\{X \to e_1, \ldots, X \to e_n\}$. If "$\top$" occurs in $Rng(C)$, then add to G the production $\top \to f(\top, \ldots, \top)$ for each function $f \in \mathcal{F}$ (including $\top \to c$ for all constants c). That C and G are equivalent in the above sense can be proved easily but tediously using induction on the nesting depth of t for the "only if" direction, and induction on the depth of the derivation of t from G for the "if" direction.

Any linear SSC C can be rewritten into an equivalent linear shallow SSC, by replacing each non-variable subterm t in $Rng(C)$ by a new variable N and adding the constraint $\langle N \equiv t \rangle$, until all the subterms in $Rng(C)$ are variables. (This is done, for example, by the algorithm in Chapter 6, which "translates" a class of sorted signatures with term declarations into equivalent shallow SSC's.) The simplification algorithm in Chapter 3 shows that every shallow SSC can be rewritten into an equivalent one in solved form. Since these two transformations preserve linearity, linear SSC's in general have equivalent term grammars.

Also, every term grammar G with non-terminal set $\mathcal{N}$ has an equivalent linear SSC C over $\mathcal{N}$. Given G, we can obtain C in three steps:

1. Rewrite G into an equivalent normal form grammar G' with non-terminal set $\mathcal{N}' \supseteq \mathcal{N}$.

2. Rewrite G' into an equivalent grammar G'' over $\mathcal{N}'$, where for every production $N \to t$, the term t does not contain repeated non-terminals: For each production $N \to t$ in G', replace each occurrence of a non-terminal A in t by a distinct new non-terminal, A_i, and add the production $A_i \to A$. Clearly, all productions $N \to t$ in the new G'' are such

that t is either a non-terminal, a constant, or $f(A_1, \ldots, A_n)$ for non-terminals A_i, where $A_i \neq A_j$ if $i \neq j$.

3. Construct from G'' an SSC C with domain $\mathcal{N}'$ that contains the set constraint $\langle N \equiv t_1 \sqcup \ldots \sqcup t_n \rangle$ for each non-terminal $N \in \mathcal{N}$, where $\{N \to t_1, \ldots, N \to t_n\}$ are the productions for N in G''. It is easy to see that C is a linear SSC, equivalent to G'' over the non-terminals of G'', and hence equivalent to G over $\mathcal{N}$.

Even though the syntax of the *set expressions* used by Heintze and Jaffar [1991; 1990] is more flexible (allowing, for example, nested intersection and union operators, as well as *projection functions*) and the *definite set constraints* in [Heintze and Jaffar, 1991] allow for subset relations, the languages that can be defined using those constraints are still in *Rec*.[9]

Example 2.24 *The following is a linear* SSC *with no cycles:*

$$C = \Big\{ \ \langle W \equiv f(X, h(Y)) \sqcup a \rangle, \ \langle X \equiv h(X) \sqcup b \rangle, \ \langle Y \equiv W \sqcup c \rangle \ \Big\}$$

C can be rewritten into a solved form SSC *by replacing $\langle W \equiv f(X, h(Y)) \sqcup a \rangle$ by $\langle W \equiv f(X, N) \rangle$ and adding the new set constraint $\langle N \equiv h(Y) \rangle$. Similarly, a term grammar in normal form equivalent to C is:*

$$\begin{array}{ll} W \to f(X, N), & W \to a, \\ N \to h(Y), & \\ X \to h(X), & X \to b, \\ Y \to W, & Y \to c \end{array}$$

2.6.3 Non-regular Tree Languages

Since a constrained term t/C generates the tree language $[t/C]$, different classes of languages are defined by different restrictions on t and C. We now consider some of these classes together with the restrictions that characterize them:

The definition of a derivation in a term grammar allows replacing the same non-terminal in a term t by *different* subterms; there is no way of ensuring that they be mapped to equal subterms. However, SSC's do allow for this. For example, Gècseg and Steinby [1984] show that the set of terms

$$S = \{t_1 = a, \ t_2 = f(a, a), \ t_3 = f(f(a, a), f(a, a)), \ \ldots, \ t_n = f(t_{n-1}, t_{n-1}), \ \ldots \}$$

is not in *Rec*. However, $S = [X/C]$, where $C = \{\langle X \equiv f(X, X) \sqcup a \rangle\}$.

We can prove that $S \notin Rec$ by showing that S cannot be represented as $[Y/C']$ for some linear C' and a variable Y: Clearly, there has to be at least one derivation from Y of a term $f(Z, W)$, where $[Z/C']$ and $[W/C']$ both contain more than one element; otherwise, $[Y/C']$ would have to be finite or equal to $[\top/C']$, the Herbrand Universe. Since C' is linear, $Z \neq W$. Therefore, there exists a term $f(t_1, t_2)$ in $[Y/C']$ such that $t_1 \neq t_2$, a contradiction.

Similarly, we can show that S cannot be represented using an arbitrary term and a linear SSC: Since all the terms in $[X/C]$ are of the form $f(t, t)$ where $t \in [X/C]$, the set of all terms t

[9] On the other hand, Heintze and Jaffar answer more questions about these definite set constraints than those we answer here for SSC's.

such that $f(t,t) \in [X/C]$ is equal to $[X/C]$ itself. Therefore, if $[X/C] = [f(t_1,t_2)/C']$ for some C', then $[t_1/C'] = [t_2/C'] = [X/C]$. By induction, it follows that $[X/C]$ cannot be represented as $[f(t_1,t_2)/C']$ for any finite t_1, t_2 and linear C'.

We can also show that not all the languages representable using an arbitrary term and a linear SSC are in *Rec*:

If $C = \{\langle X \equiv f(X) \sqcup a\rangle\}$, then the set

$$\begin{aligned}[h(X,g(X))/C] &= \{h(x,g(x))\theta \mid x\theta \in [X/C]\} \\ &= \{h(a,g(a)),\ h(f(a),g(f(a))),\ \ldots,\ h(f^n(a),g(f^n(a))),\ldots\}\end{aligned}$$

cannot be represented as $[Y/C']$ for a variable Y and a linear SSC C'. The proof is similar to the above.[10] Therefore, we have (informally) established the following:

Proposition 2.25 *The class Rec of recognizable forests is equal to the class of forests representable as $[X/C]$ for a variable X and a linear* SSC *C. This is a proper subclass of forests representable using arbitrary terms and linear* SSC*'s, which in turn is a proper subclass of the forests representable using arbitrary terms and shallow* SSC*'s.*

We already pointed out in Section 2.5 that since sorted unification under general sorted signatures with term declarations is undecidable, then deciding whether the intersection of $[t_1/C]$ and $[t_2/C]$ is empty is also undecidable for an arbitrary SSC C. This also implies that the emptyness of $M_C(X)$ for a general SSC C is undecidable.

Schmidt-Schauß [1989] shows that linear unification problems (those where no variable occurs twice in the terms to be unified) for linear signatures are decidable, but left open the problem of unification for general terms under linear signatures.[11] The unification algorithm of Chapter 4 answers this question and generalizes the result by computing the intersection of two (possibly non-regular) tree languages $[t_1/C_1]$ and $[t_2/C_2]$, for shallow SSC's C_1 and C_2. Furthermore, the algorithm shows that this intersection can be covered by a finite set of languages also representable using shallow SSC's. In the next chapter, we will see that the emptyness problem is decidable for these languages, by rewriting shallow SSC's into solved form.

We will point out what the algorithms and results look like if the the SSC's are restricted to be linear — i.e. equivalent to term grammars. It turns out that the intersection of two languages in representable using linear SSC's can be covered using a finite set of languages *also* representable using linear SSC's. The unification algorithm in Chapter 4 can be used to obtain such a set. Therefore, the algorithms in Chapters 3, 4 and 6 can be specialized to consider only the case of linear SSC's and linear signatures, which can lead to increased efficiency.

Comon's [1990] simplification algorithm for equational formulas under membership constraints is presented in terms of tree automata, where sorts (and derived *sort expressions*) correspond to languages in *Rec*. The "sorted signatures" defined there are a restricted version of the sort theories we will consider in Chapters 5 and 6: All declarations are *function declarations*, of the form $f : S_1 \times \ldots \times S_N \to S$. The membership constraints are (basically) expressed in terms of tree automata. By including negation in the language, the problem of subsumption

[10] In general, *Rec* is closed under linear tree homomorphisms, but is not closed under arbitrary ones [Gècseg and Steinby, 1984].

[11] Sorts in linear signatures correspond to regular tree languages, so that unifying linear terms under linear signatures is equivalent to computing the intersection of two languages in *Rec* (given the right translation, as described in Chapter 6).

between constrained terms can be decided (tree automata algorithms are used to compute the complement of regular sets).

These results are extended by Comon and Delor [1991] by referring to *tree automata with equality tests* [Bogaert, 1990; Bogaert and Tison, 1991]. In these tree automata, transitions are conditioned by sets of equalities and inequalities imposed on "brother terms," which share exactly the same prefix. In the automata's transition rules, these correspond to state variables nested at depth 1 — just as variables are nested at depth 1 in shallow SSC's.

The set of tree languages $[X/C]$ for a variable X and a shallow SSC C corresponds to the case where only equations are allowed in these tree automata; this class is called $REC_=$ by Bogaert and Tison [1991], who point out that it is closed under union and intersection, but not complementation.[12] The larger class of tree languages where inequalities are allowed as conditions for each transition is denoted by $REC_{\neq}$. By using operations on the extended tree automata, Bogaert and Tison [1991] show that this class is closed under union, intersection and complementation, and that its emptyness problem is decidable.

Our constrained variables will range over languages in $REC_=$, so the unification algorithm presented here is an instance of the one entailed by Comon and Delor's [1991] simplification algorithm when the new tree automata are used. While that algorithm refers to the normalization and determinization algorithms for tree automata with equality tests, the unification algorithm presented in the next two chapters is based on transformations on set expressions and systems of set constraints.

It might be possible to similarly extend the idea of SSC's to enforce *differences* between subterms, with extended set constraints that would look like $\langle W \equiv (f(X,Y) \,|\, X \neq Y) \sqcup g(Y,Z) \rangle$. It is unclear if there is any further (relatively natural) extension to $REC_{\neq}$ that still preserves closure under union and intersection and the decidability of the emptiness problem. As Bogaert and Tison [1991] point out, the emptyness problem for their tree automata is undecidable if dependencies that include variables nested at depth 2 are allowed.

[12] This is also clear when one sees that any given set constraint in an SSC cannot enforce subterms to be *different*.

Chapter 3

Simplifying Constraints

A crucial element in most constraint-solving procedures is the ability to rewrite constraints into a "solved form;" this is also the case for unification algorithms (sorted and otherwise). In a constraint logic framework, the only operational requirement is being able to test the *satisfiability* of a conjunction of constraints, that is, whether it has a (non-empty) solution [Jaffar and Lassez, 1987; Comon and Delor, 1991]; Siekmann [1989, page 236] writes:

> The essential idea of this **constrained-oriented** framework for unification is the following: instead of representing the solutions of a given equation system ... explicitly as a set of unifiers, some "solved form" of these equations themselves is taken as a representation of the solution ... The problem is that there is as yet no satisfactory foundation for the notion of a "solved form"; for example a decidable unification problem would already count on its own as a representation of its solution ..."

We will use shallow SSC's to express solutions to sets of equations between constrained terms. The actual "solved form" for these unification problems, which should *not* be confused with the solved form for SSC's, will contain ordinary substitutions together with SSC's, as we will see in Chapter 4. We will also require that all the SSC's C appearing in a "solved form" unification problem be themselves in (SSC-) solved form. Satisfiability is then simple to test, by identifying the variables $X \in Dom(C)$ such that $M_C(X) = \emptyset$ for each SSC C; we call these *empty variables*. At this point, C might be simplified further: If there are any (non-empty) solutions to the original set of constrained equations, then all empty variables can be removed from C.

An algorithm that rewrites a shallow SSC into an equivalent one in solved form (as given in Definition 2.23) is presented in Sections 3.1 and 3.2. Algorithms that identify and simplify occurrences of empty variables are presented in Section 3.3.

3.1 Simplification Transformations

We now present an algorithm that rewrites a shallow SSC into an equivalent one in solved form. This procedure does most of the work in unifying constrained terms, so we examine it in some detail. The algorithm is basically an extension of the simplification procedure in [Heintze and Jaffar, 1990]. The main difficulty appears when simplifying subexpressions of the form

$$f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n).$$

Since we are allowing variable dependencies within terms, we cannot simply do a pairwise matching of variables, where the above expression would be replaced by $f(Z_1, \ldots, Z_n)$, adding the set constraints $\langle Z_i \equiv X_i \sqcap Y_i \rangle$ for new variables $\{Z_1, \ldots, Z_n\}$. Instead, a simple form of unification must be used.

We first describe a set of transformations on a system of set constraints C. Transformations S1–S4 replace a subexpression in $Rng(C)$ by a new set expression, and transformation S5 adds new set constraints over new variables, as follows:

- **Transformation S1**: Replace $f(X_1, \ldots, X_n) \sqcap g(Y_1, \ldots, Y_m)$ by $\perp$ if $f \neq g$, for $m, n \geq 0$.
- **Transformation S2**: Replace $(exp \sqcap \perp)$ by $\perp$, $(exp \sqcap \top)$ by exp, $(exp \sqcup \perp)$ by exp, and $(exp \sqcup \top)$ by $\top$, where exp is not null.
  Replace $t \sqcap t$ by t, and replace $t \sqcup t$ by t, for any atom t.
- **Transformation S3**: Replace $V \sqcap exp$ by $DNF(C(V) \sqcap exp)$, where $V \in Dom(C)$, exp is not null, and exp contains no un-nested occurrences of V.
- **Transformation S4**: Replace $f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)$ by $f(Z_1, \ldots, Z_n)$, provided that:
  - (a) For all i, j, $1 \leq i, j \leq n$, $X_i = X_j$ or $Y_i = Y_j$ iff $Z_i = Z_j$, and
  - (b) For each i, $1 \leq i \leq n$, $C \models Z_i \equiv V_1 \sqcap \ldots \sqcap V_m$, where

$$\{V_1, \ldots, V_m\} = \{X_j \mid Z_j = Z_i\} \cup \{Y_j \mid Z_j = Z_i\}, 1 \leq j \leq n.$$

- **Transformation S5**: Add to C a set constraint $\langle V \equiv W_1 \sqcap \ldots \sqcap W_m \rangle$, such that $W_i \in Dom(C)$, $1 \leq i \leq m$, and V is a new variable.

Notice that transformation S3 is applicable only if exp is not null and all of its conjuncts are different from V. If $V \sqcap exp$ contains more than one un-nested occurrence of V, transformation S2 can be applied until only one of them remained, and transformation S3 could then be applied. In general, we do *not* replace V by $C(V)$ in a set constraint of the form $\langle X \equiv V \sqcup exp \rangle$.

In the simplification algorithm, transformation S5 is used to provide the variables needed to satisfy conditions (a) and (b) in transformation S4. Transformation S4 preserves the dependencies between repeated variables within each term without introducing new dependencies. For example, $f(X, X) \sqcap f(Y, Z)$ would be be replaced by $f(V_1, V_1)$, where V_1 is a new variable, after adding the set constraint $\langle V_1 \equiv X \sqcap Y \sqcap Z \rangle$. However, $f(X, Y) \sqcap f(Y, X)$ is replaced by $f(V_1, V_2)$ — not with $f(V_1, V_1)$ — given that $\langle V_1 \equiv X \sqcap Y \rangle$ and $\langle V_2 \equiv X \sqcap Y \rangle$ have been added. The variables X and Y are not "shared" across the two terms, in the sense that they do not have to be instantiated to the same value in both cases.[1]

We make use of the following simple observation later on:

Lemma 3.1 *Let C be a shallow* SSC *in DNF and let C' be the result of applying any of transformations* S1–S5 *to C. Then C' is also a shallow* SSC *in DNF.*

[1]The set constraints $\langle V_1 \equiv X \sqcap Y \rangle$ and $\langle V2 \equiv X \sqcap Y \rangle$ will later be rewritten to $\langle V_1 \equiv exp \rangle$, $\langle V2 \equiv V1 \rangle$, for some set expression exp.

Proof: C is a shallow SSC in DNF iff $C(X)$ is in DNF for all $X \in Dom(C)$ and all the proper subterms in $Rng(C)$ are variables. We now show that these two properties remain invariant after any of the transformations is applied.

A set constraint added by transformation S5 satisfies both properties. In the case of transformations S1–S4, we only need to consider the variable X where $\langle X \equiv C(X)\rangle$ is the transformed set constraint in C: If $C(X)$ is in DNF, then clearly $C'(X)$ is in DNF in the case of transformations S1 and S2. Transformations S3 and S4 explicitly rewrite the new set expression into disjunctive normal form to ensure this. If all the subterms in C are variables, then this is also true of C': Transformation S4 is the only one that introduces new subterms, which are always variables. ∎

Also, note that if a subset of C is itself an SSC in solved form, then this subset will never be changed by any of the simplification transformations.

The following lemma shows that all of the above transformations are correct, in that they preserve the models for the SSC over the original domain:

Lemma 3.2 *Let C be an* SSC *with no cycles. If C' is obtained by applying any of transformations* S1–S5 *to C, then for any model I of C there is a model I' of C' such that $I(X) = I'(X)$ for all $X \in Var(C)$.*

Proof: For transformations S1–S4, it suffices to show that $I(E) = I(E')$, where the disjunct E is replaced by E' by the transformation; then if I is a model of C, it will also be a model of C'. In the case of transformation S5, I is extended, in the obvious way, to obtain I'. We now consider each transformation:

- Transformation S1: $I(f(X_1, \ldots, X_n) \sqcap exp) = \emptyset = I(\perp)$.
- Transformation S2: $I(e \sqcap \perp) = I(\perp)$, $I(e \sqcap \top) = I(e)$, $I(e \sqcup \perp) = I(e)$, $I(e \sqcup \top) = I(\top)$, $I(e \sqcap e) = I(e)$, $I(e \sqcup e) = I(e)$, $I(t \sqcap t) = I(t)$, and $I(t \sqcup t) = I(t)$.
- Transformation S3:
  $I(V) = I(C(V))$, so $I(V \sqcap exp) = I(C(V) \sqcap exp) = I(DNF(C(V) \sqcap exp))$.
- Transformation S4: Let $f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)$ be the transformed set expression, and let $f(Z_i, \ldots, Z_n)$ be the new one. We have to prove that $I(f(Z_i, \ldots, Z_n)) = I(f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n))$. We now show that these two sets are subsets of each other:

  For each i, $1 \le i \le n$, let S_i be $\{X_j \mid Z_j = Z_i\} \cup \{Y_j \mid Z_j = Z_i\}$.

  – Consider $t \in I(f(Z_1, \ldots, Z_n))$; t is of the form $f(Z_1, \ldots, Z_n)\theta$, where $Z_i\theta \in I(Z_i)$. From condition (a), there exists a substitution Ω such that

  $$f(Z_1, \ldots, Z_n) = f(X_1, \ldots, X_n)\Omega.$$

  From condition (b), $I(Z_i) \subseteq I(X_j)$ for all X_j such that $X_j\Omega = Z_i$; in particular, for $j = i$. Therefore, $(X_i\Omega)\theta = Z_i\theta \in I(X_i)$, so $t \in I(f(X_1, \ldots, X_n))$.
  Similarly, we can show that $t \in I(f(Y_1, \ldots, Y_n))$, so

  $$\begin{aligned} I(f(Z_1, \ldots, Z_n)) &\subseteq I(f(X_1, \ldots, X_n)) \cap I(f(Y_1, \ldots, Y_n)) \\ &= I\,(f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)). \end{aligned}$$

- Now consider $t \in I(f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n))$.
  This holds iff $t \in I(f(X_1, \ldots, X_n))$ and $t \in I(f(Y_1, \ldots, Y_n))$. Thus there are substitutions θ_X and θ_Y such that

$$\begin{aligned} &t = f(X_1, \ldots, X_n)\theta_X = f(Y_1, \ldots, Y_n)\theta_Y = f(t_1, \ldots, t_n), \\ &t_i = X_i\theta_X \in I(X_i), \text{ and} \\ &t_i = Y_i\theta_Y \in I(Y_i). \end{aligned}$$

  Therefore, if $S'_i = \{X \mid X\theta_X = t_i\} \cup \{Y \mid Y\theta_Y = t_i\}$, then $t_i \in \bigcap_{X \in S_i} I(X)$. The set $S_i = \{X_j \mid Z_j = Z_i\} \cup \{Y_j \mid Z_j = Z_i\}$ is a subset of S'_i, so $t_i \in \bigcap_{X \in S_i} I(X) = I(Z_i)$. We know that if $Z_i = Z_j$, then $t_i = t_j$, so there is a substitution θ_Z such that $t = f(Z_i, \ldots, Z_n)\theta_Z$, and as shown above, $Z_i\theta_Z = t_i \in I(Z_i)$, which means that $t \in I(f(Z_i, \ldots, Z_n))$. Therefore, $I(f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)) \subseteq I(f(Z_1, \ldots, Z_n))$.

- Transformation S5: Let $\langle V \equiv W_1 \sqcap \ldots \sqcap W_n \rangle$ be the new set constraint. If I is a model for C, we define I' as follows: $I'(V) = I(W_1) \cap \ldots \cap (W_n)$, $I'(X) = I(X)$ if $X \in \mathit{Var}(C)$, and $I'(X) = \emptyset$ if $X \notin \mathit{Var}(C) \cup \{V\}$. Clearly, I' is a model of C', and $I(X) = I'(X)$ for all $X \in \mathit{Var}(C)$. ■

Transformation S3 is problematic since, in general, it can lead to infinite loops. As a simple example, $\{\langle X \equiv (X \sqcap c) \sqcup \mathit{exp} \rangle\}$ would be rewritten to $\{\langle X \equiv (X \sqcap c) \sqcup (\mathit{exp} \sqcap c) \sqcup \mathit{exp} \rangle\}$, and so on. However, this will never be the case if cycles are not allowed in the input SSC. When cycles are present, this problem can be solved by adding a new transformation, equivalent to what Heintze and Jaffar [1990] call *cycle removal*, where $\langle X \equiv (X \sqcap \mathit{exp}) \sqcup e \rangle$ is replaced by $\langle X \equiv e \rangle$. It can be shown that that this transformation preserves the equivalence between the *least models* of C and C'. (Notice that an SSC with cycles can have more than one model that extends a fixed parameter assignment.) We will not consider these cases, as we will only use SSC's that have a unique model given a fixed parameter assignment.

Therefore, to formally ensure that we do not need the cycle removal transformation *and* to avoid termination problems, we require that the input SSC contain no cycles. This is always the case when the simplification procedure is used as a subroutine by the main constraint-solving algorithm presented in Chapter 4.

Lemma 3.3 *If an* SSC *contains no cycles, then the result of applying any of transformations* S1–S5 *contains no cycles either.*

Proof: Let C be the original SSC and let C' be the resulting one; let $\langle X \equiv C(X) \rangle$ be the set constraint in C to which the transformation was applied. Consider each transformation:

- It should be clear that transformations S1 and S2 cannot introduce any cycles, since all $C(X)$ of the form $(V \sqcap \mathit{exp}) \sqcup e$ in C' have the same form in C, and no new set constraints are added to C.

- Transformation S3: Any cycle in C' would have to involve X, since otherwise the same cycle would occur in C. Thus there would be a sequence of variables $X = X_1, \ldots, X_n = X$, where for each i, $1 \leq i < n$, $C(X_i)$ contains an un-nested occurrence of X_{i+1}. Without loss of generality, we can assume that $X_i \neq X$ for $1 < i < n$.

$C'(X)$ is of the form $DNF(C(V) \sqcap exp) \sqcup e$ and could be expressed as $(X_2 \sqcap exp_1) \sqcup e_1$. Thus X_2 has to occur, un-nested, within $DNF(C(V) \sqcap exp)$; otherwise, it would have to occur in e, and thus the same loop could have been found in C. For the same reason, X_2 cannot occur un-nested within exp. From the properties of DNF, we can conclude that X_2 has to occur un-nested within $C(V)$. But then the sequence of variables $\langle X = X_1, V, X_2, \ldots, X_n = X\rangle$ would form a cycle in C, a contradiction. Thus, transformation S3 cannot introduce any cycles in C'.

- Transformation S4 replaces $f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)$ by $f(Z_1, \ldots, Z_n)$, so it does not add any new occurrences of non-nested variables in $Rng(C')$; thus, any cycle in C' would also be present in C (i.e. it cannot involve the new expression $f(Z_1, \ldots, Z_n)$).

- Transformation S5 adds a set constraint $\langle V \equiv exp\rangle$ for a new variable V. Again, we note that if there was a cycle in C', then that cycle would also occur in C: A cycle in C' could not involve the new variable V, since X does not occur in $Rng(C')$. Therefore, there cannot be any cycles in C' if there are no cycles in C. ■

Theorem 3.4 *Let C' be obtained by applying one of transformations* S1–S5 *to an* SSC *C that contains no cycles. Then given a parameter assignment A defined on $Par(C)$ and models I of C and I' of C' that extend A, $I(X) = I'(X)$ for all $X \in Var(C)$.*

Proof: No transformation introduces any parameters, so $Par(C) = Par(C')$. Lemma 3.3 shows that no cycles are introduced by any of the transformations, so C' has no cycles. In Section 2.4, we proved that SSC's that contain no cycles have unique models that extend any given parameter assignment. Therefore, C and C' each has a *unique* model that extends A. Lemma 3.2 states that any model I of C can be extended to a model I' of C such that $I(X) = I'(X)$ for all $X \in Var(C)$. Therefore, the unique model of C that extends A has to agree with the unique model of C' that extends A over $Var(C)$. ■

3.2 Simplification Algorithm

The simplification algorithm needs to keep track of the new variables introduced by transformation S5. Some of these new variables represent intersections of variables in $C(X)$, while others are simply "renamings" of these, used to avoid introducing new inter-variable dependencies.

To this effect, we maintain a set Int that defines a function from variables into finite sets of variables. More formally, Int is a finite set of pairs of the form $\langle V, \{V_1, \ldots, V_n\}\rangle$, where $Int(V) = S$ if $\langle V, S\rangle \in Int$. If C is the SSC at any point during the algorithm and C_0 is the initial SSC, then the function Int will be defined for all the variables in $Dom(C)$, and $Int(X)$ will always be a subset of $Dom(C_0)$. We extend the definition of Int to sets of variables in $Dom(C)$, where $Int(\{X_1, \ldots, X_n\}) \stackrel{\text{def}}{=} Int(X_1) \cup \ldots \cup Int(X_n)$.

We will also construct a one-to-one function, Rep, which maps subsets of $Dom(C_0)$ into variables in $\mathcal{V}_\mathcal{S}$. It will always be the case that if if $Rep(\{X_1, \ldots, X_n\}) = V$ then $C \models V \equiv X_1 \sqcap \ldots \sqcap X_n$. Therefore, the sets for which Rep is defined will be the *solved intersections* so far. We represent Rep as a finite set of pairs of the form $\langle \{V_1, \ldots, V_n\}, V\rangle$, where $Rep(S) = V$ iff $\langle S, V\rangle \in Rep$.

This is an extension of the labeling scheme used in [Heintze and Jaffar, 1990], where new variables are subscripted with the set of variables whose intersection they represent; here, Int keeps track of those sets.

We can now define the following variant of transformations S4 and S5, which is applied to an SSC C and the sets Int and Rep:

Transformation S4′: Consider $f(X_1, \ldots, X_n) \sqcap f(Y_1, \ldots, Y_n)$.

Let $t_1 = f(X_1, \ldots, X_n)$ and $t_2 = f(Y_1, \ldots, Y_n)$. Let Ω be a renaming substitution from $Var(t_2)$ into new variables, so that $Var(t_2\Omega)$ is a set of new variables. Let θ be a most general unifier of t_1 and $t_2\Omega$ such that $Var(t_1\theta) \subseteq Var(t_2\Omega)$.

FOR EACH $V \in Var(f(X_1, \ldots, X_n)\theta)$, DO:

- Let $S_V := Int\,(\{X_i \mid X_i\theta = V\} \cup \{Y_i \mid Y_i\Omega\theta = V\})$.
- If there exists a variable Z such that $Rep(S_V) = Z$, then add to C the set constraint $\langle V \equiv Z \rangle$.
  Otherwise, add to C the set constraint $\langle V \equiv V_1 \sqcap \ldots \sqcap V_m \rangle$, where $S_V = \{V_1, \ldots, V_m\}$, and let $Rep := Rep \cup \langle S_V, V \rangle$.
- Let $Int := Int \cup \langle V, S_V \rangle$.

Replace $(t_1 \sqcap t_2)$ by $t_1\theta$.

Lemma 3.5 *Transformation* S4′ *is equivalent to transformations* S4 *and* S5*, provided that*
(i) $Int(V) = \{X_1, \ldots, X_n\}$ *only if* $C \models V \equiv X_1 \sqcap \ldots \sqcap X_n$*, and*
(ii) $Rep(\{X_1, \ldots, X_n\}) = V$ *only if* $C \models V \equiv X_1 \sqcap \ldots \sqcap X_n$*.*
Furthermore, the above hypotheses, which we call the main invariants*, will still hold for* C' *and the new* Int *and* Rep *after transformation* S4′ *is applied.*

Proof: Let $f(Z_1, \ldots, Z_n) = t_1\theta = t_2\Omega\theta$. Since Ω is a renaming of the variables in t_2 and θ is a most general unifier of t_1 and $t_2\Omega$, $Z_i = Z_j$ implies that either $X_i = X_j$ or $Y_i = Y_j$. Conversely, if $X_i = X_j$ or $Y_i = Y_j$, then $Z_i = Z_j$, since $f(Z_1, \ldots, Z_n)$ is an instance of both t_1 and t_2. Thus, condition (a) of transformation S4 is satisfied. Note that for any variable $V \in Var(f(Z_1, \ldots, Z_n))$,

$$\{X_i \mid X_i\theta = V\} \cup \{Y_i \mid Y_i\Omega\theta = V\} = \{X_j \mid X_j = X_q\} \cup \{Y_j \mid Z_j = V\}.$$

We denote this set by $S(V)$. For any set S, the set expression $V_1 \sqcap \ldots \sqcap V_n$ where $S = \{V_1, \ldots, V_n\}$ is denoted by $(\sqcap_{X \in S} X)$.

If C_1 and C_2 are the SSC's before and after each execution of the FOR loop of transformation S4′, then $C_1 \overset{Dom(C_1)}{\Longleftrightarrow} C_2$, since only the equivalent of transformation S5 might have been applied (the set constraints in C_1 are a subset of those in C_2). Thus, if $C_1 \models E_1 \equiv E_2$, then $C_2 \models E_1 \equiv E_2$. Therefore, we only have to show that the invariants apply to the pairs added to Int and Rep.

Let V be the chosen variable in $f(Z_1, \ldots, Z_n)$. The following considerations hold for each execution of the FOR loop of transformation S4′, assuming that the main invariants hold before:

- Since $S_V = \bigcup_{X \in S(V)} Int(X)$, then $C \models (\sqcap_{X \in S_V} X) \equiv (\sqcap_{X \in S(V)} X)$, assuming that the invariant for Int holds.

- If the variable Z exists, adding the set constraint $\langle V \equiv Z \rangle$ means that $C \models V \equiv \sqcap_{X \in S_V} X$, assuming that the invariant for *Rep* holds: $C \models Z \equiv \sqcap_{X \in S_V} X$, since $Rep(S_V) = Z$.
- If Z does not exist and we add the set constraint $\langle V \equiv V_1 \sqcap \ldots \sqcap V_n \rangle$, then clearly $C \models V \equiv \sqcap_{X \in S_V} X$. This is the only case where *Rep* is changed: The pair $\langle S_V, V \rangle$ is added to it. Thus, the invariant for *Rep* is still true after each execution of the FOR loop.
- Similarly, the invariant for *Int* is also maintained, since when the pair $\langle V, S_V \rangle$ is added to *Int*, $C \models V \equiv \sqcap_{X \in S_V} X$.
- From the above points we can conclude that $C \models V \equiv \sqcap_{X \in S(V)} X$.

Let C_n be the SSC after adding all the new set constraints to C (equivalent to repeated applications of transformation S5) but before $C(X)$ is changed. Again, C is a subset of C_n, and $C \overset{Dom(C)}{\Longleftrightarrow} C_n$. Since $C_n \models V \equiv \sqcap_{X \in S(V)} X$ for each new V, this all means that condition (b) of transformation S4 also holds. Since conditions (a) and (b) for transformation S4 both hold, changing C_n into C' is an instance of it. Therefore, from Lemma 3.2, $C_n \overset{Dom(C)}{\Longleftrightarrow} C'$, and the main invariants are also true of C' and the new *Int* and *Rep*, since they were true of C_n for the same sets. ∎

A simplification algorithm can now be obtained by initializing *Int* and *Rep* and applying transformations S1–S3 and S4′ as shown in Figure 3.1. Transformations S1–S3 leave *Int* and *Rep* intact.

<u>Input</u>: An SSC C_0 in DNF.
<u>Output</u>: An SSC in solved form.
Let $C := C_0$. Let $Int := \emptyset$. Let $Rep := \{\langle \{X\}, X \rangle \mid X \in Dom(C_0)\}$.

Exhaustively apply transformation S3 to C.
WHILE any one of transformations S1, S2 or S4′ apply to C, DO:
    Apply transformation S1 or S2 to C, or apply transformation S4′ to C, *Int* and *Rep*.
    IF transformation S4′ is applied and adds new elements to *Rep*,
        THEN exhaustively apply transformation S3 to C.
END WHILE

Return the resulting C.

Figure 3.1: SSC simplification algorithm

3.2.1 Correctness

We now prove that the main invariants hold throughout the algorithm, by showing that they are preserved by all transformations:

Lemma 3.6 *At any point of the algorithm, if* $Int(X) = \{Y_1, \ldots, Y_n\}$ *or* $Rep(\{Y_1, \ldots, Y_n\}) = X$, *then* $C \models X \equiv Y_1 \sqcap \ldots \sqcap Y_n$.

Proof: Clearly, this holds at the start of the algorithm, since $Int(X) = \{X\}$ for all X and $Rep(\{X\}) = X$ for all $X \in Dom(C)$.

At any point, only variables in C can appear in Int or Rep. Transformations S1, S2 and S3 do not change either of these sets and have been shown to preserve equivalence over the variables in C. Thus, the main invariants hold whenever transformation S4$'$ is applied. Lemma 3.5 shows that this transformation also preserves them. ■

Therefore, it follows from Lemma 3.2 and Lemma 3.5 that transformations S1, S2, S3 and S4$'$ each preserve the equivalence of the interpretations over $Dom(C_0)$, where C_0 is the input SSC. So, if C' is output by the algorithm, then $C \overset{Dom(C_0)}{\Longleftrightarrow} C'$. Furthermore, Theorem 3.4 states that C_0 and C' have the exactly the same unique model over $Dom(C_0)$ that extends any parameter assignment. In particular, they have the same least model over $Dom(C_0)$, so that $M_C(X) = M_{C'}(X)$ for all $X \in Dom(C)$.

Now we show that the output is indeed in solved form:

Lemma 3.7 *If C is a shallow* SSC *with no parameters and none of transformations* S1, S2, S3 *or* S4$'$ *applies, then C is in solved form.*

Proof: If no transformation applies, then there are no occurrences of "$\sqcap$" in $Rng(C)$. We can see this by enumerating the possible forms that an expression $e_1 \sqcap \ldots \sqcap e_n$, $n \geq 2$, can have in a shallow SSC in DNF:

1. $e_i = \bot$ or $e_i = \top$: Transformation S2 would apply.
2. $e_i = V$ for a variable $V \in Dom(C)$: Transformation S3 would apply.
3. $e_i = f_1(X_1, \ldots, X_n)$: If some e_j were another term, either transformation S1, transformation S4$'$ or the $c \sqcap c$ simplification in transformation S2 would apply. All other possibilities for e_j fall under the above cases.

Therefore, $C(X) = e_1 \sqcup \ldots \sqcup e_n$ for all $X \in Dom(C)$, where e_i is an atomic set expression for $1 \leq i \leq n$.

If $n > 1$ and some e_i were "$\top$" or "$\bot$", transformation S2 would apply. Thus if $n > 1$, then no e_i is "$\top$" or "$\bot$". Therefore, C satisfies all the conditions for solved form. ■

Corollary 3.8 *If the simplification algorithm of Figure 3.1 halts and the input C_0 contains no parameters, then the output* SSC *is in solved form.*

Proof: Transformation S3 clearly cannot apply at the beginning of the WHILE loop, since it has been exhaustively applied. Now, if transformation S3 does not apply before transformations S1 and S2 are used, then it cannot be applied afterwards either, since no new occurrences of non-nested variables can introduced by transformations S1 and S2.

Similarly, if transformation S3 is not applicable and transformation S4$'$ is applied without adding new elements to Rep, then the only new occurrences of non-nested variables are in set constraints afterwards, either.

If transformation S4$'$ does introduce new elements to Rep, transformation S3 is applied exhaustively. Therefore, transformation S3 is never applicable at the end of each execution of the WHILE loop. This means that if the algorithm halts, none of transformations S1–S3 or S4$'$ can be applied, so by Lemma 3.7 the output is in solved form. ■

3.2.2 Termination

As is the case for the simplification algorithm in [Heintze and Jaffar, 1990], we have to show that only a finite number of new variables can be introduced by the algorithm in order to prove that it terminates.

Lemma 3.9 *Let C_0 be the* SSC *input to the Simplification Algorithm, and let C be the current* SSC *at any point of the algorithm. Then the following invariants hold:*

1. *Rep defines a one-to-one function.*
2. *If $X \in Var(C)$ and $Int(X) = S$, then $S \subseteq Dom(C_0)$;*
3. *If $Rep(S)$ is defined for some set S, then $S \subseteq Dom(C_0)$;*

Proof: Clearly, the above invariants hold at the start of the algorithm, from the initialization of Int and Rep. All invariants will hold after transformations S1–S3, since they do not change Rep or Int. Assume now that they hold of C before transformation S4$'$ is applied.

Invariant 1 will still hold afterwards, since $\langle S, V\rangle$ is added to Rep only when no variable Z such that $Rep(V) = Z$ exists, and V is a new variable.

Consider each execution of the loop in transformation S4$'$. By hypothesis, since all the variables X in t_1 and t_2 occur in C, $Int(X) \subseteq Dom(C_0)$, so $S_V \subseteq Dom(C_0)$. Thus, adding $\langle V, S_V\rangle$ to Int preserves invariant 2, and adding $\langle S_V, V\rangle$ to Rep preserves invariant 3.

This means that all the invariants are preserved by transformation S4$'$, and so by all the transformations in the algorithm. ∎

Corollary 3.10 *During the simplification algorithm of Figure 3.1, Rep can contain at most $2^{|Dom(C_0)|}$ elements.*

Proof: Rep is a 1–1 function defined only over subsets of $Dom(C_0)$; the range of Rep contains at most one distinct variable for each subset of $Dom(C_0)$. Therefore $|Rep| \leq 2^{|Dom(C_0)|}$. ∎

The following definition is used in the proof of the next theorem:

Definition 3.11 ($\mathcal{P}(E, C)$) *Given a set expression E and an* SSC *C, let $\mathcal{P}(E, C)$ be the set of all the set expressions that can be obtained by repeatedly replacing an occurrence of an un-nested variable X in E by $C(X)$.*

That is, $\mathcal{P}(E, C)$ is the set S such that (a) $E \in S$, and (b) if $exp \in S$ and exp contains an un-nested occurrence of a variable $Y \in Dom(C)$, then the set expression exp' obtained from exp by replacing that occurrence of Y by $C(Y)$ is also in S.

If C is finite and has no cycles, $\mathcal{P}(E, C)$ has to be finite for any E; otherwise, there would be a sequence of variables $\langle X_1, \ldots, X_n\rangle$ such that X_{i+1} occurs un-nested in $C(X_i)$ for $1 \leq i < n$ and $X_n = X_1$.

Theorem 3.12 *If C has no cycles, then transformation* S3 *can only be applied a finite number of times.*

Proof: Let P be $\sum_{X \in Dom(C)} |\mathcal{P}(X, C)|$. Since C is finite and has no cycles, P is always a finite, non-negative integer. Transformation S3 always reduces the value of P, by replacing an un-nested variable V in $C(X)$ by $C(V)$: Let C be the original SSC and let C' be the resulting one. For any variable $Y \in Dom(C)$, $\mathcal{P}(Y, C') \subseteq \mathcal{P}(Y, C)$, and if $C(X)$ was changed the transformation, then $\mathcal{P}(X, C') \subset \mathcal{P}(X, C)$, since $C(X) \notin \mathcal{P}(X, C')$ Therefore, the total sum P is always reduced by at least 1.

Since $P \geq 0$, the exhaustive application of transformation S3 terminates. ∎

Theorem 3.13 *If the input* SSC *has no cycles, then the Simplification Algorithm of Figure* 3.1 *terminates.*

Proof: Since no transformation introduces new cycles, by Lemma 3.3, if C_0 contains no cycles then the initial exhaustive application of transformation S3 and all subsequent ones will terminate.

To show that the main WHILE loop terminates, we can use as the "measure of progress" the 3-tuple $\langle B - N, I, S \rangle$, where:

1. B is a constant equal to $2^{|Dom(C_0)|}$ and N is the size of *Rep*,

2. I is the number of occurrences of "$\sqcap$" in $Rng(C)$, and

3. S is the total number of symbols in $Rng(C)$.

The values of I and S are obviously non-negative; and since $|Rep| \leq 2^{|Dom(C_0)|} = B$, $B - N$ will always be greater than or equal to 0. Therefore, the left-to-right lexicographic ordering $\prec$ on $\langle B - N, I, S \rangle$ is well-founded. We can now show termination of the algorithm by showing that each execution of the WHILE loop results in an SSC strictly smaller than the previous one in the $\prec$ ordering. At each execution of the loop, one of transformations S1 and S2 is applied, or else transformation S4$'$ is applied, possibly followed by an exhaustive application of transformation 3. Then:

- No transformation can increase $B - N$, since elements of *Rep* are never removed.

- Transformations 1 and 2 always reduce S, and possibly I. They do not add new occurrences of "$\sqcap$", so I is never increased.

- If transformation S4$'$ does not add an element to *Rep*, then it has replaced $t_1 \sqcap t_2$ by $f(V_1, \ldots, V_n)$, where the V_i are new variables, and added the set constraints $\langle V_i \equiv Z_i \rangle$ to C, for variables $Z_i \in Dom(C)$. Note that in this case, transformation S3 is *not* applied.[2] Thus, even though S may have increased, the value of I will have decreased, since one occurrence of "$\sqcap$" is deleted and no new ones are added.

  If transformation S4$'$ adds new elements to *Rep*, then transformation S3 may be applied afterwards, possibly increasing I and S. However, $B - N$ is then reduced by at least 1.

Since the 3-tuple in its lexicographical ordering $\prec$ is always strictly smaller after each iteration of the main WHILE loop, the loop terminates (since each iteration also does). Thus the algorithm terminates. ∎

[2] It *could* not be applied, even if the algorithm's control allowed it.

Transformation	$B - N$	I	S
1	=	$\leq$	<
2	=	$\leq$	<
4'+3	<	—	—
4'	=	<	—

Table 3.1: Termination of the SSC simplification algorithm

Table 3.1 is a schematic representation of the termination proof, where a "<" symbol indicates that the given measure is always decreased under the given transformation, "$\leq$" means that it possibly decreases but does *not* increase, and "=" means that it stays the same.

The following lemma shows that the algorithm halts when solved form is reached:

Lemma 3.14 *If $C(X)$ is in solved form, then none of transformations* S1, S2, S3 *and* S4$'$ *is applicable to X.*

Proof: If $C(X)$ is in solved form, it cannot contain any occurrences of "$\sqcap$". Thus, none of transformations S1, S3, or S4 can apply, as well as the "$\sqcap$" simplification operations in transformation S2. Since "$\perp$" cannot occur either, "$\top$" has to stand by itself. Since disjuncts cannot be repeated in solved form, none of the other operations in transformation S2 can apply either. ∎

3.2.3 Remarks

The question of an *efficient* implementation of this algorithm is left open, as was also the case in [Heintze and Jaffar, 1990]; so is the complexity of the simplification problem.[3] Some simple comments follow:

First of all, the algorithm could be simplified greatly if the input SSC were guaranteed to be linear. In this case the transformations would be very similar to the ones in [Heintze and Jaffar, 1990], since no inter-variable dependencies would have to be preserved.

The restriction on the application of transformation S3 is intended to make the termination proof easier. In general, many changes to the control structure are possible. For example, it is clear that we would want to give the highest priority to transformations S1 and S2. These two transformations can also be applied whenever possible during the exhaustive applications of transformation S3, without compromising the termination of the algorithm. Also, the transformations presented in Figure 3.3 of Section 3.3 for simplifying the occurrences of "$\perp$" could be integrated with high priority into this algorithm, and would help avoid unnecessary work in computing the intersection of a set of variables that includes an empty one (the intersection is simply replaced by "$\perp$").

Once the algorithm terminates, we may want to remove all the empty variables in C. Since C is now in solved form, we can use the algorithms in Section 3.3 to do this. In practice, this procedure can be integrated into the simplification algorithm; if the right control is used, it might improve efficiency considerably. It is now separate to simplify the correctness proofs — although an implementation should eventually "merge" the two procedures.

[3] Trivially, the termination proof above gives an exponential upper bound on the space and time complexity of the algorithm.

In order to minimize the number of intersections to solve, it is desirable to delay transformation S3, which replaces X by $C(X)$, as much as possible. We could also use a variant of transformation S3 that takes advantage of the solved intersections: If $\langle X \equiv (V_1 \sqcap \ldots \sqcap V_n) \sqcup exp \rangle \in C$ and $Rep(V_1, \ldots, V_n) = Y \neq X$, then we can replace this set constraint by $\langle X \equiv Y \sqcup exp \rangle$. However, the problem of guaranteeing that no new cycles are introduced complicates the correctness and termination proofs. Simplifying intersections of the form $f_1(\cdots) \sqcap \ldots \sqcap f_n(\cdots)$ can also be optimized, by extending transformation S4 to n-tuples instead of pairs.

Note that we do not want to replace an occurrence of a non-nested variable X by $C(X)$ whenever possible (as is the case in [Heintze and Jaffar, 1990]). Instead, we only replace X by $C(X)$ when we have to compute the intersection of two or more variables.

Similarly, if $\langle X \equiv exp \sqcup f(V_1, \ldots, V_n) \rangle$ and $\langle V_i \equiv Y \rangle$ are in C, and $Y \neq V_1, \ldots, V_n$, we can replace V_i in $f(V_1, \ldots, V_n)$ by Y. Note that if Y is equal to some V_i this can no longer be done, since we would be introducing new inter-variable dependencies. This is similar to "collapsing" substitutions to obtain simpler equivalent ones.

3.3 Empty Variables

In constraint logic, constrained terms and clauses are defined as *admissible* [Frisch, 1991] if there exist ground instances that satisfy the given constraints. If a term t constrained by an SSC C stands for all the ground terms in $M_C(t)$, as defined in Chapter 6, it follows that t/C is admissible if $M_C(X) \neq \emptyset$ for all variables X appearing in t. This motivates the following definition:

Definition 3.15 (Empty Variables, Admissibility) *Given an* SSC *C, we say that a variable $X \in Dom(C)$ is* empty *in C if $M_C(X) = \emptyset$. C is* admissible *over a set of variables V if no variable in V is empty in C; that is, $M_C(X) \neq \emptyset$ for all $X \in V \cap Dom(C)$. A constrained term t/C is* admissible *if C is admissible over $Var(t)$.*

We will see in Chapter 7 that the soundness of a deductive system depends on the admissibility of all the constrained terms used in a proof. Therefore, recognizing empty variables is crucial — in fact, it alone is sufficient for testing the admissibility of the constraints imposed on a term. Finding empty variables is easy if we are given an SSC in solved form, as we now show:

Figure 3.2 presents an algorithm that identifies all the empty variables in an SSC C in solved form, adapted from the corresponding one in [Heintze and Jaffar, 1990]. We assume that C has no parameters. Recall that since M_C is the least model of C, it maps all parameters to $\emptyset$, so they can be replaced by "$\perp$."

Clearly, the algorithm always terminates: $Dom(C)$ is finite and each transformation marks a variable that was previously unmarked.

Theorem 3.16 *All marked variables at the end of the algorithm in Figure 3.2 are not empty in C.*

Proof: We show that this is the case before and after each transformation is applied, that is, the theorem is an invariant for the algorithm. Initially, the invariant holds, since all variables are unmarked. Now assume that the invariant holds and one of transformations I1–I4 is applied. We show that the new marked variable is non-empty in C:

Input: A parameter-free SSC in solved form.
Output: The set of empty variables in C.
Let all variables be unmarked. Nondeterministically apply the following transformations for an unmarked $X \in Dom(C)$, until none applies:

I1: If $\langle X \equiv \top \sqcup exp \rangle \in C$, mark X.

I2: If $\langle X \equiv k \sqcup exp \rangle \in C$ for some constant k, mark X.

I3: If $\langle X \equiv Y \sqcup exp \rangle \in C$ and Y is a marked variable, mark X.

I4: If $\langle X \equiv f(Y_1, \ldots, Y_n) \rangle \in C$ and each Y_i is a marked variable, mark X.

At the end, output the set of all unmarked variables.

Figure 3.2: Identifying empty variables

1. In the case of transformation I1, $M_C(X) = M_C(\top) \neq \emptyset$, since the Herbrand Universe is assumed to be non-empty.
2. For transformation I2, $M_C(X) \supseteq \{k\} \neq \emptyset$.
3. For transformation I3, $M_C(X) \supseteq M_C(Y)$, and $M_C(Y) \neq \emptyset$ by hypothesis.
4. In the case of transformation I4, we know by hypothesis that $M_C(Y_i) \neq \emptyset$, so
$$M_C(X) \supseteq \{t = f(Y_1, \ldots, Y_n)\theta \mid Y_i\theta \in M_C(Y_i)\} \neq \emptyset. \qquad \blacksquare$$

Theorem 3.17 *All the unmarked variables at the end of the algorithm in Figure* 3.2 *are empty in* C.

Proof: We show that if $X \overset{C}{\leadsto} t$ for a term t that contains no variables in $\mathcal{V_S}$, then X is marked. By Lemma 2.17, $t \in M_C(X)$ only if $X \overset{C}{\leadsto} t$ and t contains no variables in $\mathcal{V_S}$, so this will mean that if X is unmarked, then $M_C(X) = \emptyset$.

Note that for any $X \in Dom(C)$, $X \overset{C}{\leadsto} t$ iff $C(X) \overset{C}{\leadsto} t$; that is, $X \Rightarrow t$ can only be proved from $\mathcal{A}(C)$ if $E_X \Rightarrow t$ can be proved for some disjunct E_X in $C(X)$. (See Definition 2.12 for the definition of derivations. Since we assume there are no parameters in C, the parameter assignment A is not mentioned.) We now show that if there is a derivation for $E_X \Rightarrow t$ where t has no variables in $\mathcal{V_S}$, then X is marked, using induction on the depth n of the derivation of $E_X \Rightarrow t$:
Basis: $n = 0$. Assume that $E_X \Rightarrow t$ is an axiom in $\mathcal{A}(C)$, where t has no variables in $\mathcal{V_S}$. Since C is shallow, this can only be the case if $E_X = t$, where t is a constant, or if $E_X = \top$. But then X would have been marked by one of transformations I1 and I2.
Induction Step: Assume that if there is a derivation of depth n of $E_Y \Rightarrow t$ for a disjunct E_Y in $C(Y)$ and a term t such that $Var(t) \sqcap Dom(C) = \emptyset$, then Y is marked.

Input: An SSC C such that $C(X) = \perp$ for all empty variables X in $Dom(C)$.
Output: An SSC C' equivalent to C such that no empty variables appear in $Rng(C')$.
While "$\perp$" appears in $C(Y)$ or an empty variable $Y \in Dom(C)$ appears in $Rng(C)$, do:

E1: Replace $(\perp \sqcup exp)$ by exp.

E2: Replace $(\perp \sqcap exp)$ by $\perp$.

E3: If $\langle Y \equiv \perp \rangle \in C$ and Y appears in $Rng(C)$, then
replace all the disjuncts in $Rng(C)$ that contain Y by $\perp$.

Figure 3.3: Simplifying empty variables

Assume that there is a derivation of depth $n+1$ of $E_X \Rightarrow t$, where t contains no variables in $Dom(C)$. Since C is in solved form, E_X can only be a term of the form $f(Y_1, \ldots, Y_n)$ or a variable Y, where the only inference rules that can be applied to obtain the root node $E_X \Rightarrow t$ are rules 3 and 4, respectively. Consider each case:

- $E_X = Y$, where rule 3 is used: Then there is a derivation of depth n of t from Y. But then Y has to be a marked variable, by induction hypothesis; X cannot be unmarked, since transformation I3 would have been applied.

- $E_X = f(Y_1, \ldots, Y_m) \sqcup exp$, $t = f(t_1, \ldots, t_m)$, and $Y_i \overset{C}{\mapsto} t_i$, where rule 4 is used.[4] But then each Y_i has to be marked, since t_i contains no variables in $Dom(C)$, and so X cannot be unmarked either, since transformation 4 would have applied. ∎

If $M_C(X) = \emptyset$, then for all models I of C, $I(X) = \emptyset$. Therefore, we can obtain from C an equivalent C' by replacing $\langle X \equiv C(X) \rangle$ by $\langle X \equiv \perp \rangle$ for all variables X unmarked at the end of the algorithm.

Even though a constrained term t/C might be admissible, C could still contain empty variables (not in $Var(t)$). In these cases we can rewrite C into a simpler SSC C' that is equivalent to C over $Var(t)$, so that $[t/C] = [t/C']$. This motivates a simple simplification procedure for solved form SSC's, as seen in Figure 3.3, which isolates all the empty variables in C in set constraints of the form $\langle X \equiv \perp \rangle$ and simplifies occurrences of "$\perp$": The transformations can be applied in any order, until no transformation is applicable.

To prove that the non-deterministic application of transformations E1–E3 terminates, consider the pair $\langle U, S \rangle$, where U is the number of empty variables in C that appear in $Rng(C)$ and S is the total number of symbols in C. Transformations E1 and E2 reduce S and do not increase U; transformation E3 reduces U and cannot increase S. Therefore, the pair (in either left-to-right or right-to-left lexicographic ordering) is always reduced, so the algorithm terminates. (Both U and S are clearly always non-negative integers, so the ordering is well-founded.)

[4] This is actually weaker than what the definition of rule 4 requires.

Clearly, if $C(X) = \perp$ at the end of the algorithm, then X cannot appear anywhere else in C; transformation E3 must have been applied to X at some point, removing all the occurrences of X in $Rng(C)$. Also, no element of $Rng(X)$ can contain an occurrence of $\perp \sqcup exp$ or $\perp \sqcap exp$ for any (non-empty) exp, since transformation E1 or transformation E2 would apply. Therefore, the only occurrences of "$\perp$" at the end of the algorithm are in set constraints of the form $\langle X \equiv \perp \rangle$, where X does not appear anywhere else in $Rng(C)$.

Note that unlike the previous algorithm, this one rewrites C into a new SSC. We now show that the transformations preserve the interpretations for C:

Theorem 3.18 *Let C' be the result of applying any of transformations* E1–E3 *in Figure* 3.3 *to an* SSC *C. Then I is a model of C if, and only if, I is a model of C'.*

Proof: Consider each transformation:

- Transformations E1 and E2: $I(\perp \sqcup exp) = I(exp)$ and $I(\perp \sqcap exp) = I(\perp)$, for all interpretations I and all set expressions exp.

  Therefore, in both cases $I(X) = I(C(X))$ iff $I(X) = I(C'(X))$ for all $X \in Dom(C) = Dom(C')$, so I satisfies C iff I satisfies C'.

- Transformation E3: If I satisfies C or C', then $I(Y) = \emptyset$. Therefore, $I(t) = \emptyset = I(\perp)$ if t is a term containing Y. For each $X \in Dom(X)$, $C'(X)$ is obtained by replacing all the terms containing Y in $C(X)$ by "$\perp$". Then $I(C(X)) = I(C'(X))$ for all $X \in Dom(C)$. Thus I satisfies C iff I satisfies C'. ■

Chapter 4

The Unification Algorithm

This chapter presents an algorithm that solves sets of equations between terms whose variables are constrained by shallow systems of set constraints.

Recall that given a term t and an SSC C, $[t/C] = M_C(t)$ is a set of ground terms. A term s is an *instance* of a constrained term t/C if $s \in [t/C]$. We say that s is a *common instance* of a set of constrained terms $\{t_1/C_1, \ldots, t_n/C_n\}$ if it is an instance of each, that is, if $t_i \in [t_1/C_1] \cap \ldots \cap [t_n/C_n]$. We assume that if a variable X appears in two different SSC's C_i and C_j, then either X is a parameter in both, or $C_i(X) = C_j(X)$. Therefore, the union of $C_1, \ldots, C_n$ is itself a well-defined SSC, C, and for all i, $1 \leq i \leq n$, $M_C(X) = M_{C_i}(X)$ for all $X \in Var(C_i)$. If $C_i(X) \neq C_j(X)$ for some i and j, then we could rename X into a new variable in C_i or C_j — after all, the two X's *are* different variables in this case. Thus, we talk of the common instances of $\{t_1, \ldots, t_n\}$ given a single SSC C. Note that C is in solved form if $C_1, \ldots, C_n$ are all in solved form.

More generally, we want to *unify* a set of terms $S = \{t_1, \ldots, t_n\}$ given an SSC C: A substitution ρ unifies S/C if $t_1\rho = \cdots = t_n\rho$ and $X\rho \in M_C(X)$ for all $X \in Var(S)$. The problem of finding the unifiers for constrained sets of terms can thus be rephrased as finding solutions to sets of constrained equations, as follows:

Definition 4.1 (Constrained Equation Sets) *A* constrained equation set, *written as S/C, is a finite set S of equations between terms together with an* SSC *C such that $Var(S) \subseteq Dom(C)$. A substitution θ* solves *an equation set S/C iff (i) for all equations $t_1 = t_2$ in S, $t_1\theta = t_2\theta$, and (ii) $X\theta \in [X/C]$ for all $X \in Dom(\theta)$. Equivalently, we say that θ is a* solution *of S.*

The general unification problem is to find all the substitutions (if any) that solve a constrained equation set S/C. In the case of unconstrained sets of equations, we know that we can obtain a single unifier, a *most general* one, from which all other possible solutions can be generated (by composing the appropriate substitution onto it); this is what the unsorted unification algorithm in Section 1.7 does. However, this is not always the case when constraints are introduced.

Section 4.1 presents an algorithm that solves sets of equations and *membership constraints*. In Section 4.2 this procedure is combined with the simplification algorithm of Chapter 3 to build the actual unification algorithm.

4.1 Solving Equations and Membership Constraints

Definition 4.2 (Membership Constraints) *Given an* SSC C, *a* membership constraint *is an expression* $t \Subset E$, *for a term* t *over* $\mathcal{F} \cup \mathcal{V}_{\mathcal{S}}$ *and a set expression* E. *We say that a substitution* ρ solves *a membership constraint* $t \Subset E$ *given* C *if* $t\rho \in [E_i/C]$. *A substitution* ρ *solves a set of membership constraints given* C *if it solves all of its elements given* C.

For a set of membership constraints $U = \{t_1 \Subset e_1, \ldots, t_n \Subset e_n\}$, the set of terms $\{t_1, \ldots, t_n\}$ is denoted by $lhs(U)$, and the set of set expressions $\{e_1, \ldots, e_n\}$ is denoted by $rhs(U)$. Given a set of equations S, we also use $rhs(S)$ and $lhs(S)$ to denote the sets of terms in the left and right hand sides of the equations in S.

A substitution ρ is said to *solve* an equation between terms $t_1 = t_2$ if $t_1\rho = t_2\rho$. Finding the solutions to a constrained equation set is equivalent to to finding the common solutions to a set containing both equations between terms and membership constraints. If S is such a set, then $\mathrm{MEM}(S)$ denotes the set of all membership constraints in S, while $\mathrm{EQN}(S)$ denotes the set of all equations in S. Therefore, we define the following:

Definition 4.3 (Solving Sets of Equations and Membership Constraints) *Given an* SSC C, *a substitution* ρ *is said to* solve *a set* S *of equations and membership constraints if* ρ *solves* $\mathrm{EQN}(S)$ *and* ρ *solves* $\mathrm{MEM}(S)$ *given* C.

That is, if $S = \{s_1 = s'_1, \ldots, s_m = s'_m\} \cup \{t_1 \Subset E_1, \ldots, t_n \Subset E_n\}$, then ρ solves S given C iff for all i, $1 \leq i \leq m$, $s_i\rho = s'_i\rho$ and for all j, $1 \leq j \leq n$, $t_j\rho \in [E_j/C]$.[1] The substitution ρ is said to be a *solution* of S, given C.

A constrained equation set S/C can then be written as the union of the set of equations S and the set of membership constraints $\{X_1 \Subset X_1, \ldots, X_n \Subset X_n\}$, where $\{X_1, \ldots, X_n\} = Var(S)$. (To avoid confusion, the actual unification algorithm renames the variables in S and in the right hand side of the membership constraints to be disjoint from $Var(C)$.)

The "solving algorithm" we will now present finds all the solutions of a set S of equations and membership constraints given a *solved form* SSC C, where all the set expressions E_i in $\mathrm{MEM}(S)$ are *shallow*.[2] These solutions are represented using substitutions θ_j and shallow SSC's C_j such that θ_i is a unifier of $\mathrm{EQN}(S)$, and $[t_i\theta_j/C_j] \subseteq [E_i/C]$ for all membership constraints $t_i \Subset E_i$ in $\mathrm{MEM}(S)$. Even if the initial set S contains only membership constraints, the substitution is needed because some variables in t might turn out *not* to be independent:

Example 4.4 *Consider solving* $\{f(x, y) \Subset f(Z, Z)\}$ *given the* SSC $C = \{\langle Z \equiv g(Z) \sqcup a \sqcup b\rangle\}$: *The solution is the substitution* $\{x \mapsto Z, y \mapsto Z\}$ *together with* C.

When given only a set of membership constraints of the form $\{t_1 \Subset e_1, \ldots, t_n \Subset e_n\}$, the solving algorithm will find all possible instantiations of $\{t_1, \ldots, t_n\}$ that satisfy the constraints.[3]

[1] In the notation used by Comon and Delor [1991], a set of membership constraints and equations is the conjunction $s_1 = s'_1 \wedge \ldots \wedge s_m = s'_m \wedge t_1 \in S_1 \wedge \ldots \wedge t_n \in S_n$, where each S_i is the (not necessarily regular) tree language $[E_i/C] = M_C(E_i)$.

[2] In general, the set expressions E_i can be *semi-linear*, given that C is in solved form. Hence the title of this thesis.

[3] This is similar to *weakening* substitutions to make them well-sorted (see Chapter 5), and is why the transformations that act upon the membership constraints are labeled W1–W7.

In general, there is no single pair $\langle \theta, C' \rangle$ that can capture all possible solutions; however, we will always be able to find a *finite* set of such pairs such that every solution is an instance of one of them. That is, we obtain a *set* $\{\langle \theta_i, C_i \rangle\}$ such that for all membership constraints $t \Subset E$ in MEM(S), $[t\theta_i/C_i] \subseteq [E/C]$ for all i and $\bigcup_i [t\theta_i/C_i]$ includes all the instances of t that are elements of $[E/C]$.

Example 4.5 *As a simple example, consider* $S = \{f(x, y) \Subset f(a, b) \sqcup f(c, d)\}$ *(given any* SSC*); there are two solutions to* S*, namely* $\langle \{x \mapsto a, y \mapsto b, \emptyset\} \rangle$ *and* $\langle \{x \mapsto c, y \mapsto d, \emptyset\} \rangle$*, which cannot be expressed using a single substitution and* SSC *pair.*

For simplicity, we assume that the initial SSC C contains no occurrences of "$\perp$"; we could always apply the algorithms in Section 3.3 to obtain a constraint equivalent to C such that the empty variables are isolated in set constraints of the form $\langle X \equiv \perp \rangle$ (where X appears nowhere else in C). If any of the set expressions *exp* for an initial membership constraint $t \Subset exp$ is empty in C, then there are no solutions. Otherwise, all the disjuncts in $rhs(\text{MEM}(S))$ that contain a variable empty in C can be removed from S and all the set constraints of the form $\langle X \equiv \perp \rangle$ can be removed from C. We again assume that C has no parameters, though most of the following lemmas and theorems are still true if parameters are allowed.

Like the simplification algorithm of Chapter 3, the solving algorithm is based on a set of transformations that are applied until a *normal form* is obtained. We maintain a set $\mathcal{S}$ whose elements are sets of equations and membership constraints. The main part of the solving algorithm applies transformations to the elements of such a set.

Transformations W1-W7 transform the membership constraints in a set $U \in \mathcal{S}$, possibly replacing U by new sets or removing U from $\mathcal{S}$. Transformation W3 is the only one that can change the set of equations in U, by adding new ones:

- **Transformation W1**:
  $S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset g(X_1, \ldots, X_m) \sqcup exp\}\} \implies S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset exp\}\}$, provided $f \neq g$ and exp is not null.
  If exp is null, then $S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset g(X_1, \ldots, X_n)\}\} \implies S$.

- **Transformation W2**:
  $S \cup \{U_o \cup \{t \Subset X \sqcup exp\}\} \implies S \cup \{U_o \cup \{t \Subset C(X) \sqcup exp\}\}$, for a non-variable term t.

- **Transformation W3**:
  $S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset f(X_1, \ldots, X_n)\}\} \implies$
  $S \cup \{U \cup \{t_1 \Subset X_1, \ldots, t_n \Subset X_n\} \cup \{t_i = t_j \mid X_i = X_j, 1 \leq i < j \leq n\}\}$, where $n \geq 2$.

- **Transformation W4**:
  $S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset f(t_1^1, \ldots, t_n^1) \sqcup \ldots \sqcup f(t_1^m, \ldots, t_n^m)\}\} \implies$
  $S \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset f(t_1^1, \ldots, t_n^1)\}\} \cup \ldots \cup \{U_o \cup \{f(t_1, \ldots, t_n) \Subset f(t_1^m, \ldots, t_n^m)\}\}$, where $n > 1, m > 1$.

- **Transformation W5**:
  $S \cup \{U_o \cup \{f(t) \Subset f(X_1) \sqcup \ldots \sqcup f(X_n)\}\} \implies S \cup \{U_o \cup \{t \Subset X_1 \sqcup \ldots \sqcup X_n\}\}$.

- **Transformation W6**:
  $S \cup \{U_o \cup \{c \Subset e_1 \sqcup \ldots \sqcup e_n\}\} \implies S \cup \{U_o\}$, provided c is a constant and $e_i = c$ for some i, $1 \leq i \leq n$. Otherwise, if e_i is a non-variable term and $e_i \neq c$ for all i, $S \cup \{U_o \cup \{c \Subset e_1 \sqcup \ldots \sqcup e_n\}\} \implies S$.

- **Transformation W7**:
$S \cup \{U_o \cup \{t \Subset \top \sqcup exp\}\} \implies S \cup \{U_o \cup \{X_1 \Subset \top, \ldots, X_n \Subset \top\}\}$, provided t is a non-variable term and $\{X_1, \ldots, X_n\} = Var(t)$.[4]

Transformations U1–U4 correspond to the normal unification transformations (see Section 1.7), now applied to the set of sets $\mathcal{S}$. Transformation U4, which isolates variables, is the only one that can change the membership constraints in an element of $\mathcal{S}$:

- **Transformation U1**:
$S \cup \{U_o \cup \{f(t_1, \ldots, t_n) = X\}\} \implies S \cup \{U_o \cup \{X = f(t_1, \ldots, t_n)\}\}$.

- **Transformation U2**:
$S \cup \{U_o \cup \{X = X\}\} \implies S \cup \{U_o\}$.

- **Transformation U3**:
$S \cup \{U_o \cup \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\}\} \implies S \cup \{U_o \cup \{s_1 = t_1, \ldots, s_n = t_n\}\}$;
$S \cup \{U_o \cup \{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\}\} \implies S$, if $f \neq g$.

- **Transformation U4**:
$S \cup \{U_o \cup \{X = t\}\} \implies S \cup \{U_o' \cup \{X = t\}\}$, where U_o' is the result of replacing all the occurrences of X in $\mathrm{EQN}(U_o)$ and $lhs(\mathrm{MEM}(U_o))$ by t, given that $X \notin Var(t)$ and $X \in Var(\mathrm{EQN}(U_o)) \cup Var(\mathrm{MEM}(U_o))$.
If $X \in Var(t)$ and $X \neq t$, then $S \cup \{U_o \cup \{X = t\}\} \implies S$.

We first prove some simple invariants concerning the syntax of the elements of $\mathcal{S}$:

Lemma 4.6 *If the following propositions are true of $\mathcal{S}$ before any one of transformations* W1–W7 *or* U1–U4 *is applied, then they remain true afterwards, given an* SSC C *in solved form that contains no occurrences of "$\perp$":*

- *(i) For each $U \in \mathcal{S}$, all the set expressions in $rhs(\mathrm{MEM}(U))$ are in DNF, where each disjunct is either "$\top$", a variable $V \in Dom(C)$, or $f(V_1, \ldots, V_n)$ for variables $V_1, \ldots, V_n \in Dom(C)$, $n \geq 0$ (so $Var(rhs(\mathrm{MEM}(U))) \subseteq Dom(C)$).*

- *(ii) $Var(\mathrm{EQN}(U)) \cup Var(lhs(\mathrm{MEM}(U)))$ and $Var(C)$ are disjoint.*

Proof: (i) is clearly preserved by transformations W1–W7, including transformation W2, because we require C to be in solved form and we assume that C contains no parameters and no occurrences of "$\perp$". Transformations U1–U4 also preserve invariant (i), since they do not change $rhs(\mathrm{MEM}(U))$.

Proposition (ii) is preserved by all transformations, since no new variables are ever added to either set. ∎

Definition 4.7 (Normal Form) *A set S of equations and membership constraints is in* normal form *if (i) all the elements of* $\mathrm{MEM}(S)$ *are of the form $X \Subset exp$ for a variable X and (ii)* $\mathrm{EQN}(S)$ *is of the form $\{X_1 = t_1, \ldots, X_n = t_n\}$ for distinct variables $\{X_1, \ldots, X_n\}$, where each X_i does not occur in $rhs(\mathrm{EQN}(S)) \cup lhs(\mathrm{MEM}(S))$, $1 \leq i \leq n$.*

[4] If there are no variables in t then the transformation is equivalent to $S \cup \{U_o \cup \{t \Subset \top \sqcup exp\}\} \implies S \cup \{U_o\}$.

The following lemma shows that, under certain simple conditions, a set S of equations and membership constraints is in normal form iff none of the above transformations applies:

Lemma 4.8 *Let $\mathcal{S}$ be a set of sets of equations and membership constraints given an* SSC C *in solved form, where for all $U \in \mathcal{S}$, the set expressions in $rhs(\mathrm{MEM}(U))$ are shallow and both $Rng(C)$ and $rhs(\mathrm{MEM}(U))$ contain no occurrences of "$\perp$". Then U is in normal form for all $U \in \mathcal{S}$ if, and only if, none of transformations* W1–W7 *or* U1–U4 *applies to $\mathcal{S}$.*

Proof: "Only if": If any one of transformations W1–W7 applies, then the left hand side of a membership constraint in some $U \in \mathcal{S}$ has to be a non-variable term. If all terms in $lhs(\mathrm{MEM}(U))$ are variables, then none of these transformation can apply. Transformations U1 and U3 require that an equation in some $U \in \mathcal{S}$ be of the form $s = t$ for a non-variable term s; transformations U2 and U4 can only apply if a variable in $Var(lhs(\mathrm{EQN}(U))) \cup Var(rhs(\mathrm{MEM}(U)))$ also occurs in $rhs(\mathrm{EQN}(U))$, for some $U \in \mathcal{S}$. Therefore, if U is in solved form for all $U \in \mathcal{S}$ then no transformation can apply.

"If": We show that if some $U \in \mathcal{S}$ is not in solved form, some transformation applies. If a membership constraint in U does not have a variable as its left hand side, then one of transformations W1–W7 applies: Assume that $f(t_1, \ldots, t_n) \sqsubseteq exp$ is in U for some element of $\mathcal{S}$. By hypothesis, the right hand side of the membership constraints in $\mathrm{MEM}(U)$ is of the form $e_1 \sqcup \ldots \sqcup e_m$, where each e_i is "$\top$", a variable, or $f_i(X_1, \ldots, X_n)$ for variables $X_1, \ldots, X_n$. If exp contains an un-nested variable, then transformation W2 applies. If exp contains "$\top$", then transformation W7 applies.[5] Consider the case where $exp = f_1(\cdots) \sqcup \ldots \sqcup f_m(\cdots)$: If $f_i \neq f$ for some i, $1 \leq i \leq m$, then transformation W2 would apply. Assume now that $f_i = f$ for all i, $1 \leq i \leq m$. If f is a constant or a unary function, then one of transformation W5 or W6 applies. Otherwise, transformation W3 would apply if $m = 1$, and transformation W4 would apply if $m \geq 2$.

If the left hand side of an equation in U is a non-variable term, then one of transformations U1 and U3 applies. Assume then that $\mathrm{EQN}(U)$ is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$. If some variable x_i occurred in some t_j for any i, j, $1 \leq i, j \leq n$, then one of transformations U2 or U4 applies. If x_i occurred in $rhs(\mathrm{MEM}(U))$, then transformation U4 applies. ■

The following theorem shows that transformations W1–W7 and U1–U4 are "correct and complete," in the sense that they preserve the solutions to the elements of any set $\mathcal{S}$:

Theorem 4.9 *Let $\mathcal{S}'$ be the result of applying any one of transformations* W1–W7 *or* U1–U4 *to a set of sets of equations and membership constraints $\mathcal{S}$, given an* SSC C *in solved form. Then a substitution ρ solves an element of $\mathcal{S}$ if, and only if, ρ solves an element of $\mathcal{S}'$.*

Proof: For the "only if" direction, it suffices to consider only the particular element of $\mathcal{S}$ to which the transformation applied. We show that this set is removed only if it has no solutions; otherwise, we show that if ρ solves the original set, then ρ solves one of the new sets in $\mathcal{S}$.

The "if" direction is trivial if ρ solves an element of $\mathcal{S}'$ that is also in $\mathcal{S}$, so we consider only the new sets, showing that if ρ solves one of these then ρ also solves the original one.

W1: We have two cases:

— If exp is not null, then ρ solves the original set iff ρ solves the resulting one, since no

[5] We do not have to consider occurrences of "$\perp$", since we assume that there were none in the input.

instance of $f(\cdots)$ can be in $M_C(g(\cdots))$ if $f \neq g$.
— If exp is null, then no ρ can be such that $f(t_1,\ldots,t_n)\rho \in M_C(g(X_1,\ldots,X_m))$, if $f \neq g$, so this set has no solutions and can be removed from $\mathcal{S}$.

W2: Trivially, ρ solves $U_o \cup \{t \in\!\!\!\!\_\ X \sqcup exp\}$ iff ρ solves $U_o \cup \{t \in\!\!\!\!\_\ C(X) \sqcup exp\}$, since $M_C(X) = M_C(C(X))$. Therefore, both directions hold.

W3: Let U be the set in $\mathcal{S}$ to which the transformation was applied and let E be the set of equations added to U.

"Only if": Assume that ρ solves $U = U_o \cup \{f(t_1,\ldots,t_m) \in\!\!\!\!\_\ f(X_1,\ldots,X_m)\}$. $M_C(f(X_1,\ldots,X_m))$ is a set of terms of the form $f(u_1,\ldots,u_m)$ where $u_i = u_j$ if $X_i = X_j$ and $u_i \in M_C(X_i)$ for all $i, j, 1 \leq i, j \leq m$. $f(t_1,\ldots,t_n)\rho$ is a term in $M_C(f(X_1,\ldots,X_m))$, so $t_i\rho \in M_C(X_i)$ and $t_i\rho = t_j\rho$ if $X_i = X_j$. Therefore, ρ solves $\{t_1 \in\!\!\!\!\_\ X_1,\ldots,t_n \in\!\!\!\!\_\ X_n\}$ and ρ solves the set of equations E, so ρ solves $U_o \cup \{t_1 \in\!\!\!\!\_\ X_1,\ldots,t_n \in\!\!\!\!\_\ X_n\} \cup E$.

"If": Assume that ρ solves $U' = U_o \cup \{t_1 \in\!\!\!\!\_\ X_1,\ldots,t_n \in\!\!\!\!\_\ X_n\} \cup E$. Then $t_i\rho = t_j\rho$ if $X_i = X_j$ and $t_i\rho \in M_C(X_i)$. Therefore, $f(t_1,\ldots,t_n)\rho \in M_C(f(X_1,\ldots,X_n))$, so ρ solves $U = U_o \cup f(t_1,\ldots,t_n) \in\!\!\!\!\_\ f(X_1,\ldots,X_n)$.

W4: "Only if": If ρ solves $U_o \cup \{t \in\!\!\!\!\_\ e_1 \sqcup \ldots \sqcup e_m\}$ then $t\rho \in M_C(e_1 \sqcup \ldots \sqcup e_m)$. Thus $t\rho \in M_C(e_i)$ for some e_i, so ρ also solves the new set $U_o \cup \{t \in\!\!\!\!\_\ e_i\}$.

"If": If ρ solves one of the new sets $U_o \in \{t \in\!\!\!\!\_\ e_i\}$ in $\mathcal{S}'$ for some i, $1 \leq i \leq m$, then $t\rho \in M_C(e_i)$ and so $t\rho \in M_C(e_1 \sqcup \ldots \sqcup e_m)$. Therefore, ρ also solves $U_o \cup \{t \in\!\!\!\!\_\ e_1 \sqcup \ldots \sqcup e_m\}$.

W5: For any ρ, $f(t)\rho \in M_C(f(X_1) \sqcup \ldots \sqcup f(X_n))$ iff $t\rho \in M_C(X_1 \sqcup \ldots \sqcup X_n)$. Therefore, ρ solves $U_o \cup \{f(t) \in\!\!\!\!\_\ f(X_1) \sqcup \ldots \sqcup f(X_n)\}$ iff ρ solves $U_o \cup \{t \in\!\!\!\!\_\ X_1 \sqcup \ldots \sqcup X_n\}$.

W6: Clearly, ρ solves $U_o \cup \{c \in\!\!\!\!\_\ e_1 \sqcup \ldots \sqcup e_n\}$ iff ρ solves U_o, given that some e_i is equal to c, $1 \leq i \leq n$. If e_i is $f_i(\cdots) \neq c$ for all i, $1 \leq i \leq n$, then this set has no solutions and can be removed from $\mathcal{S}$.

W7: Since $M_C(\top)$ is the set of all ground terms, any ground substitution solves

$$\{X_1 \in\!\!\!\!\_\ \top,\ldots,X_n \in\!\!\!\!\_\ \top\},$$

and any ground substitution solves $t \in\!\!\!\!\_\ \top \sqcup exp$. Therefore, both directions hold.

U1: Trivially, ρ solves $X = t$ iff ρ solves $t = X$, so both directions hold.

U2: Any substitution ρ solves $X = X$, so ρ solves $U_o \cup \{X = X\}$ iff ρ solves U_o.

U3: Trivially, again, ρ solves $U_o \cup \{f_1(s_1,\ldots,s_n) = f_2(t_1,\ldots,t_n)\}$ iff $f_1 = f_2$ (and so $m = n$) and ρ solves $U_o \cup \{s_1 = t_1,\ldots,s_n = t_n\}$; the set $U_0 \cup \{f_1(\cdots) = f_2(\cdots)\}$ has no solutions if $f_1 \neq f_2$, and so can be removed from $\mathcal{S}$ in this case.

U4: Given that X does not occur in t, ρ solves $\{X = t\}$ iff $s\rho = (s\,\{X \mapsto t\})\rho$ for any term s. Therefore, for any t_1 and t_2, ρ solves $\{X = t\} \cup \{t_1 = t_2\}$ iff ρ solves $\{X = t\} \cup \{t_1\,\{X \mapsto t\} = t_2\,\{X \mapsto t\}\}$. Similarly, for any SSC C, ρ solves $\{X = t\} \cup \{t_1 \in\!\!\!\!\_\ E\}$ given C iff ρ solves $\{X = t\} \cup \{t_1\,\{X \mapsto t\} \in\!\!\!\!\_\ E\}$ given C, since if $X\rho = t$, then $t_1\rho \in M_C(E)$ iff $t_1\,\{X \mapsto t\}\,\rho \in M_C(E)$.

If $X \in Var(t)$ and $X \neq t$, then no substitution can solve $U_o \cup \{X = t\}$, so this set can be removed from $\mathcal{S}$. ■

Figure 4.1 presents an algorithm that finds the solutions to a set of equations and membership constraints. These are simplified by an exhaustive application of transformations W1–W7 and U1–U4; we will show that if no transformation is applicable, then all the elements of $\mathcal{S}$ at the end of Part 1 are in normal form. A substitution and a shallow SSC can then be constructed from each element of $\mathcal{S}$; each constraint C_i constructed from U_i in Part 2 is a shallow SSC with no cycles. The set of pairs $\langle \theta_i, C_i \rangle$ will provide a correct and complete solution to the original problem. The variables in the equations and in the left hand side of the membership constraints are assumed to be disjoint from the variables in C.

<u>Input</u>: A set S of equations and membership constraints and an SSC C in solved form, where $Var(\mathrm{EQN}(S)) = Var(lhs(\mathrm{MEM}(S)))$, $Var(\mathrm{EQN}(S))$ and $Var(C)$ are disjoint, and C has no parameters.
<u>Output</u>: A set of pairs $\{\langle \theta_i, C_i \rangle\}$, for SSC's C_i and substitutions θ_i.
<u>Part 1</u>:
Let $\mathcal{S} = \{S\}$. Apply transformations W1–W7 and U1–U4 to $\mathcal{S}$ until none apply.

Part 2:
FOR EACH $U_i \in \mathcal{S}$
    Let C_i be the union of C and the set constraints $\langle X \equiv exp_1 \sqcap \ldots \sqcap exp_n \rangle$
        for each $X \in Var(lhs(\mathrm{MEM}(U_i)))$, where $\{X \underline{\in} exp_1, \ldots, X \underline{\in} exp_n\}$
        are the membership constraints for X in $\mathrm{MEM}(U_i)$.
    Let θ_i be $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, where $\{X_1 = t_1, \ldots, X_n = t_n\} = \mathrm{EQN}(U_i)$.

Return $\{\langle \theta_i, C_i \rangle\}$.

Figure 4.1: Solving algorithm

In the following discussion we assume that Part 1 of the solving algorithm terminates; we prove that this is the case in Section 4.1.1. Invariants (i) and (ii) of Lemma 4.6 are assumed to hold at the start of the algorithm, so they hold throughout Part 1.

Lemma 4.10 *For each pair $\langle \theta_i, C_i \rangle$ in the output of the solving algorithm of Figure 4.1,*

1. *θ_i is an idempotent substitution.*

2. *$Dom(\theta_i) \cap Dom(C_i) = \emptyset$.*

3. *$Dom(\theta_i) \cup (Dom(C_i) - Dom(C)) = Var(\mathrm{EQN}(S))$, and $Var(Rng(\theta_i)) \subseteq Dom(C_i) - Dom(C)$, where C is the input SSC and S is the input set of equations and membership constraints.*

4. *Each C_i is a shallow SSC with no cycles and no parameters, where $Rng(C_i)$ is in DNF.*

Proof: Let U_i be the set of equations and membership constraints in solved form corresponding to a pair $\langle \theta_i, C_i \rangle$ at the end of Part 1.

1. Since the solving transformations are exhaustively applied in Part 1 of the algorithm, then transformation U4, in particular, cannot be applied at the end of Part 1. Therefore, if $\{X = t\} \subseteq \mathrm{EQN}(U_i)$ for some $U_i \in S$ at the end of Part 1, then $X \notin \mathit{Var}(\mathit{rhs}(\mathrm{EQN}(U_i)))$ and $X \notin \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_I)))$. Therefore, $\mathit{Dom}(\theta_i) \cap \mathit{Var}(\mathit{Rng}(\theta_i)) = \emptyset$ (so θ_i is idempotent, by Lemma 1.1) and $\mathit{Dom}(\theta_i) \cap \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_i))) = \emptyset$.

2. We assume that $V = \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(S))) \cup \mathit{Var}(\mathrm{EQN}(S))$ is disjoint from $\mathit{Var}(C)$ and that $\mathit{Var}(\mathit{rhs}(\mathrm{MEM}(S))) \subseteq \mathit{Dom}(C)$, so by Lemma 4.6 this also holds for U_i. Therefore, $\mathit{Var}(\mathrm{EQN}(U_i))$ is disjoint from $\mathit{Var}(C)$, and so $\mathit{Dom}(\theta_i) = \mathit{Var}(\mathit{lhs}(\mathrm{EQN}(U_i)))$ is disjoint from $\mathit{Var}(C)$. $\mathit{Dom}(\theta_i)$ is also disjoint from $\mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_i)))$ (see proof for 1 above), so $\mathit{Dom}(\theta_i)$ and $\mathit{Var}(C_i)$ are also disjoint, since $\mathit{Var}(C_i) = \mathit{Var}(C) \cup \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_i)))$.

3. That $\mathit{Dom}(\theta_i) \cup (\mathit{Dom}(C_i) - \mathit{Dom}(C)) = \mathit{Var}(\mathrm{EQN}(S))$ follows from the fact that no transformation can remove or add any variables to $\mathrm{EQN}(U) \cup \mathit{lhs}(\mathrm{MEM}(U))$ for $U \in \mathcal{S}$.[6] Thus, $\mathit{Var}(\mathrm{EQN}(U_i)) \cup \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_i)))$ is equal to $\mathit{Var}(\mathrm{EQN}(S)) \cup \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(S)))$. The only way that a variable can be removed from $\mathit{lhs}(\mathrm{MEM}(U))$ for some $U \in \mathcal{S}$ during Part 1 of the algorithm is by an application of transformation U4; in this case, this variable is in $\mathit{Dom}(\theta_i)$, so it does not appear in $\mathit{Rng}(\theta_i)$. Therefore, $\mathit{Var}(\mathit{Rng}(\theta_i)) \subseteq \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(U_i))) = \mathit{Dom}(C_i) - \mathit{Dom}(C)$.

4. Each variable $X \in \mathit{lhs}(\mathrm{MEM}(U_i))$ does not appear in C, so there is exactly one equation for X in C_i, and by Part (i) of Lemma 4.6, each element of $\mathit{rhs}(\mathrm{MEM}(U_i))$ is a shallow atomic set expression. Therefore, all the elements in $\mathit{Rng}(C_i) - \mathit{Rng}(C)$ are shallow set expressions in DNF. C_i contains no parameters, since C is assumed to contain no parameters and $\mathit{Var}(\mathit{rhs}(\mathrm{MEM}(U))) \subseteq \mathit{Dom}(C)$ by Lemma 4.6. Finally, C_i contains no cycles, since C is assumed to have no cycles and the new variables (i.e. $\mathit{Dom}(C_i) - \mathit{Dom}(C)$) do not appear in $\mathit{Rng}(C)$. ∎

Lemma 4.11 *Let $\mathcal{S}$ be a set of sets of equations and membership constraints and $\mathcal{S}'$ be the result of applying any of transformations* W1–W7 *or* U1–U4*. Then if a substitution θ is a unifier of* $\mathrm{EQN}(U')$ *for some $U' \in \mathcal{S}'$, then θ is a unifier of* $\mathrm{EQN}(U)$ *for some $U \in \mathcal{S}$.*

Proof: Let $U \in \mathcal{S}$ be the set to which a transformation is applied. If a transformation removes U from $\mathcal{S}$ the result trivially holds; otherwise, it is sufficient to show that the unifiers of $\mathrm{EQN}(U')$ for any new sets U' are also unifiers of $\mathrm{EQN}(U)$. Transformations W1, W2, W4–W7 do not change the set of equations in U (i.e. $\mathrm{EQN}(U') = \mathrm{EQN}(U)$ for all new sets U'), so the result holds. Transformations U1–U4, being equivalent to the corresponding ones for normal unification as far as $\mathrm{EQN}(U)$ is concerned (see Section 1.7), preserve all the solutions for $\mathrm{EQN}(U)$. Consider now transformation W3: It adds a set of equations S to U, but does not change any of the existing equations in $\mathrm{EQN}(U)$. Therefore, if θ solves $\mathrm{EQN}(U') = \mathrm{EQN}(U) \cup S$, then θ solves $\mathrm{EQN}(U)$.

[6] This includes transformation U2, which removes equations of the form $X = X$, since when it is applied X has not yet been "isolated" by transformation U4, and thus must still appear in $\mathit{lhs}(\mathrm{MEM}(U))$, since we assume that that $\mathit{Var}(\mathrm{EQN}(S)) = \mathit{Var}(\mathit{lhs}(\mathrm{MEM}(S)))$.

Corollary 4.12 *For each pair $\langle \theta_i, C_i \rangle$ in the output of the solving algorithm of Figure 4.1, the substitution θ_i is a unifier of* EQN(S)*, where S is the input set of equations and membership constraints.*

Proof: By Lemma 4.11, if a substitution unifies EQN(U_i) for some set U_i at the end of Part 1 of the algorithm, then it also unifies EQN(S). Since θ_i (trivially) unifies EQN(U_i), then θ_i unifies EQN(U_i). ■

The following theorem shows that the output substitution/constraint pairs always generate correct solutions to the initial set of membership constraints:

Theorem 4.13 *For each pair $\langle \theta_i, C_i \rangle$ in the output of the algorithm in Figure 4.1, $M_{C_i}(t\theta_i) \subseteq M_C(exp)$ for all membership constraints $t \mathrel{\underline{\in}} exp$ in the input set S.*

Proof: Let $s \in M_{C_i}(t\theta_i)$. Then there is a substitution ρ over the variables in t such that $X\rho \in M_{C_i}(X)$ for all $X \in Dom(C_i)$ and $t\rho = s$. Thus $X\rho \in M_C(exp)$ if $\{X \mathrel{\underline{\in}} exp\} \in U_i$ for the corresponding set U_i at the end of Part 1 of the algorithm, from the construction of C_i. We can extend ρ so that $X\rho = t\rho$ if $X\theta_i = t$ for all variables X, since $Dom(\theta_i)$ is disjoint from $Var(lhs(\mathrm{MEM}(U_i)))$ and by Lemma 4.10, $Var(Rng(\theta_i)) \subseteq Dom(C_i)$ and $X\theta_i = X$ if $X \in Var(lhs(\mathrm{MEM}(U_i)))$. Therefore, ρ solves U_i, so by Theorem 4.9, ρ also solves the initial set S. Therefore, $t\rho \in M_C(exp)$ if $t \mathrel{\underline{\in}} exp$ is a membership constraint in S. But $t\rho = s$, so $s \in M_C(exp)$. Therefore, $M_{C_i}(t\theta_i) \subseteq M_C(exp)$. ■

The following is a simple observation true of all SSC's C:

Lemma 4.14 *Let θ and ρ be substitutions and let C be an* SSC*. If $X\rho \in M_C(X\theta)$ for all $X \in Var(t)$, then $t\rho \in M_C(t\theta)$.*

Proof: $M_C(s)$ is the set of all terms $t\Omega$, where $V\Omega \in M_C(V)$ for all $V \in Var(s)$. Therefore, if $X\rho \in M_C(X\theta)$ for all $X \in Var(t)$, then $X\rho = X\theta\Omega$ for some substitution Ω such that $V\Omega \in M_C(V)$ for all $V \in Var(t\theta)$. Therefore $t\rho = t\theta\Omega\rho \in M_C(t\theta)$. ■

The following theorem shows that all possible solutions for the input are covered by the output, and that all the solutions generated by the output are correct solutions of the input. That is, $\bigcup_i M_{C_i}(t\theta_i)$ is exactly equal to the set of all the instances of t that can belong to $M_C(exp)$ for each $t \mathrel{\underline{\in}} exp$ in the input:

Theorem 4.15 *Let S be the set of equations and membership constraints input to the solving algorithm of Figure 4.1. Then a substitution ρ solves S if, and only if, for some output pair $\langle \theta_i, C_i \rangle$, $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(\mathrm{EQN}(S))$.*

Proof: "Only if": Assume that ρ solves the initial set S. Then, by Theorem 4.9, ρ solves some element $U_i \in \mathcal{S}$ at the end of Part 1. Let $\langle \theta_i, C_i \rangle$ be the pair constructed from U_i. Then $X\rho = X\theta_i\rho$ for all $X \in Dom(\theta_i)$ and $X\rho \in [X/C_i]$ for all $X \in Var(lhs(\mathrm{MEM}(U_i))) = Dom(C_i) - Dom(C)$. By Lemma 4.10, $Var(Rng(\theta_i)) \subseteq Dom(C_i) - Dom(C)$ and $Dom(\theta) \cap Dom(C_i) = \emptyset$, so $X\theta_i = X$ for all $X \in Dom(C_i) - Dom(C)$. We know that $Dom(\theta_i) \cup (Dom(C_i) - Dom(C)) = Var(\mathrm{EQN}(S))$, so $X\rho = X\theta_i\rho$ and $X\theta_i\rho \in M_{C_i}(x\theta_i)$ for all $X \in Var(\mathrm{EQN}(S))$. Therefore $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(\mathrm{EQN}(S))$.

"If": Assume that $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(\mathrm{EQN}(S))$. Then for all $X \in Dom(\theta_i)$, $X\theta_i\rho = X\rho$, so ρ solves EQN(U_i) for some U_i at the end of Part 1. Therefore, by Lemma 4.11, ρ solves EQN(S).

Let $t \Subset exp$ be a membership constraint in S. If $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(t)$, then $t\rho \in M_{C_i}(t\theta_i)$, by Lemma 4.14, and for all $X \in Dom(\theta_i)$, $X\rho = X\theta_i\rho$. Therefore $t\rho \in M_{C_i}(t\theta_i) \subseteq M_C(exp)$, by Theorem 4.13, so $t\rho \in M_C(exp)$.

Therefore, ρ solves $\mathrm{EQN}(S) \cup \mathrm{MEM}(S)$, that is, ρ solves S. ∎

4.1.1 Termination

Theorem 4.16 *Given that the input* SSC *C is in solved form, the solving algorithm of Figure* 4.1 *terminates.*

Proof: Note that if the input SSC is in solved form, then (by definition) it has no cycles. Clearly, Part 2 of the algorithm terminates, given that $\mathcal{S}$ is finite, so we focus only on Part 1: We prove termination by defining an ordering $\prec$ on the elements of $\mathcal{S}$ that is well-founded, such that the induced *multiset ordering* (see [Dershowitz and Manna, 1979]) is well-founded and is always reduced by all transformations. The ordering $\prec$ is based on the 6-tuple $\langle V, F_E, F_M, D, O, Z\rangle$, whose elements are defined as follows:

V: The number of different variables occurring in $lhs(\mathrm{MEM}(U))$ and $rhs(\mathrm{EQN}(U))$ — that is, $|\,Var(lhs(\mathrm{MEM}(U)) \cup rhs(\mathrm{EQN}(U)))|$.

F_E: The total number of function symbol occurrences in $lhs(\mathrm{EQN}(U))$.

F_M: The total number of function symbol occurrences in $lhs(\mathrm{MEM}(U))$.

P: The sum over all $e_i \in rhs(\mathrm{MEM}(U))$ of $|\mathcal{P}(e_i, C)|$, where $\mathcal{P}(exp, C)$ is the set that contains exp and all the set expressions that can be obtained by repeatedly replacing an un-nested occurrence of a variable Y in exp by $C(Y)$ (see Definition 3.11 and Theorem 3.12). Since C has no cycles, $\mathcal{P}(exp, C)$ is always finite, and so is P. Clearly, P is reduced each time transformation W2 replaces an un-nested variable X by $C(X)$ in the right hand side of a membership constraint.

For example, if C is the SSC $\{\langle X \equiv Y \sqcup a\rangle,\ \langle Y \equiv f(X) \sqcup Z\rangle,\ \langle Z \equiv b \sqcup f(Y)\rangle\}$ and e_i is the set expression $X \sqcup d$, then

$$\mathcal{P}(e_i, C) = \{X \sqcup d,\ Y \sqcup a \sqcup d,\ f(X) \sqcup Z \sqcup a \sqcup d,\ f(X) \sqcup b \sqcup f(Y) \sqcup a \sqcup d\}\,.$$

If X is replaced by $C(X)$ in e_i yielding e_i', then $\mathcal{P}(e_i', C) = \mathcal{P}(e_i, C) - \{e_i\}$.

P is always reduced by transformation W2, unchanged by transformation W4, never increased (and possibly reduced) by transformations W1, W6 and W7, and possibly increased by transformations W3 and W5. Clearly, it is not changed by transformations U1–U4, since these do not change $rhs(\mathrm{MEM}(U))$.

O: The number of occurrences of the symbol "$\sqcup$" in $rhs(\mathrm{MEM}(U))$.

Z: The size of U (that is, the number of equations and membership constraints in U).

The usual left-to-right lexicographic ordering $\prec$ on these tuples is well-founded, since all 6 components are always non-negative integers. The ordering $\prec_M^*$ used for $\mathcal{S}$ is the multiset ordering induced by $\prec$, as follows: $\prec_M^*$ is the transitive closure of $\prec_M$, where $\mathcal{S}_2 \prec_M \mathcal{S}_1$ for multisets $\mathcal{S}_1$ and $\mathcal{S}_2$ if, and only if, $\mathcal{S}_2$ is the result of replacing an element e of $\mathcal{S}_1$ by a finite number of elements that are strictly smaller than e in the ordering $\prec$ (this includes removing e from $\mathcal{S}_1$). If $\prec$ is well-founded, then the induced multiset ordering $\prec_M^*$ is also well-founded [Dershowitz and Manna, 1979].

We now show that each transformation removes an element U in $\mathcal{S}$ or else replaces U with a finite number of new elements that are strictly smaller than U in the ordering $\prec$.

No transformation can increase V, since no transformation adds new variables to $\mathrm{EQN}(U) \cup lhs(\mathrm{MEM}(U))$. Transformations W1–W7 do not change $\mathrm{EQN}(U)$, so they cannot change F_E. Similarly, transformations U1–U3 cannot change F_M, since they do not change $\mathrm{MEM}(U)$.

- When transformation W1 replaces $t \Subset g(\cdots) \sqcup exp$ by $t \Subset exp$, O is reduced. Otherwise, the transformation removes an element of $\mathcal{S}$. This transformation never changes any of the other values in the tuple.
- Transformation W2 reduces P by replacing a variable X in $rhs(\mathrm{MEM}(U))$ by $C(X)$, and does not change V, F_M or F_E, since it does not change $\mathrm{EQN}(U)$ or $lhs(\mathrm{MEM}(U))$.
- Transformation W3 always reduces F_M, since a term $f(t_1, \ldots, t_n)$ in $lhs(\mathrm{MEM}(U))$ is replaced by its subterms $\{t_1, \ldots, t_n\}$.
- Transformation W4 adds a finite number of new pairs in which O, the number of "$\sqcup$" symbols in $rhs(U)$, is reduced, and for which all other values in the tuple remain unchanged.
- Transformation W5 reduces F_M by replacing a term $f(X)$ in $lhs(\mathrm{MEM}(U))$ by X.
- Transformation W6 reduces F_M by removing an occurrence of a constant $c \in lhs(\mathrm{MEM}(U))$ or else removes an element from $\mathcal{S}$.
- Transformation W7 always reduces F_M by replacing a non-variable term in $lhs(\mathrm{MEM}(U))$ by its variable subterms.
- Transformation U1 reduces F_E by replacing a non-variable term in $lhs(\mathrm{EQN}(U))$ by a variable.
- Transformation U2 reduces Z by removing the equation $\{X = X\}$ from U, and cannot change any of the other elements of the tuple.[7]
- Transformation U3 reduces F_E by replacing a non-variable term in $lhs(\mathrm{EQN}(U))$ by its subterms or else removes an element from $\mathcal{S}$.
- Transformation U4 removes an element from $\mathcal{S}$ or else reduces V, since the variable X no longer appears in $rhs(\mathrm{EQN}(U)) \cup lhs(\mathrm{MEM}(U))$.

Transformation W4 can only add a finite number of new elements to $\mathcal{S}$, whose measure is strictly less than that of their parent, so the algorithm always terminates with a finite number of solutions. Table 4.1 shows a schematic representation of the termination proof. A "$<$" entry indicates that the corresponding measure always decreases, and an "$=$" entry indicates that it

[7] This transformation could in general also reduce V, but under our assumptions this is never the case.

Transformation	V	F_M	F_E	P	O	Z
W1	=	=	=	=	<	—
W2	=	=	=	<	—	—
W3	=	<	—	—	—	—
W4	=	=	=	=	<	—
W5	=	<	=	—	—	—
W6	=	<	—	—	—	—
W7	=	=	=	=	<	—
U1	=	=	<	—	—	—
U2	=	=	=	=	=	<
U3	=	=	<	—	—	—
U4	<	—	—	—	—	—

Table 4.1: Termination of solving transformations

never increases. An "—" entry indicates that whether this variable increases or not does not affect the termination proof. Note that when traversing any row in Table 4.1 from left to right, a "<" entry always precedes the "—" ones. ■

There are many ways in which the efficiency of the solving process can be improved. For example, we could introduce a variant of transformation W4 in the case that the left hand side of the membership constraint $t \in exp$ is a ground term; in this case, there is no need to split the partial solution into a set of new ones; instead, a subroutine could be used to check that the t is indeed an element of $M_C(exp)$.[8] If this is the case, the membership constraint $t \in exp$ can be removed; otherwise, the set containing it can be removed from $\mathcal{S}$, since it has no solutions.

To avoid duplication of work and detect failed branches early, transformation W4 should be applied only after all others have been exhausted, and if this transformation can be applied to more than membership constraint in a given set, then the constraint with the smallest number of disjuncts should be chosen. This minimizes the branching at the early stages, increasing the likelihood of early failure and reducing unnecessary work. Transformations U1–U4, on the other hand, should be given the highest priority, so that they are always applied exhaustively after transformation W3. In this way, the equations in EQN(U) always define an idempotent substitution, whose domain is disjoint from the variables in lhs(MEM(U)) and whose range contains only variables appearing in lhs(MEM(U)).

Finally, this algorithm is also simpler if the input SSC C is linear; in this case, no new equations are ever added to the sets in $\mathcal{S}$, so the substitution θ_i is always a most general unifier for the equations in the input set.[9] All the output SSC's are also linear, in this case.

4.2 Unifying Constrained Terms

Given a constrained set of terms $S = \{t_1, \ldots, t_n\}/C$, unifying S is equivalent to finding all the solutions to the constrained equation set $\{t_1 = t_2, \ldots, t_{n-1} = t_n\}/C$.

[8] This could be done efficiently using specialized versions of transformations W1–W7 and U1–U4.

[9] If there are no equations in the input set, then each θ_i is always the identity, if C is linear.

Given a constrained equation set E/C, we want to find all possible ways to strengthen the constraints on the variables in $Var(E)$ such that all the equalities hold. That is, we want to characterize all the substitutions ρ that solve E such that $X\rho \in M_C(X)$ for all $X \in Var(E)$. Note that unifying a constrained set of variable-disjoint terms should yield a representation of all the common instances of the terms in the set.

Figure 4.2 presents the unification algorithm for constrained equation sets.

<u>Input</u>: A constrained equation set S/C, where C is a parameter-free SSC in solved form.
<u>Output</u>: A set of pairs $\{\langle\theta_i, C_i\rangle\}$ where θ_i is a substitution and C_i is a parameter-free SSC in solved form.

Step 1: Let π be a renaming substitution that maps $Var(S)$ into variables not in $Var(C)$.

Let $U := S\pi \cup \{X\pi \mathrel{\underline{\in}} X \mid X \in Var(S)\}$, and let $\mathcal{T} = \{\langle\theta_i, C_i\rangle\}$ be the result of solving the set of equations and membership constraints U given C, using the algorithm in Figure 4.1.

Step 2: Rewrite each SSC C_i in $\mathcal{T}$ into solved form, using the algorithm in Figure 3.1.

If at any point $M_{C_i}(X) = \emptyset$ for some $X \in Var(Rng(\pi))$, then remove the pair $\langle\theta_i, C_i\rangle$ from $\mathcal{T}$.

Step 3: For each pair in $\mathcal{T}$, replace θ_i by $\theta_i' := \pi \cdot \theta_i$, and return the new $\mathcal{T}$.

Figure 4.2: SSC unification algorithm

It should be clear that the algorithm terminates, given that its constituent sections do: In Step 1, the input for the solving algorithm is in the required form, since we assume that C is in solved form and $Var(S) \subseteq Dom(C)$. Thus the solving procedure terminates, and the SSC's C_i in $\mathcal{T}$ are shallow. Thus, rewriting the SSC's into solved form in Step 2 also terminates. Step 3, trivially, terminates too, so the entire algorithm runs in finite time.

The following theorem shows that the unification algorithm is both *correct* and *complete*:

Theorem 4.17 *A substitution ρ solves the constrained equation set S/C if, and only if, for some pair $\langle\theta_i', C_i\rangle$ in the output, $X\rho \in M_{C_i}(X\theta_i')$ for all $X \in Var(S)$.*

Proof: "Only if": Assume that ρ solves the input set of equations S. Therefore, $t_1\rho = t_2\rho$ if $t_1 = t_2$ is an equation in S and $X\rho \in M_C(X)$ for all $X \in Var(S)$. Let $\rho' = \pi^{-1}\rho$; since $X\pi\rho' = X\rho \in M_C(X)$ for all $X \in Var(S)$, ρ' solves the set $U = S\pi \cup \{X\pi \mathrel{\underline{\in}} X \mid X \in Dom(S)\}$ at the start of Step 1. Theorem 4.15 shows that $X\rho' = X\theta_i\rho' \in M_{C_i}(X\theta_i)$ for some pair $\langle\theta_i, C_i\rangle$ in $\mathcal{T}$ at the end of Step 1, for all $X \in Var(lhs(U))$.

Note that $X\rho' \neq \emptyset$ for any $X \in Var(lhs(U)) = Var(Rng(\pi))$ by hypothesis, so the pair in $\mathcal{T}$ containing C_i cannot be removed in Step 2. Rewriting each C_i into solved form preserves M_{C_i} for each C_i over the variables in $Dom(C_i)$, as shown in Chapter 3, so the above is still true if we take C_i to be the solved form SSC at the end of Step 2. By Lemma 4.14, if $Var(t) \subseteq Var(lhs(U))$, then

$t\rho' \in M_{C_i}(t\theta_i)$. Since $Var(X\pi) \subseteq Var(lhs(U))$, then $X\pi\rho' \in M_{C_i}(X\pi\theta_i)$ for all $X \in Var(S)$. But $X\pi\rho' = X\rho$, and $X\pi\theta_i = X\theta_i'$, since $\theta_i' = \pi\theta_i$, so $X\rho \in M_{C_i}(X\theta_i')$, for all $X \in Var(S)$.

"If": Assume that $X\rho \in M_{C_i}(X\theta_i') = M_{C_i}(X\pi\theta_i)$ for all $X \in Var(S)$, for some output pair $\langle\theta_i, C_i\rangle$. Again, let $\rho' = \pi^{-1}\rho$, so that $X\rho = X\pi\rho'$ and $X\rho' \in M_{C_i}(X\theta_i)$ for all $X \in Var(S)$. This is still true if we take M_{C_i} to be the corresponding SSC at the end of Step 2, since rewriting into solved form yields an SSC equivalent over the original domain. Therefore, by Theorem 4.15, ρ' solves the set of equations and membership constraints $U = S\pi \cup \{X\pi \mathrel{\underline{\in}} X \mid X \in Var(S)\}$ at the beginning of Step 1, so ρ solves the initial equation set S/C. ■

Note that as a corollary of Theorem 4.17 above, the unification algorithm returns $\emptyset$ if the input constrained equation set has no solutions. Another consequence of Theorem 4.17 is that the algorithm behaves as expected when asked to unify a single pair of variable-disjoint terms:

Corollary 4.18 *Let $S = \{t_1 = t_2\}$ and the* SSC *C in solved form be the input to the main algorithm, where $Var(t_1) \cap Var(t_2) = \emptyset$, and let $\{\langle\theta_1, C_1\rangle, \ldots, \langle\theta_n, C_n\rangle\}$ be the output. Then*

$$M_C(t_1) \cap M_C(t_2) = \bigcup_{i=1}^{i=n} M_{C_i}(t_1\theta_i) = \bigcup_{i=1}^{i=n} M_{C_i}(t_2\theta_i).$$

Proof: It suffices to show that $M_C(t_1) \cap M_C(t_2) = \bigcup_{i=1}^{i=n} M_{C_i}(t_1\theta_i)$, since $t_1\theta_i = t_2\theta_i$ for all i.

If $s \in M_C(t_1) \cap M_C(t_2)$ and $Var(t_1) \cap Var(t_2) = \emptyset$, then there is some substitution ρ such that $t_1\rho = t_2\rho = s$ and $X\rho \in M_C(X)$ for all $X \in Var(S)$. By Theorem 4.17, this is the case iff $X\rho \in M_{C_i}(X\theta_i)$ for some i, which means that $s \in M_{C_i}(t_1\theta_i)$.

The other direction is basically the same: θ_i is a unifier of t_1 and t_2, so $t_1\theta = t_2\theta$. Therefore, if $s \in M_{C_i}(t_1\theta_i)$, then there exists some substitution ρ such that $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(t_1) \cup Var(t_2) = Var(S)$. By Theorem 4.17, then $s = t_1\rho = t_2\rho$ and $X\rho \in M_C(X)$ for all $X \in Var(S)$, so $s \in M_C(t_1) \cap M_C(t_2)$. ■

We say that the pairs of substitutions and SSC's output by the algorithm of Figure 4.2 are SSC-*solutions* of the input constrained set of equations, and define SSC-*unifiers* accordingly:

Definition 4.19 (SSC-**unifiers**) *Let S/C be a constrained set of terms where $S = \{t_1, \ldots, t_n\}$ and C is a solved form* SSC*. An* SSC-unifier *of S/C is a pair $\langle\theta_i, C_i\rangle$ obtained by solving the constrained equation set $\{t_1 = t_2, \ldots, t_{n-1} = t_n\}/C$ using the unification algorithm of Figure* 4.2.

An SSC-unifier $\langle\theta_i, C_i\rangle$ applied to a constrained term t/C results in the new constrained term $t\theta_i/C_i$, where $C \subseteq C$. Note that $t\theta_i/C_i$ is admissible, given that t/C is also admissible, since we explicitly require that no variable in the range of θ_i be empty in the new SSC C_i. In Chapter 7, this guarantees the admissibility of all the clauses constrained by SSC's that can be used in deriving an empty clause from a set of admissible clauses, thus ensuring the *soundness* of the corresponding deductive system.

The set of SSC-unifiers output by the algorithm of Figure 4.2 is not guaranteed to be minimal, in the sense that it could be the case that all the (ground) solutions covered by one of the output pairs are covered by another. The problem of extracting a set of incomparable SSC-unifiers is decidable, as a consequence of the results in [Bogaert and Tison, 1991]; how to do so efficiently is left open, as well as the problem of improving the efficiency of the entire algorithm (see Chapter 8).

It might be possible, for instance, to merge the solving procedure of Figure 4.1 with the simplification algorithms of Chapter 3, so that empty variables are detected and simplified as the solving proceeds. This also would encourage the early detection of failure. For example, every time a membership constraint is simplified to be of the form $X \in exp$, we could check that the outermost function symbol of a disjunct in exp matches that of of some disjunct in exp' for all other membership constraints $X \in exp'$ in that set. If this is not the case, X is an empty variable in the corresponding output SSC, so this set can be removed.

4.3 Unification under Shallow SSC's is $\mathcal{NP}$-hard

The "yes/no" version of the unification problem for SSC's is: Does there exist a solution for a constrained equation set S/C, for an SSC C? Schmidt-Schauß [1989] proves that unification is $\mathcal{NP}$-hard for elementary sorted signatures. We can rewrite this proof in terms of unification under SSC's, showing that unification is also $\mathcal{NP}$-hard for terms constrained by shallow SSC's. The proof is a simple reduction of the boolean satisfiability problem to a constraint unification one, as follows:

Consider the shallow SSC C_Σ in solved form shown in Figure 4.3, where "*and*" and "*or*" are binary functions, "*not*" is unary, and T and F are constants.

$$\left\{\begin{array}{lcl} \langle\Upsilon_{bool} & \equiv & \Upsilon_{true} \sqcup \Upsilon_{false}\rangle, \\ \langle\Upsilon_{true} & \equiv & and(X_T, Y_T) \sqcup or(\Upsilon_{true}, \Upsilon_{bool}) \sqcup or(\Upsilon_{bool}, \Upsilon_{true}) \sqcup not(\Upsilon_{false}) \sqcup T\rangle, \\ \langle\Upsilon_{false} & \equiv & and(\Upsilon_{false}, \Upsilon_{bool}) \sqcup and(\Upsilon_{bool}, \Upsilon_{false}) \sqcup or(X_F, Y_B) \sqcup not(\Upsilon_{true}) \sqcup F\rangle, \\ \langle X_T & \equiv & \Upsilon_{true}\rangle, \\ \langle Y_T & \equiv & \Upsilon_{true}\rangle, \\ \langle X_F & \equiv & \Upsilon_{false}\rangle, \\ \langle Y_F & \equiv & \Upsilon_{false}\rangle \end{array}\right\}$$

Figure 4.3: A shallow SSC for boolean logic

For any arbitrary boolean expression e over a set of variables $\{V_1, \ldots, V_n\}$, we can construct an equivalent term e' using *and*, *or*, *not*, T and F and build the SSC

$$C = \{\langle V_1 \equiv \Upsilon_{bool}\rangle, \ldots, \langle V_n \equiv \Upsilon_{bool}\rangle\} \cup \{\langle N \equiv \Upsilon_{true}\rangle\}$$

in time polynomial in the size of e. It is easy to show that the original boolean expression e is satisfiable iff e' and N unify under $C \cup C_\Sigma$, that is, if the constrained equation set $\{e' = N\}/(C \cup C_\Sigma)$ has any solutions. In fact, the unification algorithm of Figure 4.2 will find all the truth value assignments for $\{V_1, \ldots, V_n\}$ that make e true — if any exist; variables that could be either true or false are restricted to Υ_{bool}.

Note that the SSC in Figure 4.3 is *linear*, so this shows unification to be $\mathcal{NP}$-hard under linear SSC's, in particular. The sorted signature corresponding to C_Σ is used by Schmidt-Schauß [1989] to show that sorted unification is $\mathcal{NP}$-complete for *elementary* sorted signatures, where all term declarations are of the form $f(V_1, \ldots, V_n)$ for variables V_i, where $V_i \neq V_j$ if $i \neq j$, and the signature is *regular* (see Section 5.5).

4.4 Example

We can use the "boolean signature" from the previous section to give an example of how the new unification works. Let C_Σ be the SSC of Figure 4.3 above, and consider solving the equation set

$$\{and(X, not(Y)) = Z\}$$

given the SSC

$$C = \{\langle X \equiv \Upsilon_{bool}\rangle,\ \langle Y \equiv \Upsilon_{bool}\rangle,\ \langle Z \equiv \Upsilon_{true}\rangle\} \cup C_\Sigma.$$

After renaming the variables in the terms to be unified with the substitution

$$\pi = \{X \mapsto x',\ Y \mapsto y',\ Z \mapsto z'\},$$

the set of equations and membership constraints to be solved is:

$$\{z' = and(x', not(y')),\ x' \mathrel{\underline{\in}} X,\ y' \mathrel{\underline{\in}} Y,\ z' \mathrel{\underline{\in}} Z\}.$$

By isolating the variable z' and replacing Z with $C(Z) = \Upsilon_{true}$, we get:

$$\{z' = and(x', not(y')),\ and(x', not(y')) \mathrel{\underline{\in}} \Upsilon_{true},\ x' \mathrel{\underline{\in}} X,\ y' \mathrel{\underline{\in}} Y\}.$$

After we replace Υ_{true} by

$$C_\Sigma(\Upsilon_{true}) = and(X_T, Y_T) \sqcup or(\Upsilon_{true}, \Upsilon_{bool}) \sqcup or(\Upsilon_{bool}, \Upsilon_{true}) \sqcup not(\Upsilon_{false}) \sqcup T,$$

there is only one disjunct that matches the left hand side of the membership constraint, namely $and(X_T, Y_T)$, so we get:

$$\{z' = and(x', not(y')),\ and(x', not(y')) \mathrel{\underline{\in}} and(X_T, Y_T),\ x' \mathrel{\underline{\in}} X,\ y' \mathrel{\underline{\in}} Y\} \Longrightarrow$$

$$\{z' = and(x', not(y')),\ x' \mathrel{\underline{\in}} X_T,\ not(y') \mathrel{\underline{\in}} Y_T,\ x' \mathrel{\underline{\in}} X,\ y' \mathrel{\underline{\in}} Y\}.$$

Now we replace Y_T in the membership constraint $not(y') \mathrel{\underline{\in}} Y_T$ by $C_\Sigma(Y_t)$, which is Υ_{true}. This is then replaced by the same expression as above, and again there is only one disjunct that matches:

$$\{z' = and(x', not(y')),\ x' \mathrel{\underline{\in}} X_T,\ not(y') \mathrel{\underline{\in}} not(\Upsilon_{false}),\ x' \mathrel{\underline{\in}} X,\ y' \mathrel{\underline{\in}} Y\}$$

We then obtain a set of equations and membership constraints in normal form:

$$\{z' = and(x', not(y')),\ x' \mathrel{\underline{\in}} X_T,\ y' \mathrel{\underline{\in}} \Upsilon_{false},\ x' \mathrel{\underline{\in}} x,\ y' \mathrel{\underline{\in}} Y\}$$

Part 1 of the algorithm is now over; note that we have a single set, since the partial solution was never split by transformation W3. In Part 2, we have to simplify the following SSC:

$$\left\{\ \langle x' \equiv X_T \sqcap X\rangle,\ \langle y' \equiv \Upsilon_{false} \sqcap Y\rangle,\ \langle X \equiv \Upsilon_{bool}\rangle,\ \langle Y \equiv \Upsilon_{bool}\rangle.\ \right\} \cup C_\Sigma.$$

It should be easy to see that this reduces to

$$C_1 = \Big\{ \; \langle x' \equiv \Upsilon_{true} \rangle, \; \langle y' \equiv \Upsilon_{false} \rangle, \; \langle X \equiv \Upsilon_{bool} \rangle, \; \langle Y \equiv \Upsilon_{bool} \rangle \; \Big\} \cup C_\Sigma .$$

Therefore, we obtain a single pair $\langle \theta_1, C_1 \rangle$ at the end of Part 2, where

$$\theta_1 = \{ z' \mapsto and(x', not(y')) \} .$$

Composing the original renaming substitution π and θ_1, we finally obtain the ouput pair

$$\langle \{ Z \mapsto and(x', not(y')), \; X \mapsto x', \; Y \mapsto y' \} , \; C_1 \rangle ,$$

which represents all the solutions to the initial constrained equation set.

Consider now changing the original unification problem above by replacing the set constraint $\langle X \equiv \Upsilon_{bool} \rangle$ by $\langle X \equiv \Upsilon_{false} \rangle$. Part 1 would be exactly the same; however, in Part 2 the unification process fails, since $X \sqcap X_T$ is simplified to "$\perp$".

Chapter 5

Sorted Logic

As discussed in Chapter 1, there are many different formulations of sorted logic. This chapter reviews two of the most general approaches: The first specifies sorts through the use of monadic constraints that refer to a background *sort theory*, while the second defines a *sorted signature* that describes well-sorted and ill-sorted terms. Our presentation of sort theories is based on the main example used by Frisch [1991], while the description of sorted signatures is based on the work of Schmidt-Schauß [1989].[1] The terminology and results in this chapter provide the background for showing, in Chapter 6, how the unification algorithm of Chapter 4 can be used to solve sorted unification problems under these two approaches.

Section 5.1 describes the sort theory formulation, and Section 5.2 describes sorted signatures. Section 5.3 discusses the *relativization* of sorted axiomatizations in clausal form, under both approaches. Section 5.4 shows how unification under sorted signatures can be seen as a particular case of unification under a sort theory. Finally, Section 5.5 discusses previous results for sorted unification.

See [Cohn, 1989b] for an overview of other dimensions among which sorted logics can vary. Again, we note that this discussion applies only to sorted unification for uninterpreted terms, without a background equational theory.

5.1 Sort Theories

In the logic presented by Frisch [1991], sorts are defined as unary predicates in a background logical theory, as opposed to an algebraic specification of "well-sorted terms." Given the *sort theory*, variables are constrained to range over subsets of the domain. Let $\mathcal{P}$ be the set of predicates in the main language, and let $\mathcal{S}_\Sigma$ be a finite set of unary predicates, disjoint from $\mathcal{P}$. The elements of $\mathcal{S}_\Sigma$ are the *sort predicates*, also called *sort symbols*.

Definition 5.1 (Sort Theory) *A* sort theory *is a set of FOPC sentences whose predicates are included in the set $\mathcal{S}_\Sigma$.*

The sort predicates can only appear in the sort theory. The only syntactic difference between the sorted language and an unsorted one is the use of *sorted variables*: A unique sort S is associated with each variable in $\mathcal{V}$, such that there are countably infinitely many variables

[1] Note that the deductive system presented in [Schmidt-Schauß, 1989] can also be described as "substitutional."

associated with each sort. We write $x{:}S$ to denote a variable of sort S.[2] For notational convenience, two sorted variables $x_1{:}r_1$ and $x_2{:}r_2$ will always be *different* variables if either $r_1 \neq r_2$ or $x_1 \neq x_2$.

All the usual definitions of terms, formulas and substitutions still apply. Substitutions are defined exactly as in Section 1.5, and all the definitions and lemmas presented there still hold. A **sorted axiomatization** is simply a set of FOPC sentences in the sorted language (i.e. whose variables are sorted variables). We should emphasize that the predicates that appear in the sort theory do not appear as such in the main language; the connection between the axiomatization and the sort theory is made only through the sorted variables.

Any model of a set of sorted axioms has to satisfy the sort theory Σ associated with it; that is, it should make all the sentences in Σ true. For such a model M and a sort S, $[\![S]\!]^M$ is a set of objects in the domain. A restricted variable $x{:}S$ is semantically constrained to range over the set denoted by S. The "null" sort always denotes the entire domain.

In Section 5.4 we show that traditional order-sorted logic without equality can be seen as an instance of the theory framework. Normal unsorted logic can also be thought of as an instance of this case, where there is only one universal sort to which all objects in the domain belong.

As Frisch [1991] points out, this is all "syntactic sugar," and does not increase the expressive power of the language: The **normalization** ψ^N of a sorted formula ψ in prenex normal form can be recursively defined as follows:

- If ψ contains no restricted variables, $\psi^N = \psi$.
- If ψ is $\forall x{:}S(\Gamma)$, then $\psi^N = \forall y \left(S(y) \rightarrow (\Gamma\{x{:}S \mapsto y\})^N\right)$ for a new variable y.
- If ψ is $\exists x{:}S(\Gamma)$, then $\psi^N = \exists y \left(S(y) \wedge (\Gamma\{x{:}r \mapsto y\})^N\right)$ for a new variable y.

As a special case, if ψ is a clause and $Var(\psi) = \{x_1{:}S_1, \ldots, x_n{:}S_n\}$, then ψ^N is the formula

$$\overline{\forall}(S_1(y_1) \wedge \ldots \wedge S_n(y_n) \rightarrow \psi\pi)$$

where π is the substitution $\{x_1{:}S_1 \mapsto y_1,\ \ldots,\ x_n{:}S_n \mapsto y_n\}$ for new variables $y_1, \ldots, y_n$.[3] The above formula for ψ^N can, in turn, be written as the clause

$$\neg r_1(y_1) \vee \ldots \vee \neg r_n(y_n) \vee \psi\pi.$$

The normalization of a set of formulas $\{\psi_1, \ldots, \psi_n\}^N$ is the set of its normalized formulas, $\left\{\psi_1^N, \ldots, \psi_n^N\right\}$. *By definition, any sorted sentence ψ is logically equivalent to its normalization ψ^N.* Therefore, any sorted axiomatization can be rewritten into an equivalent unsorted one, where the sorts are unary predicates.

We introduce a distinct constant k_i for each sort S_i and require that Σ contain the axiom $S_i(k_i)$ for each sort S_i.Therefore, that for all sorts S, $\Sigma \models S(g)$ for some ground term g.

[2] In [Frisch, 1991], variables can be restricted with boolean combinations of sort symbols. The domains denoted by these restrictions are defined in terms of the union, intersection and complement of the domains denoted by the corresponding sort predicates. We will not need to consider these more general sortal restrictions, so we avoid them to simplify this presentation. As we will see in Chapter 6, SSC's can be used to represent a special class of these boolean combinations.

[3] Recall that we write $\overline{\forall}\psi$ to denote the *universal closure* of ψ, that is, $\overline{\forall}\psi \stackrel{\text{def}}{=} \forall x_1 \ldots \forall x_n(\psi)$, where $\{x_1, \ldots, x_n\}$ is the set of free variables in ψ.

This preempts the "empty sort problem," which rightly deserves to be called "infamous" and "annoying" — as in [Smolka, 1989]. The unique constants are also useful in a number of proofs later on.

We close this section with the definition of the *well-sorted* substitutions with respect to a sort theory Σ:

Definition 5.2 (Well-sorted substitutions relative to a sort theory) *Given a sort theory Σ, a substitution θ is* well-sorted relative to a sort theory Σ, *or* Σ-well-sorted, *if $\Sigma \models \overline{\forall} S((x{:}S)\theta)$ for all variables $x{:}S \in \mathcal{V}$.*

5.2 Sorted Signatures

Other formulations of sorted logics define an *order-sorted signature* that fixes the sets of terms that belong to each sort [Walther, 1985b; Schmidt-Schauß, 1989]. Whereas in the sort theory formulation sorts are a syntactic shorthand for semantic restrictions, sorts are "built into" the language in the sorted signature approach. The following description of sorted signatures is adapted from [Schmidt-Schauß, 1989].

The main language is identical to that in the previous section: We are given a set of sort symbols $\mathcal{S}_\Sigma$ and an assignment of sorts to variables such that for each sort $S \in \mathcal{S}_\Sigma$ there is a countably infinite set of variables of sort S. Again, we write $x{:}S$ to indicate that x is of sort S. Together with the set of sorted variables, a sorted signature Σ contains:

- A set of **subsort declarations**, of the form $S_i \sqsubset S_j$ for $S_i, S_j \in \mathcal{S}_\Sigma$.
- A set of **term declarations**, of the form $S[t]$ for a non-variable term t and a sort S. If a term declaration is of the form $S_i[f(x_1{:}S_1, \ldots, x_n{:}S_n)]$ where $x_i \neq x_j$ if $i \neq j$, then it is called a **function declaration** for f.

We only consider finite signatures, which have a finite number of sorts and a finite number of subsort, function and predicate declarations.

The reflexive and transitive closure of the $\sqsubset$ relation defined by the subsort declarations is denoted by $\sqsubseteq_\Sigma$. We require that $\sqsubseteq_\Sigma$ be a partial ordering on the set of sorts $\mathcal{S}_\Sigma$. This means that there are no cycles in the directed graph defined by the subsort declarations.[4]

To avoid the empty sort problem, Schmidt-Schauß [1989] assumes that for every sort S_i there exists a ground term g_i such that $g_i \in \mathcal{T}_{\Sigma,S_i}$. As in the previous section, we enforce this by requiring that the signature contain a term declaration of the form $S_i[k_i]$ for each sort S_i, where each k_i is a constant that appears nowhere else in the signature. We also require that Σ have a *top sort*, to which all well-sorted terms (as defined below) belong. Variables written with no sort restriction on them are understood to belong to the top sort.

Definition 5.3 ($\mathcal{T}_{\Sigma,S}$) *For a sort symbol S, the set of terms of sort S, denoted by $\mathcal{T}_{\Sigma,S}$, is recursively defined as follows:*

1. *$x{:}S' \in \mathcal{T}_{\Sigma,S}$ if $S' \sqsubseteq_\Sigma S$.*

[4] Since we are not dealing with equality, we could always collapse each equivalence class induced by $\sqsubseteq_\Sigma$ into a single sort, if $\sqsubseteq_\Sigma$ were only a quasi-ordering.

2. *$t \in \mathcal{T}_{\Sigma,S}$ if $S'[t]$ is a term declaration in Σ and $S' \sqsubseteq_\Sigma S$.*

3. *$t \in \mathcal{T}_{\Sigma,S}$ if $t' \in \mathcal{T}_{\Sigma,S}$, t' contains the variable $x{:}S_1$, and t can be obtained from t' by applying the substitution $\{x{:}S_1 \mapsto s\}$, where $s \in \mathcal{T}_{\Sigma,S_2}$ and $S_2 \sqsubseteq_\Sigma S_1$.*

It follows from this definition that if $S_1 \sqsubseteq_\Sigma S_2$, then $\mathcal{T}_{\Sigma,S_1} \subseteq \mathcal{T}_{\Sigma,S_2}$. The set of ground terms in $\mathcal{T}_{\Sigma,S}$ is denoted by $\mathcal{T}_{\Sigma,S,gr}$.

The union of $\mathcal{T}_{\Sigma,S}$ over all sorts S is the set of **well-sorted terms**. Given a term t, the **sortal image** of t, denoted by $\mathcal{S}_\Sigma(t)$, is the set of all sorts S such that $t \in \mathcal{T}_{\Sigma,S}$. It is often sufficient to focus only on the *minimal* sorts in the sortal image, if we assume that $\sqsubseteq_\Sigma$ is a partial order. Trivially, a sorted variable $x{:}S$ has S as its least sort: $\mathcal{S}_\Sigma(x{:}S) = \{S' \mid S \sqsubseteq_\Sigma S'\}$. We usually say that S is *the* sort of $x{:}S$. However, in general $\mathcal{S}_\Sigma(t)$ need not contain a single least sort for a given term t. A term t is **ill-sorted** if $\mathcal{S}_\Sigma(t) = \emptyset$.

Definition 5.4 (Well-sorted substitutions relative to a sorted signature) *A substitution θ is a* well-sorted substitution relative to a sorted signature Σ, *or* Σ-well-sorted, *if $\mathcal{S}_\Sigma(x{:}S) \subseteq \mathcal{S}_\Sigma(x{:}S\theta)$ for all variables $x{:}S \in \mathcal{V}$.*

That is, the set of all sorts to which $(x{:}S)\theta$ belongs should be a subset of the set of all sorts to which $x{:}S$ belongs. This is the case iff $S' \sqsubseteq_\Sigma S$ for all $S' \in \mathcal{S}_\Sigma(x{:}S\theta)$. The following lemma provides a more intuitive characterization of Σ-well-sorted substitutions for sorted signatures and is useful later on:

Lemma 5.5 *θ is a Σ-well-sorted substitution if, and only if, $(x{:}S)\theta \in \mathcal{T}_{\Sigma,S}$ for all variables $x{:}S \in \mathcal{V}$.*

Proof: For any variable $x{:}S$, let $t = (x{:}S)\theta$. "Only if": Follows directly from the definition: If θ is Σ-well-sorted then $\mathcal{S}_\Sigma(x{:}S) \subseteq \mathcal{S}_\Sigma(t)$, and so $S \in \mathcal{S}_\Sigma(t)$, which means that $t \in \mathcal{T}_{\Sigma,S}$.

"If": Assume that for all variables $x{:}S$, if $t = (x{:}S)\theta$, then $t \in \mathcal{T}_{\Sigma,S}$. We need to show that $\mathcal{S}_\Sigma(x{:}S) \subseteq \mathcal{S}_\Sigma(t)$. That is, if a sort S' is in $\mathcal{S}_\Sigma(x{:}S)$, then $S' \in \mathcal{S}_\Sigma(t)$. Recall that $\mathcal{S}_\Sigma(x{:}S) = \{S' \mid S \sqsubseteq_\Sigma S'\}$. Let $S' \in \mathcal{S}_\Sigma(x{:}S)$. Then $S \sqsubseteq_\Sigma S'$ implies $\mathcal{T}_{\Sigma,S} \subseteq \mathcal{T}_{\Sigma,S'}$, and so $t \in \mathcal{T}_{\Sigma,S}$ only if $t \in \mathcal{T}_{\Sigma,S'}$. This means that $S' \in \mathcal{S}_\Sigma(t)$. ■

That is, a substitution is Σ-well-sorted iff it maps each sorted variable into a term of that sort. In general, Schmidt-Schauß [1989] shows that if $t \in \mathcal{T}_{\Sigma,S}$, then $t\theta \in \mathcal{T}_{\Sigma,S}$ for all Σ-well-sorted substitutions θ.[5] It is also shown that the set of non-variable terms of a sort S can be generated by applying Σ-well-sorted substitutions to the term declarations for S and subsorts of S. This is the "only if" direction of the following (the "if" direction follows directly from the definition of $\mathcal{T}_{\Sigma,S}$ above):

Theorem 5.6 ([Schmidt-Schauß, 1989]) *For every sort S and non-variable term t, $t \in \mathcal{T}_{\Sigma,S}$ if, and only if, there exists a term declaration $S'[t']$ such that $S' \sqsubseteq_\Sigma S$ and $t = t'\theta$ for some Σ-well-sorted substitution θ.*

Sorted signatures can also contain *predicate declarations*, which specify what combinations of sorts are allowed as arguments for each predicate and can be used to determine which formulas are well-sorted and which are not. However, Schmidt-Schauß [1989, page 7] notes:

---

[5] Note that this and Lemma 5.5 also hold for a *sort theory* Σ if we define $\mathcal{T}_{\Sigma,S}$ as the set $\{t \mid \Gamma \models \overline{\forall} S(t)\}$, and $\mathcal{S}_\Sigma(t)$ as the set of all sorts S such that $t \in \mathcal{T}_{\Sigma,(}S)$.

...the distinction between well-sorted and ill-sorted formulae is not an essential one. The satisfiability of a formula does not change, if we modify the signature and consider all ill-sorted expressions as well-sorted (of a particular sort). This justifies our assumption in the following that we can ignore the problem of the deduction of ill-sorted formulae and can always assume that all formulae are well-sorted. However, the restriction remains that only well-sorted substitutions and instantiations are to be used.

5.3 Relativizations

Presentations of sorted logics usually include a method for translating a set of sorted axioms S under a theory or signature Σ into an unsorted one S', such that S (given Σ) is satisfiable if, and only if, S' is satisfiable. The unsorted axiom set S' is said to be the *relativization* of S under Σ, denoted by $Rel(S, \Sigma)$.

In the case of sort theories, the relativization is simply the union of the sort theory and the *normalization* of the sorted axioms, as defined in Section 5.1. Satisfiability is trivially preserved, since sorted formulas are defined to be logically equivalent to their normalization.

In the case of sorted signatures, the relativization process involves translating the signature Σ into a set of axioms, denoted by $\mathcal{A}_\Sigma$, as follows: Introduce a new unary predicate for each sort $S \in \mathcal{S}_\Sigma$; for convenience, we assume that each predicate has the same name as its respective sort. For each subsort declaration $S_1 \sqsubset S_2$ in Σ, $S_1 \neq S_2$, add to $\mathcal{A}_\Sigma$ the sentence $\forall x\ S_1(x) \rightarrow S_2(x)$. Similarly, for every term declaration $S_0[t]$ in Σ, add to $\mathcal{A}_\Sigma$ the sentence:

$$\forall x_1 \ldots \forall x_n\ (S_1(x_1) \wedge \ldots \wedge S_n(x_n)) \rightarrow S_0(t\pi)$$

where $\{y_1{:}S_1, \ldots, y_n{:}S_n\} = Var(t)$ and π is the substitution $\{y_1{:}S_1 \mapsto x_1, \ldots, y_n{:}S_n \mapsto x_n\}$, where $\{x_1, \ldots, x_n\}$ is a set of distinct new variables. Note that $\mathcal{A}_\Sigma$ is equivalent to a set of Horn clauses. We call $\mathcal{A}_\Sigma$ the sort *theory* corresponding to Σ or sometimes simply the *relativization* of Σ.

For a sentence ϕ in prenex normal form, the *relativization* of ϕ is syntactically identical to its normalization, ϕ^N, as defined in Section 5.1.[6] Given a set of clauses S and a sorted signature Σ, the relativization of S, $Rel(S, \Sigma)$, is the union of $\mathcal{A}_\Sigma$ and the clauses c^N for each clause $c \in S$ (that is, S^N). Note that $Rel(S, \Sigma)$ can be immediately rewritten into clausal form.

Example 5.7 *Table 5.1 shows a fragment of a sorted signature Σ and its relativization A_Σ. The clause $Prime(sum(x{:}\text{ODD}, y{:}\text{EVEN})) \vee Divides(z{:}\text{INTEGER}, w{:}\text{ODD})$ is relativized to*

$$\neg Odd(x) \vee \neg Even(y) \vee \neg Integer(z) \vee \neg Odd(w) \vee Prime(sum(x, y)) \vee Divides(z, w)$$

together with the relativized signature.

Note that the function sum is severely overloaded. If we defined the addition of a complex number and a real in the main logic, we would want to declare $sum(x{:}\text{COMPLEX}, y{:}\text{REAL})$ to be of sort COMPLEX. *We still might choose not to declare the sum of* VECTOR *and* REAL *to be well-sorted.*

[6] Both Schmidt-Schauß [1989] and Frisch [1991] discuss the problem of translating arbitrary sentences in their first order sorted logics into clausal form.

Declarations in Σ	Axioms in $\mathcal{A}_\Sigma$
$\text{INTEGER} \sqsubseteq \text{NUMBER}$ $\text{REAL} \sqsubseteq \text{NUMBER}$ $\text{COMPLEX} \sqsubseteq \text{NUMBER}$ $\text{EVEN} \sqsubseteq \text{INTEGER}$ $\text{ODD} \sqsubseteq \text{INTEGER}$	$\forall x\ Integer(x) \rightarrow Number(x)$ $\forall x\ Real(x) \rightarrow Number(x)$ $\forall x\ Complex(x) \rightarrow Number(x)$ $\forall x\ Even(x) \rightarrow Integer(x)$ $\forall x\ Odd(x) \rightarrow Integer(x)$
$\text{REAL}(sum(x{:}\text{REAL}, y{:}\text{REAL}))$ $\text{INTEGER}(sum(x{:}\text{INTEGER}, y{:}\text{INTEGER}))$ $\text{EVEN}(sum(x{:}\text{EVEN}, y{:}\text{EVEN}))$ $\text{EVEN}(sum(x{:}\text{INTEGER}, x{:}\text{INTEGER}))$	$\forall x \forall y\ Real(x) \wedge Real(y) \rightarrow Real(sum(x, y))$ $\forall x \forall y\ Integer(x) \wedge Integer(y) \rightarrow Integer(sum(x, y))$ $\forall x \forall y\ Even(x) \wedge Even(y) \rightarrow Even(sum(x, y))$ $\forall x\ Integer(x) \rightarrow Even(sum(x, x))$

Table 5.1: Sort declarations and the corresponding axioms in a sort theory.

Figure 5.1 *shows the sort hierarchy for the signature in Table* 5.1.

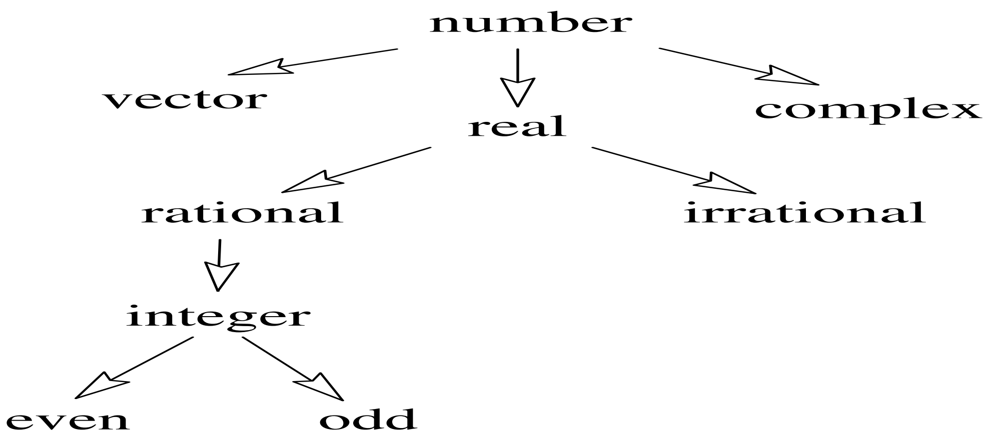

Figure 5.1: A tree-structured sort hierarchy

After a relativization is defined, a **sort theorem** is usually proved, as in [Schmidt-Schauß, 1989], establishing that that a sorted axiomatization S is satisfiable by a sorted model if, and only if, $Rel(S, \Sigma)$ is satisfiable by an unsorted one. This follows from the restrictions imposed on the models of a set of sorted formulas given a sorted signature. In the sort theory case, the equivalent result (that S is satisfiable iff S^N is) is obvious, since a sorted formula is a syntactic abbreviation of its normalization. Therefore, a set of axioms S under a sorted signature Σ is satisfiable if, and only if, $S \cup \mathcal{A}_\Sigma$ is satisfiable.

5.4 Equivalence of Well-sorted Substitutions and Unifiers

We now show that sorted unification under a sorted signature is equivalent to unification under its relativization, by showing that the set of well-sorted substitutions and the generality ordering between them are preserved. This will help us gain some insight on how we can view a sorted signature Σ as equivalent to the corresponding set of Horn clauses $\mathcal{A}_\Sigma$.

Recall that we assume that there is a distinct constant k_i for each sort S_i. In the sort theory (resp. sorted signature), this is expressed with the set of axioms (resp. term declarations) $\{S_i(k_i)\}$ (resp. $\{S_i[k_i]\}$).

In the sort theory case, ψ^N is by definition logically equivalent to $\overline{\forall}\psi$, so $\mathcal{A}_\Sigma \models \overline{\forall}\psi$ iff $\mathcal{A}_\Sigma \models \psi^N$. Therefore, we could have defined the relativization $\mathcal{A}_\Sigma$ of a sorted signature Σ as the union of the following FOPC sentences:

- $(\overline{\forall}S(t))^N$, where $S[t]$ is a term declaration in Σ, and
- $(\overline{\forall}S_2(x{:}S_1))^N$, if $S_1 \sqsubset S_2$ is a subsort declaration in Σ.

We now show that the $\sqsubseteq_\Sigma$ relation induced by the subsort declarations in Σ is equivalent to the subsort relation among sort predicates defined by $\mathcal{A}_\Sigma$:

Lemma 5.8 *Given a sorted signature Σ, $S_1 \sqsubseteq_\Sigma S_2$ if, and only if, $\mathcal{A}_\Sigma \models \forall x(S_1(x) \rightarrow S_2(x))$.*

Proof: "Only if:" Assume that $S_1 \sqsubseteq_\Sigma S_2$. $\mathcal{A}_\Sigma$ contains the sentence $\forall x(S_1(x) \rightarrow S_2(x))$ iff $S_1 \sqsubset S_2$ is a subsort declaration in Σ. Since $\sqsubseteq_\Sigma$ is the transitive closure of the subsort declarations in Σ, $S_1 \sqsubseteq_\Sigma S_2$ iff there exists a finite sequence of sorts $\{S_1 = S'_1, \ldots, S'_n = S_2\}$ such that for all i, $1 \leq i < n$, $S'_i \sqsubset S'_{i+1}$ is a subsort declaration in Σ. Then $\forall x(S'_i(x) \rightarrow S'_{i+1}(x))$ is in $\mathcal{A}_\Sigma$, so $\mathcal{A}_\Sigma \models \forall x(S_1(x) \rightarrow S_2(x))$, by repeated applications of modus ponens.

"If:" Assume that $\mathcal{A}_\Sigma \models (\forall x S_1(x) \rightarrow S_2(x))$. Therefore, $\mathcal{A}_\Sigma \models S_2(k_1)$, where k_1 is the Skolem constant for the sort S_1. Since $\mathcal{A}_\Sigma$ is a set of Horn clauses, then there exists an SLD-resolution deduction of the empty clause from $\neg S_2(k_1)$.[7] We now notice that k_1 does not appear anywhere in $\mathcal{A}_\Sigma$ except in the clause $S_1(k_1)$, and thus can only resolve with literals of the form $S_1(x)$. The only clauses in $\mathcal{A}_\Sigma$ that contain literals of the form $S_i(x)$ are those that correspond to subsort declarations $S_j \sqsubset S_i$, of the form $S_i(x) \vee \neg S_j(x)$. Therefore, the intermediate negative clauses in an SLD-resolution refutation of $\neg S_2(k_1)$ are all of the form $\neg S_i(k_1)$, where the final resolution step resolves $\neg S_1(k_1)$ and $S_1(k_1)$. This means that there is in fact a finite sequence of sorts $\{S_1 = S'_1, \ldots, S'_n = S_2\}$ such that $S'_i \sqsubset S'_{i+1}$ is a subsort declaration in Σ for all i, $1 \leq i < n$, and thus $S_1 \sqsubseteq_\Sigma S_2$. ■

This means that we can write $S_1 \sqsubseteq_\Sigma S_2$ when $\mathcal{A}_\Sigma \models \forall x S_1(x) \rightarrow S_2(x)$, or equivalently, when $\mathcal{A}_\Sigma \models \overline{\forall}S_2(x{:}S_1)$.

The following lemma is the counterpart to Theorem 5.6, applied to the relativization $\mathcal{A}_\Sigma$ of a sorted signature Σ:

Lemma 5.9 *Let Σ be a sorted signature. For any non-variable term t, $\mathcal{A}_\Sigma \models \overline{\forall}S(t)$ if, and only if, $\mathcal{A}_\Sigma$ contains $S'(t')^N$ for some sort $S' \sqsubseteq_\Sigma S$ and some non-variable term t' that is $\mathcal{A}_\Sigma$-more-general than t.*

[7] From the completeness of SLD-resolution for sets of Horn Clauses.

Proof: "If": Follows directly from the definitions: Assume that $\mathcal{A}_\Sigma$ contains $S'(t')^N$ and that $t' \geq_{\mathcal{A}_\Sigma} t$. Since $S'(t)^N$ iff $\overline{\forall} S'(t')$, then $\mathcal{A}_\Sigma \models \overline{\forall} S'(t)$. Let θ be the $\mathcal{A}_\Sigma$-well-sorted substitution such that $t'\theta = t$. Let $z{:}S$ be a new variable. Then $\Omega = \{z{:}S \mapsto t'\}$ is a $\mathcal{A}_\Sigma$-well-sorted substitution: If $S' \sqsubseteq_\Sigma S$, then $\mathcal{A}_\Sigma \models \forall x(S'(x) \to S(x))$, by Lemma 5.8; since $\mathcal{A}_\Sigma \models \overline{\forall} S'(t')$, then $\mathcal{A}_\Sigma \models \overline{\forall} S(t')$. Then $\Omega \cdot \theta$ is also $\mathcal{A}_\Sigma$-well-sorted, so $\mathcal{A}_\Sigma \models \overline{\forall} S((z{:}S)(\Omega \cdot \theta))$ implies $\mathcal{A}_\Sigma \models \overline{\forall} S(t'\theta)$ implies $\mathcal{A}_\Sigma \models \overline{\forall} S(t)$.

"Only if": All the clauses in $\mathcal{A}_\Sigma$ are of the form $S_i(g)$ for a ground term g or $S_1(x_1) \wedge \ldots \wedge S_n(x_n) \to S_0(t_0)$ where $x_i \neq x_j$ for $i \neq j$ and $\{x_1, \ldots, x_n\} = Var(t_0)$. If $\mathcal{A}_\Sigma \models \overline{\forall} S(t)$, then if we replace each variable $x_i{:}S_i$ in t by the Skolem constant k_{S_i}, the resulting term g is an $\mathcal{A}_\Sigma$-ground instance of t, so $\mathcal{A}_\Sigma \models S(g)$. But then there has to be a clause in $\mathcal{A}_\Sigma$ whose head is of the form $S'(t')$, where $\mathcal{A}_\Sigma \models \forall x(S'(x) \to S(x))$ and t' is $\mathcal{A}_\Sigma$-more-general than g. By Lemma 5.8, this can only be the case if the clause is of the form $S'(t')^N$, where $S' \sqsubseteq_\Sigma S$. From the construction of g, this means that t' is $\mathcal{A}_\Sigma$-more-general than t. ■

Theorem 5.10 *Let Σ be a sorted signature. Then $t \in \mathcal{T}_{\Sigma,S}$ if, and only if, $\mathcal{A}_\Sigma \models \overline{\forall} S(t)$ for any sort S and term t.*

Proof: We use induction on the term t:
Basis: t is a constant or a (sorted) variable. If t is a constant c, then $c \in \mathcal{T}_{\Sigma,S}$ iff $S_1[c]$ is a term declaration in Σ and $S_1 \sqsubseteq_\Sigma S$. This is the case iff $S_1(c) \in \mathcal{A}_\Sigma$ and (by Lemma 5.8) $\mathcal{A}_\Sigma \models \forall x(S_1(x) \to S(x))$, which is the case iff $\mathcal{A}_\Sigma \models S(c)$.

If t is a variable $x{:}S_1$, then $x{:}S_1 \in \mathcal{T}_{\Sigma,S}$ iff $S_1 \sqsubseteq_\Sigma S$, and, by Lemma 5.8, this is the case iff $\mathcal{A}_\Sigma \models \forall x S_1(x) \to S(x)$.
Induction Step: "Only if": By Theorem 5.6, $t \in \mathcal{T}_{\Sigma,S}$ iff $S'[t']$ is in Σ, $S \sqsubseteq_\Sigma S'$, and $t = t'\theta$ for a Σ-well-sorted substitution θ. For all variables $x_i{:}S_i \in Var(t)$, if $(x_i{:}S_i)\theta = t_i$ then, since θ is Σ-well-sorted, $t_i \in \mathcal{T}_{\Sigma,S_i}$. The nesting depth of t_i has to be less than that of t, for all i, since t' is a non-variable term. This, by induction hypothesis, means that $\mathcal{A}_\Sigma \models \overline{\forall} S_i(t_i)$. Also, since $S'[t']$ is in Σ, $S'(t')^N \in \mathcal{A}_\Sigma$. By Lemma 5.9, we can conclude that $\mathcal{A}_\Sigma \models \overline{\forall} S(t)$.

"If": Assume that $\mathcal{A}_\Sigma \models \overline{\forall} S(t)$. Let $t = f(t_1, \ldots, t_n)$. Then by Lemma 5.9 there has to be a clause in $\mathcal{A}_\Sigma$ of the form $S'(t')^N$, where $S \sqsubseteq_\Sigma S'$, $t' = f(t'_1, \ldots, t'_n)$ and there exists a Σ-well-sorted substitution θ such that $t = t'\theta$. Then if $(x{:}S_i)\theta = s_i$, $\mathcal{A}_\Sigma \models \overline{\forall} S_i(s_i)$, Since the nesting depth of each t_i is less than that of t, then $s_i \in \mathcal{T}_{\Sigma,S_i}$ for all i, $1 \leq i \leq n$, by induction hypothesis. Therefore, θ is also Σ-well-sorted. If $(t')^N$ is in $\mathcal{A}_\Sigma$, then $S[t']$ is a term declaration in Σ; we can now conclude from Theorem 5.6 that $t \in \mathcal{T}_{\Sigma,S}$. ■

In general, we can say that a sort theory and a sorted signature are equivalent if the set of terms that belong to each sort are the same:

Definition 5.11 (Equivalence of Sorted Signatures and Theories) *We say that a sorted signature Σ and a sort theory Γ are* equivalent *if for all sorts $S \in \mathcal{S}_\Sigma$, $\mathcal{T}_{\Sigma,S} = \left\{t \mid \Gamma \models \overline{\forall} S(t)\right\}$.*

Theorem 5.10 shows that a sorted signature Σ is equivalent, in this sense, to its relativization $\mathcal{A}_\Sigma$. The equivalence of the unification problem under a theory Γ and an equivalent signature Σ is clear: If the set of terms of sort S is the same as the set of terms such that $\Gamma \models \overline{\forall} S(t)$, then the set of well-sorted substitutions is the same:

Theorem 5.12 *Let Σ be a sorted signature and let Γ be a sort theory equivalent to Σ. Then a substitution θ is Σ-well-sorted if, and only if, θ is Γ-well-sorted.*

Proof: Let $\theta = \{x_1{:}S_1 \mapsto t_1, \ldots, x_n{:}S_n \mapsto t_n\}$. By Lemma 5.5, θ is Σ-well-sorted iff $t_i \in \mathcal{T}_{\Sigma,S_i}$. Since Γ is equivalent to Σ, this holds iff $\Gamma \models \overline{\forall} S_i(t_i)$, i.e. θ is a Γ-well-sorted unifier. ■

In the rest of this section we use "Σ" to refer to a sort theory or a sorted signature; all definitions, propositions and theorems apply equally in either case. Since the set of well-sorted unifiers is the same under the two definitions, the orderings for well-sorted unifiers and well-sorted terms, defined below, will be equivalent too. We begin with a basic proposition proved in both [Frisch, 1991] and [Schmidt-Schauß, 1989] (in different ways, since they are based on different definitions):

Proposition 5.13 *1. The composition of two Σ-well-sorted substitutions is Σ-well-sorted for any sort theory Σ. 2. The identity substitution is Σ-well-sorted for any sort theory Σ.*

A solution for a set of equations $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ is defined as in the unsorted case: It is a substitution θ such that $s_i\theta = t_i\theta$ for all i, $1 \le i \le n$; a *unifier* for a set of terms $\{t_1, \ldots, t_n\}$ is a solution to the set of equations $\{t_1 = t_2, \ldots, t_{n-1} = t_n\}$. We will now be interested only in those solutions and unifiers that are Σ-well-sorted for a given Σ. Defining a generality ordering among Σ-well-sorted substitutions requires some new definitions, since Σ-well-sorted unifiers may have to introduce new variables; we want to make clear that the *names* of these new variables are irrelevant.[8] This is done by restricting the domains of substitutions: For a set of variables W and a substitution θ, $\theta|_W$ denotes the substitution such that $x(\theta|_W) = x\theta$ if $x \in W$ and $x(\theta|_W) = x$ otherwise. In this way we can focus on the behavior of substitutions over the set of variables W:

Definition 5.14 (Σ-more-general) *Let θ and Ω be Σ-well-sorted substitutions. We say that θ is* Σ-more-general *than Ω with respect to a set of variables W if $(\theta \cdot \phi)|_W = \Omega|_W$ for some Σ-well-sorted substitution ϕ. In this case we write $\theta \ge_\Sigma \Omega[W]$.*

If the set of sorts is finite, it can be shown that there are no strictly ascending chains of Σ-well-sorted unifiers.[9] Two substitutions θ and Ω are **Σ-incomparable** with respect to a set of variables W if neither $\theta \ge_\Sigma \Omega[W]$ nor $\Omega \ge_\Sigma \theta[W]$ is true. The set W is usually the set of all variables that appear in any of the syntactic objects mentioned up to the point at which the unifiers being compared are introduced. As in [Frisch, 1991], we call this the set of *variables of concern*, and will always be clear from the context. Thus, we usually omit the references to W from the above notation.

A term t is *Σ-more-general* than a term s if $s = t\theta$ for a Σ-well-sorted substitution θ. A term g is a **Σ-ground instance** of t if g is ground and t is Σ-more-general than g. Finally, we define a *Σ-complete set of incomparable unifiers* (ΣCIU) for a set of terms S:

Definition 5.15 (ΣCIU) *Let S be a finite set of sorted terms and let Γ be a set of Σ-well-sorted substitutions. Then we say that Γ is:*

1. *A* correct *set of Σ-unifiers for S if all substitutions $\theta \in \Gamma$ are Σ-well-sorted unifiers of S.*

2. *A* complete *set of Σ-unifiers for S if for any Σ-unifier Ω of S there exists a substitution $\theta \in \Gamma$ such that $\theta \ge_\Sigma \Omega[Var(S)]$.*

[8] Here we follow the definition in [Frisch, 1991]; the one in [Schmidt-Schauß, 1989] is equivalent.

[9] Frisch [1991] proves that this is never the case when restrictions are arbitrary boolean combinations of sorts, given that the set of sorts is finite.

3. *An* incomparable *set of Σ-unifiers if any two different substitutions in Γ are Σ-incomparable with respect to $Var(S)$.*

A ΣCIU for S is a correct, complete and incomparable set of unifiers of S, with respect to Σ.

Similarly, given a finite set of equations between sorted terms E, we can talk of a correct, complete and incomparable set of Σ-solutions for E; the solutions of a set of equations will also be called the *unifiers* of E, so that we can also talk of ΣCIU's for equation sets.

A ΣCIU for a set of terms or equations is empty if the set is not unifiable. Note that ΣCIU's are well-defined (that is, they always exist), since there are no strictly ascending chains of Σ-well-sorted substitutions in the ordering given by $\geq_\Sigma$. Frisch [1991] proves that for any set of terms S, all ΣCIU's for S have the same cardinality. Furthermore, they are all variants of each other, in the sense that if C_1 and C_2 are ΣCIU's for S, then for any substitution $\theta_1 \in C_1$ there is a substitution $\theta_2 \in C_2$ such that $\theta_1 \leq_\Sigma \theta_2$ and $\theta_2 \leq_\Sigma \theta_1$.

5.4.1 Inverse Relativization, Part 1

In Section 5.3 we translated a sorted signature Σ into an equivalent sort theory $\mathcal{A}_\Sigma$. This translation can also go the other way, as follows: Given a sort theory Γ, let Σ be the sorted signature constructed using the following two rules:

- **Rule 1**: For each pair of sorts S_1 and S_2, $S_1 \neq S_2$, add the subsort declaration $S_1 \sqsubset S_2$ to Σ if $\Gamma \models \forall x(S_1(x) \rightarrow S_2(x))$.
- **Rule 2**: For each non-variable term t such that $\Gamma \models \overline{\forall} S(t)$ and there is no term $t' \geq_\Gamma t$ where $\Gamma \models \overline{\forall} S(t')$, add the term declaration $S[t]$ to Σ.

Note that the signature Σ might not contain all the information about sorts described in the theory Γ.[10] The set of declarations introduced by Rule 2 may be infinite, and might possibly not even be recursive. For what kinds of sort theories is the above translation effectively computable is left as an open problem. (In Chapter 6 we outline a procedure to do this for a simple set of cases.)

5.5 Results on Sorted Unification

Sorted signatures are often classified according to the maximum cardinality of a correct, complete and incomparable set of well-sorted most general unifiers for a finite set of equations E. We can do the same for sort theories; if the sortal constraints are simple sorts, as in the previous two sections, and we only consider sort theories equivalent to sorted signatures, then the classification results will also be equivalent, since well-sorted unifiers and the generality ordering between them are the same.

For a sorted signature (or sort theory) Σ and a finite set of equations E, let $\Sigma CIU(E)$ denote any ΣCIU for E; recall that all ΣCIU's of E have the same cardinality. A sorted signature (or sort theory) Σ is said to be of type

- **unitary**, if $|\Sigma CIU(E)| \leq 1$ for all E,

[10] If the sortal restrictions are atomic, it follows from the results in [Frisch, 1991] that this is all the information that is needed by a substitutional system.

- **finitary**, if $|\Sigma CIU(E)|$ is finite for all E, and
- **infinitary**, if $|\Sigma CIU(E)|$ is infinite for some E.

A signature is called *monomorphic* if for every n-ary function symbol f, all the well-sorted instances of $f(x_1, \ldots, x_n)$ belong to the same sorts. Walther [1985b] shows how the type of monomorphic signatures depends on the maximum cardinality of a set of greatest lower bounds (glb's) for any finite set of sorts: If Σ is a monomorphic signature under a quasi-ordering $\sqsubseteq_\Sigma$, then Σ is of type unitary if $|glb(E)| \leq 1$ for all S, of type finitary if $|glb(E)|$ is finite for all S, and of type infinitary if $|glb(E)| = \infty$ for some S, where S is a set of sorts. A monomorphic signature can be of type infinitary only if it contains an infinite number of sorts.

Schmidt-Schauß [1989] gives an extensive classification of sorted signatures with term declarations. We now define some of these classes. A sorted signature is said to be:

- **Elementary**, if all each term declaration is of the form $S_o(f(x_1{:}S_1, \ldots, x_n{:}S_n))$, where $x_i \neq x_j$ for $i \neq j$, called a *function declarations for f*. In the literature, the term "order-sorted signature" sometimes refers exclusively to these types of signatures, as in [Comon and Delor, 1991]; each declaration for f is sometimes described as an ordered $n+1$-tuple of sorts, and written as $f(S_1, \ldots, S_n) : S_0$ or $f : S_1 \times \ldots \times S_n \rightarrow S_0$.
- **Simple**, if the signature is elementary and there is exactly one declaration for each function symbol. (These are basically equivalent to what we call *monomorphic* signatures or theories.)
- **Regular**, if $\sqsubseteq_\Sigma$ is a partial ordering and each term t has a unique least sort.

  As opposed to the above, this is not a merely syntactical restriction on the declarations; in fact, Schmidt-Schauß [1989] shows that this is an undecidable property of sorted signatures, in the general case.[11]
- **"Polymorphic,"** if the signature is elementary, regular and has a top sort.

  Note that these signatures correspond to a small subset of what we call *polymorphic* sort theories in this thesis. We will write the name in quotes when referring to the more restricted class. The existence of a top sort is not a particularly distinctive requirement — we can always add one — but the regularity condition is.
- **Linear**, if all term declarations are *linear*; that is, they do not contain repeated variables.
- **Almost elementary**, if the signature is linear and for each declaration $S(f(t_1, \ldots, t_n))$, t_i is either a ground term or a variable for all i, $1 \leq i \leq n$.

For all our purposes, we can collapse each equivalence class induced by $\sqsubseteq$ into a single sort, so we will assume that $\sqsubseteq_\Sigma$ is always a *partial* ordering.[12] Sorted signatures where $\sqsubseteq_\Sigma$ is not a partial ordering are not very interesting without equality.

Simple signatures are regular: If a term t is a variable $x{:}S$, its unique least sort is S; if t is not a variable, then there exists a single sort declaration that applies to it, which specifies its least sort. Simple signatures are therefore "polymorphic," so we have:

[11] This definition of regularity is weaker than the one used in other papers.

[12] That is, if $S_1 \sqsubseteq_\Sigma S_2$ and $S_2 \sqsubseteq_\Sigma S_1$, for sorts $S_1 \neq S_2$, then we can replace S_2 by S_1 in all declarations; this process can be iterated until no two different sorts are equivalent under $\sqsubseteq_\Sigma$.

- simple ⇒ "polymorphic" ⇒ elementary ⇒ almost elementary ⇒ linear, and
- simple ⇒ "polymorphic" ⇒ regular.

There are elementary signatures that are not regular. Figure 5.2 shows the resulting hierarchy of sorted signatures.

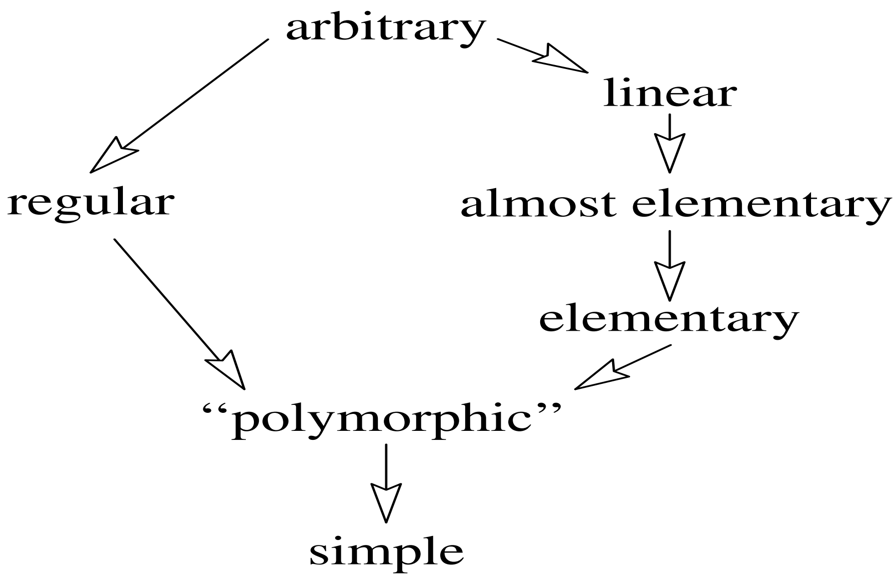

Figure 5.2: A hierarchy of sorted signatures

Some of the results in [Schmidt-Schauß, 1989] include:

- Σ-unification is decidable for "polymorphic" signatures (i.e. regular, elementary and having a top sort). Also, these signatures are of unification type *finitary*.
- Σ-unification is decidable for elementary signatures.
- The *regularity* of signatures (that is, whether every term t has a unique least sort) is undecidable.

To determine the regularity of a finite signature, we could test all possible pairs of incomparable sorts, asking whether the intersection each pair of sorts is empty or not. If this intersection is empty in all cases, the signature is regular. But if the intersection of two incomparable sorts is non-empty, there will exist a t such that $\mathcal{S}_\Sigma(t)$ contains these two incomparable sorts, so Σ cannot be regular. Thus, if we are able to decide whether $\mathcal{T}_{\Sigma,S_1} \cap \mathcal{T}_{\Sigma,S_2} = \emptyset$ for any two sorts S_1 and S_2, then we can decide if Σ is regular or not. This means that regularity is decidable for all finite signatures for which unification is decidable.

Table 5.2 summarizes the results from [Schmidt-Schauß, 1989] and [Walther, 1985b] mentioned above, assuming that there is a finite number of sorts and that $\sqsubseteq_\Sigma$ is a partial ordering. The signatures are listed in ascending complexity. Note that for simple signatures, unification is quasi-linear, while for "polymorphic" signatures it is $\mathcal{NP}$-complete. Unification is also $\mathcal{NP}$-complete for regular, elementary signatures.

Class of Σ	Type of Σ	Decidability/Complexity	Regularity
Simple (*"monomorphic"*)	Unitary or finitary, depending on $\sqsubseteq$	Decidable, quasi-linear	Always regular
"Polymorphic"	Finitary	Decidable, $\mathcal{NP}$-complete	Decidable
Elementary	Finitary if regular	Decidable	Decidable
Almost elementary	Finitary if regular	Decidable	Decidable
Linear	Infinitary	Decidable	Decidable
Arbitrary	Infinitary	Undecidable	Undecidable

Table 5.2: Type and decidability of some sorted unification problems

As a corollary of the results in Chapter 6, we fill in the previous unknowns in Table 5.2, showing that unification and regularity are decidable for linear signatures. Even though these signatures are in general of type infinitary, we can always obtain a finite number of SSC-unifiers after translating the original unification problem into the new SSC representation. Furthermore, this result is extended to a broader class of signatures, which we call *semi-linear* (see Section 6.3).

Chapter 6

Representing Sort Theories

This chapter describes how certain sort theories can be represented using the *shallow* systems of set constraints defined in Chapter 2. We give a formal definition of what it means for a sort theory to be equivalent to an SSC, and present an algorithm that carries out this "translation" for a certain class of sorted signatures.

Comon and Delor [1991, page 2] point out some of the advantages of expressing sorts as membership constraints:

> This point of view is simpler and avoids inadequate properties such as "regularity", "co-regularity", "downwards completeness" ... Indeed, such properties become necessary when one wants to ensure special features of solved forms (such as to represent a most general unifier without adding new sort symbols) but are actually useless for deciding the solvability [of unification problems]. On the other hand, solvability is the only (operational) requirement when the formulae are used as constraints in some constraint logic programming language.

Section 6.1 shows how certain classes of SSC's are equivalent to certain classes of logic programs. Section 6.2 describes how sort theories and signatures can be equivalent to SSC's. Section 6.3 shows how a class of sorted signatures (or the corresponding sort theories), the *semi-linear* ones, can be translated into equivalent shallow SSC's.[1] Finally, Section 6.4 presents some simple examples of sorted unification using SSC's.

6.1 SSC's and Logic Programs

We now show how systems of set constraints that contain no occurrences of "$\sqcap$" define the same sets of terms as certain logic programs over unary predicates. These logic programs, in turn, correspond to the relativization of sorted signatures.

Let $\mathcal{P}$ be a set of unary predicates. Borrowing notation from [Heintze and Jaffar, 1990], we identify a unique set variable Υ_P in $\mathcal{V}_\mathcal{S}$ for each predicate P. Now we can say that an SSC C represents the least Herbrand model of a logic program iff the predicate P and the constrained set variable Υ_P define exactly the same set of ground terms; that is:

[1] Section 6.3.4 discusses how a sort theory can be extracted from an unsorted axiomatization in the light of the previous definitions.

Definition 6.1 (Representing the least Herbrand model) *A system of set constraints* C represents the least Herbrand model $\mathcal{M}$ *of a logic program if for all unary predicates* P *in the logic program,* $M_C(\Upsilon_P) = [\![P]\!]^{\mathcal{M}}$.

Let L be a logic program where all the clauses are of the form $P(x)$ for a variable x or $P_1(x_1) \wedge \ldots \wedge P_n(x_n) \rightarrow P_0(t)$, where $x_j \neq x_j$ if $i \neq j$ and $Var(t) = \{x_1, \ldots, x_n\}$, $n \geq 0$. Define an SSC C_L as follows:

- $Dom(C_L)$ includes a variable $\Upsilon_P \in \mathcal{V_S}$ for each predicate P in L, together with a finite set of *auxiliary* set variables that contains as many distinct variables as there are negative literals in L (i.e. atoms in the bodies of the rules in L).

- If $P(x)$ is a clause in L, then $C_L(\Upsilon_P) = \top$; otherwise, $C_L(\Upsilon_P)$ should contain exactly the disjuncts $t_i\pi_i$ such that $P_1(x_1) \wedge \ldots \wedge P_n(x_n) \rightarrow P(t_i)$ is a rule in L and $\pi_i = \{x_1 \mapsto N_1, \ldots, x_n \mapsto N_n\}$, where N_j is a distinct auxiliary set variable for $1 \leq j \leq n$, and the set constraint $\langle N_i \equiv \Upsilon_{P_i}\rangle$ is included in C_L.

  If there are no such rules in L, then let $C_L(\Upsilon_P) = \bot$.

We assume that L contains no clauses of the form $P(x) \rightarrow P(x)$ and no distinct predicates P_1 and P_2 such that $P_1 \neq P_2$ and $L \models \forall x(P_1(x) \leftrightarrow P_2(x))$.[2] As a particular case, this means that there is at most one rule of the form $\forall x P(x)$. It is easy to verify that these conditions ensure that C_L has no cycles.

Lemma 6.2 *Let* $\mathcal{L}$ *be a logic program and let* $C_{\mathcal{L}}$ *be the corresponding* SSC *as defined above. Then* $C_{\mathcal{L}}$ *represents the least Herbrand model of* $\mathcal{L}$.

Proof: We know that $\mathcal{L}$ has a least Herbrand model, since all sets of Horn clauses do. Also, by Corollary 2.19, C_L has a *unique* model, since C_L has no parameters and no cycles.

M is the least Herbrand model of $\mathcal{L}$ iff for each sort predicate P_0 in $\mathcal{L}$,

$$[\![P_0]\!]^M = \left\{t\theta \mid (P_1(x_1) \wedge \ldots \wedge P_n(x_n) \rightarrow P_0(t)) \in \mathcal{L} \text{ and } x_i\theta \in [\![P_i]\!]^M\right\}$$

where θ is a ground substitution over $\{X_1, \ldots, X_n\}$. Therefore, if we define $I(\Upsilon_{P_0}) = [\![P_0]\!]^M$ for all P_0, it suffices to show that I is a model of C_L, that is, that for each set constraint $\langle P_0 \equiv exp\rangle$ in C_L, $I(P_0) = I(exp)$.

If $P_0(x)$ is a clause in $\mathcal{L}$, then $[\![P_0]\!]^M$ is the Herbrand universe, and $C_L(\Upsilon_{P_0}) = \top$, so indeed $I(\Upsilon_{P_0}) = I(\top)$. Consider now the case where $C_L(\Upsilon_{P_0}) \neq \top$: For any ground term t', $t' \in I(\Upsilon_{P_0})$ iff there is some clause $(P_1(x_1) \wedge \ldots \wedge P_n(x_n) \rightarrow P_0(t)) \in \mathcal{L}$ and $t' = t\theta$ for some ground substitution θ such that $x_i\theta \in M(P_i) = I(\Upsilon_{P_i})$.

Let $\pi = \{x_1 \mapsto N_1, \ldots, x_n \mapsto N_n\}$, where the N_i are the same auxiliary variables used in the definition of C_L above. Then $t' = t\theta = t\pi\theta'$ for some ground substitution θ' with domain $\{N_1, \ldots, N_n\}$. Furthermore, $N_i\theta' \in I(\Upsilon_{P_i})$, since $C_L(N_i) = \Upsilon_{P_i}$. From the construction of C_L, one of the disjuncts e_i in $C_L(P_0)$ is precisely $t\pi$. Therefore, $t' \in I(e_i)$, and thus $t' \in I(C_L(\Upsilon_{P_0}))$.[3]

[2] If this were the case, then we could always replace all the occurrences of P_1 by P_2 in L.

[3] Note that the auxiliary variables are needed in order to avoid introducing new inter-variable dependencies; it can be the case that $P_i = P_j$ for $i \neq j$.

All the implications go both ways, so that if $t' \in I(C_L(\Upsilon_{P_0}))$, then $t' \in I(\Upsilon_{P_0})$. This means that $I(X) = I(C_L(X))$ for all $X \in Dom(C_L)$, so I is a model of C_L. ∎

Note that C_L is not shallow, so the unification algorithm of Chapter 4 *cannot* be used to compute the intersection of $[\![t_1]\!]^M = M_{C_L}(t_1)$ and $[\![t_2]\!]^M = M_{C_L}(t_2)$ unless C_L can be transformed into an equivalent shallow SSC in solved form. This is done by the algorithm in Section 6.3 for a restricted class of SSC's.

Comon [1989] noticed that a frontier-to-root tree automaton can be immediately defined using the sort declarations in the case that the signature is *elementary*, that is, when all the declarations are function declarations. In this case, the corresponding Horn clauses are either $P(x)$ for the predicate corresponding to the top sort, or of the form $S_1(x_1) \wedge \ldots \wedge S_n(x_n) \rightarrow S_o(t)$, where t is linear and $Var(t) = \{x_1, \ldots, x_n\}$, $n \geq 0$ (see Section 5.5). The above procedure produces a linear SSC, i.e. a term grammar, equivalent to that automaton (after adding the appropriate rules for the nonterminal $\top$ — see Section 2.6).

Translating logic programs into SSC's will be our main concern, but we now briefly discuss the corresponding transformation in the other direction:

Let C be an SSC with no cycles and no occurrences of "$\sqcap$"; that is, $C(X) = t_1 \sqcup \ldots \sqcup t_n$ for atomic set expressions t_i. From C we obtain a logic program $\mathcal{L}_C$ as follows:

- For each variable $X \in Dom(C)$, let P_X be a distinct unary predicate.

- Let $\mathcal{L}_C$ contain exactly the following rules:
  If $X_0 \in Dom(C)$ and "$\top$" is not a disjunct in $C(X_0)$, $\mathcal{L}$ should contain
  $$P_{X_1}(x_1) \wedge \ldots \wedge P_{X_n}(x_n) \rightarrow P_{X_0}(t\pi),$$
  where t is a disjunct in $C(X_0)$, $Var(t) = \{X_1, \ldots, X_n\}$, and $\pi = \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$.
  If "$\top$" is a disjunct in $C(X)$, then $\mathcal{L}_C$ should contain the rule $P_X(x)$.

Theorem 6.3 *Let C be an* SSC *with no cycles and no occurrences of "$\sqcap$". Then C represents the least Herbrand model of $\mathcal{L}_C$.*

Proof: We have to show that the least Herbrand model M_L of $\mathcal{L}_C$ corresponds to the least model M_C of C, that is, $M_C(X) = [\![P_X]\!]^{M_L}$ for all variables $X \in Dom(C)$.

Again, we know that M_L and M_C are unique, so it suffices to show that one of them is equivalent to the other. In this case, we show that the least model of C corresponds to the least Herbrand model of $\mathcal{L}_C$:

Let I be the least model of C. Then $I(X)$ is a subset of the Herbrand universe for all $X \in Dom(C)$. Let M be a model such that $[\![P_X]\!]^M = I(X)$ for all $X \in Dom(C)$. If "$\top$" is a disjunct in $C(X)$, then $P_X(x)$ is a rule in $\mathcal{L}_C$, so $I(X) = [\![P_X]\!]^{M_L}$ is equal to the Herbrand Universe. Assume now that "$\top$" does not appear in $C(X)$. Since I is a model of C, if $C(X) = e_1 \sqcup \ldots \sqcup e_n$, then:
$$[\![P_X]\!]^M = I(X) = \cup_{i=1}^n \{e_i\theta \mid Y_i\theta \in I(Y_i)\}.$$
By the construction of $\mathcal{L}_C$, this is equal to:
$$\{t\theta \mid (P_{X_1}(x_1) \wedge \ldots \wedge P_{X_n}(x_n) \rightarrow P_X(t)) \in \mathcal{L} \text{ and } x_i\theta' \in I(X_i)\}.$$

By definition, $I(X) = [\![P_X]\!]^M$ for all i, so

$$[\![P_X]\!]^M = \left\{ t\theta \mid (P_{X_1}(x) \wedge \ldots \wedge P_{X_n}(x_n) \rightarrow P_X(t)) \in \mathcal{L} \text{ and } x_i\theta' \in [\![P_{X_i}]\!]^M \right\},$$

and so M is the least Herbrand model M_L of $\mathcal{L}$. ∎

6.2 SSC's and Sorted Signatures

In this section, we examine the correspondence between SSC's, sorted signatures and their equivalent sort theories. The relativization of a sorted signature Σ is a logic program that satisfies the conditions outlined in the previous section. The algorithm outlined there can be used to translate this relativization (which is also a sort theory) into an SSC that represents its least Herbrand model. However, this C_Σ need not be a *shallow* SSC in solved form, so we cannot directly use the algorithm of Chapter 4 to solve Σ-unification problems. Section 6.3 presents a simple procedure for generating a shallow SSC that corresponds to a particular class of sorted signatures.

The set of predicate symbols in a sort theory Σ is the set of sort symbols, and the corresponding variables in an SSC are called **type variables**:

Definition 6.4 (Type Variables) *Given the set of sorts $\mathcal{S}_\Sigma$, we assign a unique* type variable *in $\mathcal{V}_\mathcal{S}$ for each sort $S \in \mathcal{S}_\Sigma$. The type variable for S is denoted by Υ_S.*

Note that $\Upsilon_{S_1} = \Upsilon_{S_2}$ iff $S_1 = S_2$. Type variables will *only* appear within SSC's, and we sometimes also use the notation $\Upsilon_{\langle\rangle}$ for other variables that appear only in SSC's.

In the following, we assume that we have a one-to-one function Π that maps $\mathcal{V}$ into $\mathcal{V}_\mathcal{S} - Var(C_\Sigma)$, given C_Σ; that is, Π is a substitution that renames normal variables into set variables that do not appear in C_Σ.

Definition 6.5 (SSC-relativization) *For any syntactic object ϕ, the* SSC-relativization *of ϕ, denoted by ϕ^{ssc}, is ϕ'/C_ϕ, where $\phi' = \phi\Pi$ and $C_\phi = \{\langle x_1\Pi \equiv \Upsilon_{S_1}\rangle, \ldots, \langle x_n\Pi \equiv \Upsilon_{S_n}\rangle\}$, for $Var(\phi) = \{x_1{:}S_1, \ldots, x_n{:}S_n\}$.*

Purely as a matter of notational convenience, we omit Π from the above formulation, so that ϕ^{ssc} will be denoted simply as ϕ/C_ϕ, with the implicit understanding that the variables in ϕ are now set variables that do not appear in C_Σ. For any variable $x{:}S \in \mathcal{V}$, the corresponding set variable in $\mathcal{V}_\mathcal{S}$ is written as x.

For example, $sum(x{:}\text{ODD}, y{:}\text{EVEN})^{\text{ssc}} = sum(x, y)/\{\langle x \equiv \Upsilon_{odd}\rangle, \langle y \equiv \Upsilon_{even}\rangle\}$. Note that the parameters in C_ϕ are always a subset of the type variables. These should be given an interpretation that satisfies the sort theory; Υ_S should range over the domain for sort S. Since there is no fixed interpretation for sorts, we simply restrict Υ_S to range over all the ground terms that we *know* to be of the sort S; that is, $\mathcal{T}_{\Sigma,T,gr}$. This is enforced by adding to C_ϕ a system of set constraints C_Σ that represents the least Herbrand model of Σ. Recall that $\mathcal{T}_{\Sigma,S} = \left\{ t \mid \Sigma \models \overline{\forall} S(t) \right\}$, and so $\mathcal{T}_{\Sigma,S,gr}$ is equal to $[\![S]\!]^M$, where M is the least Herbrand model of Σ. If C_Σ represents the least Herbrand model of Σ as in Definition 6.1, then the set of all ground terms of sort S, $\mathcal{T}_{\Sigma,S,gr}$, is equal to $M_{C_\Sigma}(\Upsilon_S)$.

Theorem 6.6 *Let t/C_t be the* SSC*-relativization of a term t and let C_Σ represent the least Herbrand model of Σ. Then $[t/C_t \cup C_\Sigma]$ is the set of Σ-ground instances of t.*

Proof: By the construction of t^{ssc}, $[t/C_t \cup C_\Sigma] = \{t\theta \mid x{:}S\theta \in C_\Sigma(\Upsilon_S)\}$. Since $C_\Sigma(\Upsilon_S) = \mathcal{T}_{\Sigma,S,gr}$, this set is equal to $\{t\theta \mid (x{:}S\theta) \in \mathcal{T}_{\Sigma,S,gr}\}$, that is, the set of Σ-ground instances of t. ■

The following theorem, finally, shows how sorted unification problems can be translated into equivalent SSC-unification problems:

Theorem 6.7 *Let E be a set of sorted equations under a sort theory Σ and let C_Σ be an* SSC *that represents the least model of Σ. Then a ground substitution ρ is a Σ-well-sorted unifier of E if, and only if, ρ solves the set of constrained equations $E/C_E \cup C_\Sigma$, where $E/C_E = E^{ssc}$.*

Proof: ρ is a Σ-well-sorted unifier of E iff ρ solves E and $\Sigma \models S(x{:}S\rho)$. This is the case iff ρ solves E and $x{:}S\rho \in [S/C_E \cup C_\Sigma]$, since $C_E(x{:}S) = \Upsilon_S$ and C_Σ represents the least Herbrand model of Σ. This is the case iff ρ solves $E/C_E \cup C_\Sigma$. ■

Therefore, we can solve a sorted unification problem E given Σ if we can find the solutions to the constrained equation set $E/C_E \cup C_\Sigma$, where C_Σ represents the least model of Σ.

Corollary 6.8 *Sorted unification is decidable for any sort theory whose least Herbrand model can be represented as a shallow* SSC *with no cycles.*

Proof: Given a sort theory Σ, an equation set E has a Σ-unifier iff E has a ground Σ-unifier (recall that we assume that for every sort there is a ground term that belongs to that sort). By Theorem 6.7, this is the case iff the constrained equation set $E/C_E \cup C_\Sigma$ has a solution, where $E^{ssc} = E/C_E$, which is the case iff the SSC-unification algorithm of Figure 4.2 returns an SSC-unifier of $E/C_E \cup C_\Sigma$. Thus, this algorithm can be used to decide if E is Σ-unifiable. ■

We will see in Chapter 7 that as far as obtaining a correct and complete deductive system based on the new unification algorithm for constrained terms, it is sufficient that we have a C_Σ that represents the least Herbrand model of Σ. However, C_Σ might not encode all the useful information contained in Σ. In particular, we would want Υ_S to represent not only $\mathcal{T}_{\Sigma,S,gr}$, but $\mathcal{T}_{\Sigma,S}$. The problem is that the set constraint $\langle \Upsilon_S \equiv exp \rangle$ should really express the fact that the set of objects of sort S is a *superset* of the terms described by exp. This can be "simulated" by adding a parameter to $C(\Upsilon_S)$ for each sort S, which can have an arbitrary interpretation for models of Σ other than the least Herbrand model.

Equivalently, the parameter for Υ_S could range over the set of all variables $x{:}S$ of sort S in the main language; we want to obtain an SSC C_Σ such that $M_{C_\Sigma}(\Upsilon_S)$ is equal to $\mathcal{T}_{\Sigma,S}$, the set of all terms of sort S in the sorted language. Therefore we define the following:

Definition 6.9 (**SSC's equivalent to** Σ) *Let Σ be a sorted signature and let C be an* SSC *whose domain includes the set of type variables for all sorts in Σ. Let C_P be obtained from C by defining $C_P(\Upsilon_S) \stackrel{\text{def}}{=} C(\Upsilon_S) \sqcup P_S$, where P_S is a distinct new parameter for the sort S, and $C_P(X) \stackrel{\text{def}}{=} C(X)$ if X is not a type variable. Let $\mathcal{A}$ be a parameter assignment that maps each parameter P_S into the set of variables of sort S. Then C is* equivalent *to Σ, or its relativization $\mathcal{A}_\Sigma$, if $M(\Upsilon_S) = \mathcal{T}_{\Sigma,S}$, where M is the unique model of C_P that extends $\mathcal{A}$.*

We will see in Section 6.3 that the SSC C_L defined in Section 6.1 is indeed equivalent to $\mathcal{L}$, if the logic program $\mathcal{L}$ is seen as a relativized sorted signature. Note that if an SSC C is equivalent to a sorted signature Σ or its relativization $\mathcal{A}_\Sigma$, then C also represents the least Herbrand model of $\mathcal{A}_\Sigma$.

Example 6.10 *Consider a sorted signature with two sorts, A and B, and a constant c, where $B \sqsubseteq A$ and c is declared to be of sort B. The logic program A_Σ corresponding to Σ is:*

$$\{\forall x(B(x) \to A(x)), \quad B(c)\}.$$

In the least Herbrand model M of $\mathcal{A}_\Sigma$, $[\![A]\!]^M = [\![B]\!]^M = [\![c]\!]^M = \{c\}$.
Thus, each of the following two SSC*'s represents the least Herbrand model of A_Σ:*

$$C_1 = \left\{ \ \langle \Upsilon_A \equiv \Upsilon_B \rangle, \quad \langle \Upsilon_B \equiv c \rangle \ \right\} \text{ and } C_2 = \left\{ \ \langle \Upsilon_A \equiv c \rangle, \quad \langle \Upsilon_B \equiv c \rangle \ \right\}$$

We say that C_1 is equivalent to Σ, while C_2 is not: C_1 can express the fact that B is a subset of A when the parameters P_A and P_B are added for each type variable, as in:

$$C_1' = \left\{ \ \langle \Upsilon_A \equiv \Upsilon_B \sqcup P_A \rangle, \quad \langle \Upsilon_B \equiv c \sqcup P_B \rangle \ \right\}$$

Note that for any *model I of C_1' (not necessarily the least model), $I(\Upsilon_B) \subseteq I(\Upsilon_A)$.*

Note that in the simplification and unification algorithms of Chapters 3 and 4 we were careful not to replace a variable X in $Rng(C)$ by $C(X)$ unless it was strictly necessary. This is intended to retain the "generality" of our new SSC's when some (or all) of the initial variables are taken to be sorts. The correctness of the unification algorithm shows that the output SSC-unifiers represent all the *ground* substitutions that solve the original set of sorted equations. These correspond to all the ground Σ-well-sorted unifiers, given that the initial SSC represents the least Herbrand model of $\mathcal{A}_\Sigma$. However, I now conjecture that the SSC-unification can be used to represent *all* of the Σ well-sorted unifiers:

Conjecture 6.11 *Let C_Σ be a shallow* SSC *with no cycles that represents a sorted signature Σ, and let S be a set of terms. Then the* SSC*-unifiers of $S/C_S \cup C_\Sigma$ can represent not only all the Σ-ground unifiers of S, but* also *all the Σ-most-general unifiers of S, when parameters are added for each type variable in each new* SSC *as described in Definition* 6.9.

In any case, a sufficient condition for obtaining a sound and (refutationally) complete deductive system, within the substitutional framework, is that all the ground substitutions are covered, as we show in Chapter 7.

6.3 Translation Algorithm

An SSC in solved form equivalent to a linear sorted signature is produced by the procedure outlined in Section 6.1. The algorithm presented here works for a broader class of theories, which we call **semi-linear**:

Definition 6.12 (Semi-linear) *A term t is* semi-linear *if (a) t is a constant or a variable, or (b) $t = f(t_1, \ldots, t_n)$, the subterms $t_1, \ldots, t_n$ are semi-linear, and if t_i and t_j share a variable for any $i \neq j$, then $t_i = t_j$. A sort theory is semi-linear if all its clauses are of the form $S_1(X_1) \wedge \ldots \wedge S_n(X_n) \to S_0(t)$, where $X_i \neq X_j$ for all $i \neq j$, and t is semi-linear.*

That is, a term t is semi-linear if both t and all its subterms have the property that repeated variables occur only in equal immediate subterms of t. For example, $f(g(x), g(x), y)$ is semi-linear, but $f(h(x, g(x)), y, h(x, g(x)))$ is not (since $h(x, g(x))$ is not.)

Input: The relativization, Σ, of a semi-linear sorted signature.
Output: A shallow SSC C equivalent to Σ.

Part 1:
Let $C = \{\langle \Upsilon_T \equiv \top \rangle\}$, where T is the top sort (i.e. $T(x)$ is a clause in Σ).
FOR EACH clause $S_1(x_1) \wedge \ldots \wedge S_n(x_n) \rightarrow S_0(t)$ in Σ, DO:
  Let $\theta = \{x_1 \mapsto N_1, \ldots, x_n \mapsto N_n\}$ for new variables N_i.
  Replace $\langle \Upsilon_{S_0} \equiv exp \rangle$ in C by $\langle \Upsilon_{S_0} \equiv exp \sqcup t\theta \rangle$.
  Add $\langle N_i \equiv \Upsilon_{S_i} \rangle$ to C, for $1 \leq i \leq n$.

Part 2:
Exhaustively apply the following transformation to C:

- Consider a disjunct in $Rng(C)$ of the form $f(t_1, \ldots, t_n)$, $n \geq 1$, where not all the t_i's are variables.
  FOR EACH distinct non-variable term t in $\{t_1, \ldots, t_n\}$, DO:
    Let Y_t be a new variable.
    FOR EACH i, $1 \leq i \leq n$, DO:
      If $t_i = t$, replace t_i in $f(t_1, \ldots, t_n)$ by Y_t.
    Add $\langle Y_t \equiv t \rangle$ to C.

Figure 6.1: Translation algorithm

A sorted signature Σ is semi-linear if its relativization $\mathcal{A}_\Sigma$ is semi-linear, or, equivalently, if for all term declarations $S[t]$ in Σ, t is semi-linear. A set expression is semi-linear if it is "$\top$", "$\bot$", a semi-linear term, or is made up of semi-linear set expressions. An SSC C is semi-linear if $C(X)$ if semi-linear for all $X \in Dom(C)$. (Note that any shallow SSC is trivially semi-linear.)

Figure 6.1 shows an algorithm for building a *shallow* SSC C_Σ equivalent to a semi-linear sorted signature Σ. We assume that Σ includes a top sort, that the subsort ordering $\sqsubseteq_\Sigma$ is a partial order, and that there is a term declaration $S[k_S]$ for each sort $S \in \mathcal{S}_\Sigma$. To avoid the distinction between term, function and subsort declarations, the signature is expressed as the equivalent sort theory, which is a set of Horn clauses (see Section 5.3). Note that if $S(t)$ is a positive unit clause in Σ, then either t is ground or a variable x. In the first case, Part 1 simply adds the disjunct t to $C(\Upsilon_S)$. In the second, S is the top sort, and $C(\Upsilon_S) = \top$. Also, we assume that any clauses of the form $S(x) \rightarrow S(x)$ have been removed from Σ.

Part 1 is equivalent to the translation of a logic program into an SSC described in Section 6.1; therefore, we know that the SSC C at the end of Part 1 represents the least model of Σ.

Recall that to ensure that no sort is empty we introduce a Skolem constant k_{S_i} for each sort S_i. This will simplify some of the following proofs. As pointed out by Schmidt-Schauß, the emptyness of sorts "*...is more of a problem at a meta-level and should not be confused with the pure sorted calculus ...the proofs of the nonemptyness of sorts could be carried out in a preprocessing step and afterwards the deduction system would have a solid basis*" [Schmidt-Schauß, 1989, page 20]. We could always start out with $C(\Upsilon_{S_i})$ set to "$\bot$" for all the type variables, and then, at the end, apply the algorithms in Section 3.3 to identify and simplify the occurrences of empty variables in C; this would identify empty sorts.

Some simple invariants for the translation algorithm are:

- All the elements of $Rng(C)$ are of the form $t_1 \sqcup \ldots \sqcup t_n$: This is trivially true at the start of the algorithm. In Part 1, the set expressions in $Rng(C)$ are always obtained by taking the union of $t_1 \sqcup \ldots \sqcup t_m$ and a term, so the invariant still holds true at the start of Part 2. It is maintained during Part 2, since subterms are replaced by variables and only set constraints of the form $\langle Y_t \equiv t \rangle$ are added to C.

  Therefore, assuming that the algorithm halts, the output is a *shallow* SSC in solved form; if the transformation in Part 2 is no longer applicable, then the disjuncts in $C(X)$ can only contain variables as subterms.

- C is semi-linear: Again, this trivially holds at the start of the algorithm. It should also be clear that the SSC C at the end of Part 1 is semi-linear, given that Σ was semi-linear too. Also, the transformation in Part 2 preserves the semi-linearity of C, since subterms are uniformly replaced by new variables and the subterms themselves by definition have to be semi-linear.

  Similarly, if the input sort theory corresponds to a linear signature, then the output SSC is linear.

- The output SSC has no parameters: At the end of Part 1, a set constraint has been added for each variable in $Var(Rng(C))$, and all the variables introduced in Part 2 are constrained variables.

- Part 2 can not introduce any cycles in C, since all new variables appear nested in $Rng(C)$.

Finally, we have:

Lemma 6.13 *If $\sqsubseteq_\Sigma$ is a partial ordering, then the* SSC *output by the translation algorithm of Figure* 6.1 *has no cycles.*

Proof: Consider the SSC C at the end of Part 1. A cycle in C could only involve type variables, since all auxiliary variables appear nested in $Rng(C)$. Since C has no "$\sqcap$" symbols, a cycle in C would have to be a sequence of variables $\langle \Upsilon_{S_1}, \ldots, \Upsilon_{S_n} \rangle$, for $n \geq 2$, such that $\Upsilon_{S_n} = \Upsilon_{S_1}$ and $\Upsilon_{S_{i+1}}$ is a disjunct in $C(\Upsilon_{S_i})$ for $1 \leq i \leq n-1$. Then Σ would contain the sequence of clauses $S_i(x) \rightarrow S_{i+1}(x)$. Since we assume that Σ contains no clauses of the form $S(X) \rightarrow S(X)$, then $S_1 \neq S_2$. Therefore, $\Sigma \models \forall x (S_i(x) \leftrightarrow S_j(x))$ for $S_i \neq S_j$, so $\sqsubseteq_\Sigma$ would not be a partial ordering, a contradiction. Therefore, C contains no cycles. Since the transformation in Part 2 cannot introduce any new cycles, the output SSC contains no cycles either. ∎

6.3.1 Termination

Part 1 clearly terminates, since Σ is finite. Each application of the transformation in Part 2 terminates, too, since both FOR loops can execute at most n times.[4]

[4] Obviously, this formulation of the algorithm is *not* the most efficient one.

All that is left to show is that the transformation can only be applied a finite number of times. The following is a formal way of showing that the transformation can be applied at most as many times as there are function and constant symbols in the original $Rng(C)$:

Let $F(t)$ be the sum of $d(t')$ over all the subterms t' of t (treating the set of subterms as a multiset). Define $\alpha(C)$ to be the sum over all $V \in Dom(V)$ of $\alpha(V,C) = \sum_{i=1}^{i=n} F(t_i)$, where $C(V) = t_1 \sqcup \ldots \sqcup t_n$. Each time the transformation is applied, $\alpha(C)$ is reduced:

Let $C(X)$ be the transformed set expression and let $t = f(t_1, \ldots, t_n)$ be the transformed disjunct. Define $D(t) = d(t) + 1$, where $d(t)$ is the depth of term t; $F(t) = D(t) + F(t_1) + \ldots + F(t_n)$. Let $t' = f(t'_1, \ldots, t'_n)$ be the new term that replaced t. Since the t'_i are all variables, $\alpha(t') = 1$. Now, for each t_i at most one new set constraint $\langle \Upsilon_i \equiv t_i \rangle$ is generated. $\alpha(\Upsilon_i, C') = F(t_i)$, where C' is the new SSC. Thus:

$$
\begin{array}{rcl}
\alpha(C) - \alpha(C') & = & \alpha(C, X) - (\alpha(C', X) + \sum_i \alpha(\Upsilon_i, C')) \\
& \geq & D(t) + F(t_1) + \cdots + F(t_n) - 1 - F(t_1) - \cdots - F(t_n) \\
& = & D(t) - 1
\end{array}
$$

Not all the subterms of t are variables, so $d(t) \geq 1$ implies $D(t) \geq 2$ implies $\alpha(C) - \alpha(C') \geq 1$.

6.3.2 Correctness

We claim that the SSC output by the translation algorithm is in solved form and represents Σ, in the sense of Definition 6.9.

Lemma 6.14 *Let C be the* SSC *produced at the end of Part 1 of the translation algorithm of Figure 6.1. $S_1 \sqsubseteq_\Sigma S_2$ if, and only if, there is a derivation from Υ_{S_1} of Υ_{S_2} in C ($\Upsilon_{S_1} \overset{C}{\leadsto} \Upsilon_{S_2}$).*

Proof: The distinct Skolem constants simplify the proof: $S_1 \sqsubseteq_\Sigma S_2$ iff $\Sigma \models \forall x(S_1(x) \rightarrow S_2(x))$ iff $\Sigma \models S_2(k_{S_1})$, where k_{S_1} is the Skolem constant for the sort S_1. Since C represents the least model of Σ, this is the case iff $k_{S_1} \in M_C(S_2)$, which is the case iff $S_2 \overset{C}{\leadsto} k_{S_1}$. But the only place where k_{S_1} appears in $Rng(C)$ is in $C(\Upsilon_{S_1})$, so this is equivalent to saying that $\Upsilon_{S_2} \overset{C}{\leadsto} \Upsilon_{S_1}$. ■

Notice that the above proof also holds if C contains the parameters P_S described in Definition 6.9 and the derivations are relative to the parameter assignment $\mathcal{A}$. Recall that there is some derivation from E of t in C (written as $E \overset{C}{\leadsto} t$) iff for any parameter assignment $\mathcal{A}$, that derivation is in C relative to $\mathcal{A}$ (written as $E \overset{C,\mathcal{A}}{\leadsto} t$).

Theorem 6.15 *The* SSC *C produced by Part 1 of the translation algorithm of Figure* 6.1 *is equivalent to Σ.*

Proof: Assume that the parameters P_S have already been added to $C(\Upsilon_S)$, and let $\mathcal{A}$ map P_S into the set of variables $x{:}S$ of sort S. Here we find the characterization of $\mathcal{T}_{\Sigma,S}$ in [Schmidt-Schauß, 1989] useful: Recall from Theorem 5.6 that any term in $\mathcal{T}_{\Sigma,S}$ can be obtained from applying a Σ-well-sorted substitution to a term declared to be of sort S. We now show, by induction on the term t, that $t \in \mathcal{T}_{\Sigma,S}$ if, and only if, $S \overset{C,\mathcal{A}}{\leadsto} t$.

<u>Basis</u>: (i) t is a constant:

"Only if": If $t \in \mathcal{T}_{\Sigma,S}$ then for some $S' \sqsubseteq_\Sigma S$, $S'(t)$ is a clause in Σ; by the construction of C, then t is a disjunct in $C(\Upsilon_{S'})$. By Lemma 6.14 $S' \sqsubseteq_\Sigma S$ iff $\Upsilon_S \overset{C}{\mapsto} \Upsilon_{S'}$, so $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} t$.
"If": If $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} t$, then, since t is ground and $\mathcal{A}$ only maps parameters into variables, it has to be the case that for some $S' \in Dom(C)$, t is a disjunct in $C(S')$ and $\Upsilon_S \overset{C}{\mapsto} \Upsilon_{S'}$. Then $S'(t)$ is a clause in Σ and by Lemma 6.14, $S' \sqsubseteq_\Sigma S$, so $t \in \mathcal{T}_{\Sigma,S}$.

(ii) t is a variable, $x{:}S'$: By definition of $\mathcal{A}$, $x{:}S' \in \mathcal{A}(P_{S'})$, where $P_{S'}$ is the parameter for sort S', so $S' \overset{C,\mathcal{A}}{\mapsto} t$. Therefore, $t \in \mathcal{T}_{\Sigma,S}$ iff $S' \sqsubseteq_\Sigma S$ iff (by Lemma 6.14) $\Upsilon_S \overset{C}{\mapsto} \Upsilon_{S'}$ iff (by the construction of C and $\mathcal{A}$) $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} t$.
Induction Step: Assume that the hypothesis is true for $t_1, \ldots, t_n$, and let $t = f(t_1, \ldots, t_n)$.

"Only if": From Theorem 5.6, if $t \in \mathcal{T}_{\Sigma,S}$ then for some $S' \sqsubseteq S$, there exists a well-sorted substitution θ such that $t = d\theta$ for some term declaration $S'[d]$ in Σ. By the construction of C, then $C(\Upsilon_{S'}) = d\rho \sqcup exp$, where $\{x_1{:}S_1, \ldots, x_m{:}S_m\} = Var(d)$, ρ is the substitution $\rho = \{x_1{:}S_1 \mapsto N_1, \ldots, x_m{:}S_m \mapsto N_m\}$, and $C(N_i) = \Upsilon_{S_i}$. Therefore, $t = (d\rho)\theta'$, where $N_i\theta' \in \mathcal{T}_{\Sigma,S_i}$ for each i, $1 \leq i \leq n$. We can now apply the induction hypothesis to the subterms of t, so we know that $N_1\theta' \in \mathcal{T}_{\Sigma,S_i}$ iff $\Upsilon_{S_i} \overset{C,\mathcal{A}}{\mapsto} N_i\theta$. But since $C(N_i) = \Upsilon_{S_i}$, then trivially $N_i \overset{C}{\mapsto} \Upsilon_{S_i}$. Therefore, $N_i \overset{C,\mathcal{A}}{\mapsto} N_i\theta$, which is the case iff $d\rho \overset{C,\mathcal{A}}{\mapsto} t$ so $\Upsilon_{S'} \overset{C,\mathcal{A}}{\mapsto} t$. Since $S' \sqsubseteq_\Sigma S$, $\Upsilon_S \overset{C}{\mapsto} \Upsilon_{S'}$, by Lemma 6.14, so $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} t$.

"If": If $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} t$, then by the construction of C (and since the range of $\mathcal{A}$ are sets of only variables) there exists $\Upsilon_{S'} \in Dom(C)$ such that $\Upsilon_S \overset{C,\mathcal{A}}{\mapsto} \Upsilon_{S'}$ and $C(\Upsilon_S)$ contains a disjunct t' such that $t'\rho = t$, where $\rho = \{N_1 \mapsto t_1, \ldots, N_m \mapsto t_m\}$ and for each i, $1 \leq i \leq m$, $C(N_i) = \Upsilon_{S_i}$, where $\Upsilon_{S_i} \overset{C,\mathcal{A}}{\mapsto} t_i$. We can then apply the induction hypothesis: $t_i \in \mathcal{T}_{\Sigma,S_i}$. By the construction of C, there exists a well-sorted substitution $\theta = \{x_1{:}S_1 \mapsto t_1, \ \ldots, \ x_m{:}S_m \mapsto t_m\}$ and a term declaration $S'[d]$ such that $t = d\theta$. Since, again, $S' \sqsubseteq_\Sigma$ from Lemma 6.14, then $t \in \mathcal{T}_{\Sigma,S}$. ∎

Note that the above two theorems do not depend on the semi-linearity of Σ; that is, they show that for *any* signature Σ whose relativization satisfies the simple conditions outlined in Section 6.1, the SSC obtained by Part 1 of the translation algorithm is equivalent to Σ. We now show that the transformation of Part 2 preserves the models of C.

Lemma 6.16 *Let C be a semi-linear* SSC *with no cycles and let C' be the* SSC *obtained by applying to C the transformation in Part 2 of the translation algorithm of Figure 6.1. Then $C \overset{Dom(C)}{\Longleftrightarrow} C'$.*

Proof: We already showed that the transformation introduces no new parameters and no cycles. Since both C and C' have no cycles and the same parameters, by Corollary 2.19 they both have a unique model that extends any given parameter assignment. It suffices to show that any model I of C can be extended to a model I' of C' such that $I(X) = I'(X)$ for all $X \in Dom(C)$:

Let $\langle V \equiv t \sqcup exp\rangle$ be the transformed set constraint in C, where $t = f(t_1, \ldots, t_n)$ is the transformed term. Let $\langle V \equiv t' \sqcup exp\rangle$ be the new set constraint for V. Given a model I of C, we construct an interpretation I' as follows: $I'(X) = I(X)$ if $X \in Dom(C)$, and let $I'(\Upsilon_t) = I(t)$ if Υ_t is a new variable and $\langle \Upsilon_t \equiv t\rangle \in C'$.

Clearly, I' satisfies all the set constraints in C' except perhaps for $\langle V \equiv t' \sqcup exp \rangle$. We now show that $I'(t') = I(t)$, and so I' also satisfies this set constraint:

$$I(t) = \{t\theta \mid X\theta \in I(X)\} = \{f(t_1, \ldots, t_n)\theta \mid X\theta \in I(X)\} = \{f(t_1\theta, \ldots, t_n\theta) \mid X\theta \in I(X)\}$$

Let $t'_1, \ldots, t'_m$ be the distinct terms in $\{t_1, \ldots, t_n\}$. Since t is semi-linear, $Var(t'_i) \sqcap Var(t'_j) = \emptyset$ if $t_1 \neq t_j$; $t' = f(V_1, \ldots, V_n)$, where $V_i = \Upsilon_{t'_j}$ if $t_i = t'_j$. Therefore,

$$I(t) = \{f(g_1, \ldots, g_n) \mid g_i \in I(t_i) \text{ and } g_i = g_j \text{ if } t_i = t_j\}.$$

By definition, $I'(\Upsilon_{t'_i}) = I(t'_i)$, so $I'(t') = \{f(V_i, \ldots, V_n)\theta \mid V_i\theta \in I'(V_i)\}$ is equal to $I(t)$.

Therefore, I' is a model of C', and $I(X) = I'(X)$ for all $X \in Dom(C)$. ■

Corollary 6.17 *The* SSC C *output by the translation algorithm of Figure* 6.1 *is equivalent, in the sense of Definition* 6.9, *to* Σ.

Proof: Lemma 6.2 implies that the only model of the SSC C_0 at the end of Part 1 is equivalent to Σ. From Lemma 6.16, the unique model of C that extends any given parameter assignment A is equal to the model of C_0 that extends A, over $Dom(C_0)$. This includes, in particular, the parameter assignment $\mathcal{A}$ used in Definition 6.9. Therefore, C is also equivalent to Σ. ■

Therefore, if C_Σ is the output of the translation algorithm, then for any sorted term t, $[t/C_t \cup C_\Sigma]$ is precisely the set of all Σ-ground instances of t, and, furthermore, $\Sigma \models S_i(t)$ iff $I(t) \subseteq I(\Upsilon_i)$ for all models I of $C_t \cup C_\Sigma$.

6.3.3 Remarks

The above translation algorithm can be extended to include more general kinds of signatures. For example, if Σ contained a clause $\neg S_e(x)$, we can add $\langle \Upsilon_{S_e} \equiv \bot \rangle$ to C_Σ. Empty variables in C can be simplified after the translation is done, using the algorithms in Section 3.3. Obviously, not all signatures can be represented as a shallow SSC in solved form, since sorted unification is, in general, undecidable.[5] Things get more complicated if we allow arbitrary sort theories, which are not the relativization of any sorted signature and could include "auxiliary" non-unary predicates. The unification algorithm can be used for any sort theory that can be expressed as a (finite) shallow SSC. Program approximation algorithms such as the one presented in [Heintze and Jaffar, 1990] can be useful in these cases, if they are exact. Heintze and Jaffar's algorithm can be viewed as preprocessing the sort theory so that inferences from it can be carried out more efficiently from then on; we obtain a more explicit representation of the information originally implicit in Σ.

The extra variables in the SSC's correspond to the intersection of sorts and subsets of sorts. The new variables can be added to C_Σ as new intersections of sorts are computed. Note that C_Σ is shared by all the constrained terms. The new sorts are added as they are needed, and can be used from then on.

[5] Representability of Σ as a shallow SSC is a *sufficient* condition for decidability of Σ-unification, but it is not necessary.

6.3.4 Inverse Relativization, Part 2

Given a set of clauses, we can "extract" a sort theory if we can identify a set of unary predicates P that satisfy the following conditions:

1. All negative literals from S are of the form $\neg P(X)$ for a variable X, and no clause contains only negative literals from S.

   If a clause in S contains the negative literals $\neg P_1(X_1), \ldots, \neg P_n(X_n)$ from P (and no positive literal from P) then $X_i \neq X_j$ if $P_i \neq P_j$.[6]

2. The union of the clauses where a P occurs positively forms a theory (the sort theory), whose least Herbrand model can be expressed as a shallow system of set constraints. This set of clauses might contain extra predicates not in S (could be n-ary predicates), as long as those predicates do not appear in the rest of the clause set.[7]

   This is the case if the logic program corresponds to a semi-linear signature, or if the algorithm in [Heintze and Jaffar, 1990] can be used to obtain an *exact* representation of its least Herbrand model.

If we wanted to directly use the translation algorithm of Figure 6.1, then a predicate $P \in S$ can appear positively in a clause only if the clause is of form $\neg P_1(x_1) \vee \ldots \vee \neg P_n(x_n) \vee P(t)$, where $x_i \neq x_j$ if $P_i \neq P_j$ and t is semi-linear.

6.4 Examples of Sorted Unification Using SSC's

Figure 6.2 shows a sort hierarchy that is not tree-structured, with sorts A, B, C and D, where $C \sqsubseteq A$, $C \sqsubseteq B$, $D \sqsubseteq A$ and $D \sqsubseteq B$. Let Σ be a monomorphic sort theory under this hierarchy.

A ΣCIU for the set $\{x{:}A,\ y{:}B\}$ is:

$$\{\{x{:}A \mapsto z{:}C,\ y{:}B \mapsto z{:}C\},\ \{x{:}A \mapsto z{:}D,\ y{:}B \mapsto z{:}D\}\}$$

Note that the variables of concern are x and y. In general, if A and B had n different greatest lower bounds, a ΣCIU for $\{x{:}A,\ y{:}B\}$ would have size n. In [Schmidt-Schauß, 1989], this sort hierarchy is used as an example of how the number of incomparable Σ-mgu's can grow exponentially in the size of the terms to be unified: Consider the terms $t_1 = f(x_1{:}A, \ldots, x_n{:}A)$ and $t_1 = f(y_1{:}B, \ldots, y_n{:}B)$.

All the elements of a ΣCIU for $\{t_1,\ t_2\}$ are substitutions of the form

$$\{x_1{:}A \mapsto z_1,\ y_1{:}B \mapsto z_1,\ \ldots,\ x_n{:}A \mapsto z_n,\ y_n{:}B \mapsto z_n\}$$

for distinct new variables $\{z_1, \ldots, z_n\}$. Observe that z_i can be either of sort C or sort D for each i, $1 \leq i \leq n$. Since the sorts C and D are incomparable, this means that a ΣCIU of t_1 and t_2 will be of size 2^n.

[6] This is necessary so that the clause can be relativized in the usual way. I conjecture that this could be extended to allow more than one sortal constraint on one variable, and furthermore, literals of the form $P(t)$ for arbitrary terms t. The relativization would then have to make use of the constraint unification algorithm of Chapter 4 to find the corresponding constraints on the variables appearing in the rest of the clause.

[7] If they did, this translation does not work. Finding a translation that does is left as an open problem.

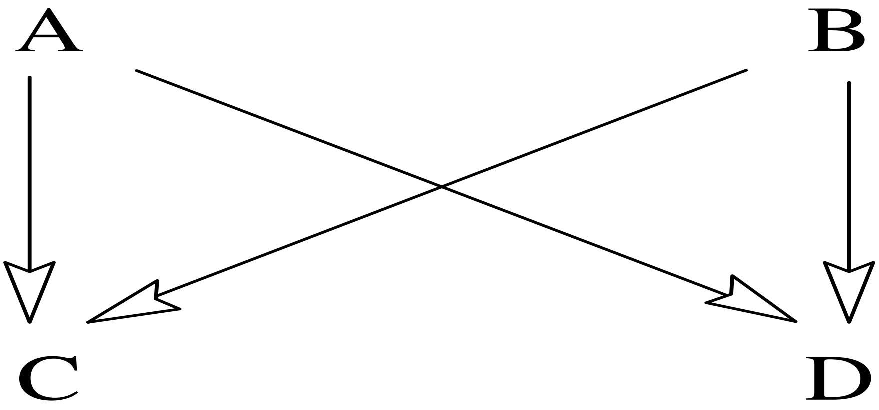

Figure 6.2: Sort hierarchy for a monomorphic signature of type finitary

Schmidt-Schauß points out how here "*...we have the curious situation that the number of unifiers grows exponential, but [the unification problem] can be solved in quasi-linear time, i.e. a representation for all unifiers can be computed in quasi-linear time.*" [1989, page 108]

Indeed, if we define a new sort N that is the intersection of A and B, then a ΣCIU for $\{t_1, t_2\}$ has only one element, of the form:

$$\{x_1{:}A \mapsto z_1{:}N,\ y_1{:}B \mapsto z_1{:}N, \ldots,\ x_n{:}A \mapsto z_n{:}N,\ y_n{:}B \mapsto z_n{:}N\}.$$

The sort N can be defined adding the declarations $N \sqsubseteq A$, $N \sqsubseteq B$, $C \sqsubseteq N$ and $D \sqsubseteq N$. This is done by the *unification over the weakest range* proposed by Irani and Shin [1985] for monomorphic sorted signatures, where the sort hierarchy is dynamically embedded into a complete lattice,[8] thus guaranteeing singleton ΣCIU's. The new sorts are only added to the signature when the need for them arises.

Similarly, the SSC-unification algorithm adds new set variables to SSC's as needed, verifying that they are not empty; it will, indeed, unify t_1 and t_2 above in time polynomial in n. A solved form SSC that represents Σ is:

$$C_\Sigma = \left\{ \begin{array}{c} \langle \Upsilon_A \equiv \Upsilon_C \sqcup \Upsilon_D \sqcup k_A \rangle, \\ \langle \Upsilon_B \equiv \Upsilon_C \sqcup \Upsilon_D \sqcup k_B \rangle, \\ \langle \Upsilon_C \equiv k_C \rangle, \langle \Upsilon_D \equiv k_D \rangle \end{array} \right\}$$

and the SSC-relativization of $\{t_1, t_2\}$ is:

$$\{t_1, t_2\} \ / \qquad \{\langle x_i \equiv \Upsilon_A \rangle\} \cup \{\langle y_i \equiv \Upsilon_B \rangle\} \cup C_\Sigma.$$

The SSC unification algorithm of Figure 4.2 will produce a single SSC-unifier $\langle \Theta, C' \rangle$, where

[8] Every set of sorts S will have a greatest upper bound sort that represents the intersection of the sorts in S.

$$\Theta = \{x_i \mapsto N_i,\ y_i \mapsto N_i\},$$

and the new SSC is

$$C' = \{\langle N_i \equiv \Upsilon_C \sqcup \Upsilon_D\rangle \cup C_\Sigma\}.$$

This is done in $O(n)$ time.

The idea of adding new sorts to the sort hierarchy in order to make it into a complete lattice is often proposed. However, this only eliminates multiple incomparable Σ-most-general unifiers if the original signature is *monomorphic*. Polymorphic signatures can be of type infinitary even if the sort hierarchy is a complete lattice. For example, the set of restrictions on a variable x that ensures that a term $f(x)$ belongs to a given sort may be impossible to express as the union and intersection of existing sorts (or even as an arbitrary boolean combination of them). Cohn and Frisch [1991] present the following example of an infinite set of Σ-MGCU's:

$$\begin{array}{rcl} \Sigma & = & \{\forall x\ \mathrm{T}(i(x)) \rightarrow \mathrm{T}(i(s(x))),\ \mathrm{T}(i(a))\} \\ S & = & \{z{:}\mathrm{T},\ i(s(y))\} \end{array}$$

The set S has an infinite number of Σ-most general unifiers, none of which is strictly Σ-more general than any other:

$$\left\{ \begin{array}{cc} \{y \mapsto a,\ z{:}\mathrm{T} \mapsto i(s(a))\}, & \{y \mapsto s(a),\ z{:}\mathrm{T} \mapsto i(s(s(a)))\},\ \ldots \\ \{y \mapsto s^n(a),\ z{:}\mathrm{T} \mapsto i(s(s^n(a)))\}, & \ldots \end{array} \right\}$$

That is, Σ is of type infinitary. The following solved form SSC represents Σ:

$$C_\Sigma = \left\{ \begin{array}{c} \langle \Upsilon_{Top} \equiv \top\rangle, \\ \langle \Upsilon_T \equiv i(V) \sqcup k_T\rangle, \\ \langle V \equiv a \sqcup s(V)\rangle \end{array} \right\}$$

Unifying $i(s(y))$ and z under the constraint

$$\{\langle y \equiv \Upsilon_{Top}\rangle,\ \langle z \equiv \Upsilon_T\rangle\} \cup C_\Sigma$$

results in the single SSC-unifier

$$\langle \{y \mapsto N,\ z \mapsto i(s(N))\},\ \{\langle N \equiv V\rangle\} \cup C_\Sigma\rangle.$$

Thus, an infinite ΣCIU can be represented using a single SSC-substitution. Note that the sort represented by the new variable N is not equal to any boolean combination of the original sorts. See Section 7.3 for another simple example of how SSC-unification can represent infinite ΣCIU's finitely.

Chapter 7

Resolution and Constraints

This chapter shows how the unification algorithm of Chapter 4 can be integrated into a resolution-based deductive system. The proofs of the soundness and completeness of the resulting hybrid system follow the proof outline suggested by Frisch's [1991] *substitutional framework* for sorted deduction, which presents the theoretical basis for building such systems: Section 7.1 briefly reviews normal resolution and the proof of its soundness and completeness. Section 7.2 shows how these results can be adapted to the case where SSC's are used as constraints. Section 7.3 presents a simple example of a derivation using terms constrained by SSC's.

7.1 The Resolution Principle

The *resolution* rule of inference was introduced by Robinson [1965] and has been widely used ever since in a large variety of systems; countless refinements and extensions for it have been proposed. We now very briefly describe it; for a formal and detailed discussion of resolution and associated topics, see [Chang and Lee, 1973], for example.

Recall that a clause is a *set* of literals, interpreted as their logical disjunction, while a set of clauses represents the logical conjunction of its clauses. Following Frisch [1991], we will treat clauses as *multisets*, so that they can possibly contain repeated occurrences of the same literal. In this way, all the ground instances of any given clause have the same number of clauses as the original; for example, $\{P(a), P(a)\}$ is a ground instance of $\{P(x), P(y)\}$, but $\{P(a)\}$ is not. Thus, when we talk about "sets" of literals we will allow the possibility of repeated occurrences of literals within them. This is necessary for the simple proofs of soundness and completeness of resolution presented here to hold. If S is a set of clauses, the set containing all the ground instances of all clauses in S is denoted by S_{gr}. The symbol "$\square$" denotes the empty clause, which is satisfied by no model (i.e. is always false). Thus we write $S \models \square$ if S is unsatisfiable. The *complement* of a literal l is denoted by $\bar{l}$; i.e. if $l = P(t_1, \ldots, t_n)$, then $\bar{l} = \neg P(t_1, \ldots, t_n)$, and if $l = \neg P(t_1, \ldots, t_n)$, then $\bar{l} = P(t_1, \ldots, t_n)$. For a set of literals S, $\overline{S}$ denotes the set containing the complements of the elements of S.

Resolution can now be described as follows:

> **Resolution**: From clauses $c_1 = A \cup S_1$ and $c_2 = S_2 \cup B$, derive the clause $c_3 = (A \cup B)\theta$, where θ is the most general unifier of the set of literals $S_1 \cup \overline{S_2}$. The clause c_3 is called a *resolvent* of c_1 and c_2.

The variables in one of the two clauses being resolved should be renamed, if necessary, so that the two clauses are variable-disjoint.

Definition 7.1 (Derivations, $\vdash_{\overline{\mathrm{RES}}}$) *A* resolution derivation *of a clause c from a set of clauses S is a finite binary tree whose root is c, whose leaves are clauses in S, and where each internal node is a resolvent of its two children nodes. We write $S \vdash_{\overline{\mathrm{RES}}} c$ if there is a resolution derivation of c from S.*

Resolution is clearly a *sound* rule of inference: If $S \vdash_{\overline{\mathrm{RES}}} c$, then $S \models c$. The following theorem, proved in [Robinson, 1965], states that resolution for ground clauses is also *refutationally complete*:

Proposition 7.2 (Ground Resolution Theorem [Robinson, 1965]) *Let S be any finite set of ground clauses. Then S is unsatisfiable if, and only if, there is a resolution derivation of $\Box$ from S. That is, $S \models \Box$ if, and only if, $S \vdash_{\overline{\mathrm{RES}}} \Box$.*

In conjunction with the *Herbrand Theorem* and a *Lifting Lemma*, this is the basis for proving the completeness of a full first-order resolution-based deductive system. The Herbrand Theorem [Herbrand, 1930] states, in its simplest form:[1]

Theorem 7.3 (Herbrand Theorem) *A set of clauses S is unsatisfiable if, and only if, there is a finite unsatisfiable set S' of ground instances of clauses in S.*

A set of ground clauses is unsatisfiable if, and only if, it has a finite unsatisfiable subset. Therefore, the Herbrand Theorem can be rephrased as follows:

$$S \models \Box \text{ if, and only if, } S_{gr} \models \Box.$$

The Lifting Lemma shows that if there is a resolvent of the ground instances of two clauses, then the clauses themselves can be resolved and the resulting clause is more general than the original ground resolvent. Therefore, any derivation at the ground level can be "lifted" to obtain a corresponding derivation from the original clauses. The Lifting Lemma for resolution can be stated as follows:

Proposition 7.4 (Lifting Lemma for Resolution) *If S is a set of clauses and c' is a ground clause such that $S_{gr} \vdash_{\overline{\mathrm{RES}}} c'$, then there is a clause $c \geq c'$ such that $S \vdash_{\overline{\mathrm{RES}}} c$.*

The proof shows that a single resolution step "lifts:" If $c1$ and $c2$ are clauses with ground instances $c1'$ and $c2'$, and $c3'$ is a resolvent of $c1'$ and $c2'$, then there is a clause $c3$ such that $c3 > c3'$ and $c3$ is a resolvent of $c1$ and $c2$. This is proved based on the properties of substitutions and most general unifiers. Similarly, the converse of the Lifting Lemma also holds: If a clause c_3 is a resolvent of c_1 and c_2, then some ground instance of c_3 is a resolvent of ground instances of c_1 and c_2.

Note that every clause has a ground instance (we assume that the Herbrand Universe is not empty) and that we treat clauses as multisets. If clauses are treated as sets, then the converse of the Lifting Lemma might not apply, since there would be derivations at the non-ground level

[1] This is the formulation found in [Chang and Lee, 1973, page 61].

that do not lift any ground derivations. Also, note that the only ground instance of $\Box$ is $\Box$ itself, and the only clause that has $\Box$ as a ground instance is $\Box$. Thus, the property of being a derivation of $\Box$ is preserved by lifting and its inverse operation. Thus we have Robinson's [1965] *Resolution Theorem*, which states that resolution is both sound and (refutationally) complete:

Theorem 7.5 (Resolution Theorem) *A finite set of clauses S is unsatisfiable if, and only if, $S \vdash_{\overline{\text{RES}}} \Box$.*

Proof: By the Herbrand Theorem, S is unsatisfiable iff S_{gr} is unsatisfiable. By the soundness and completeness of ground resolution, S_{gr} is unsatisfiable iff $S_{gr} \vdash_{\overline{\text{RES}}} \Box$. By the Lifting Lemma, $S_{gr} \vdash_{\overline{\text{RES}}} \Box$ iff $S \vdash_{\overline{\text{RES}}} \Box$. ■

7.2 SSC-Resolution

As pointed out by Frisch [1991], the resolution rule combines two fundamental ideas: The first is that at the ground level it detects "*local evidence of unsatisfiability*" by "resolving away" complementary literals. The second idea is to treat variables as "*schematic for the set of all its ground instances.*" The first idea is generalized by *theory resolution* [Stickel, 1985], while the second idea is generalized by the introduction of sorts: A variable is now schematic for the set of all its *well-sorted* ground instances. When SSC's are used to represent sortal constraints, the relationship is particularly direct: The SSC associated with a clause will generate exactly the set of its well-sorted ground instances.

As presented in [Burckert, 1990], the resolution principle can be extended to arbitrary constraints as follows:

> From $(A \cup S_1)/C_1$ and $(S_2 \cup B)/C_2$ (whose variables have been renamed apart as necessary), infer $(A \cup B)/C_3$, where $\{t_1, \ldots, t_n\} = S_1 \cup \overline{S_2}$ and $C_3 = C_1 \wedge C_2 \wedge t_1 = t_2 \wedge \ldots \wedge t_{n-1} = t_n$, given that C_3 is satisfiable.

In the sorted case, the satisfiability check is carried out through sorted unification. We can apply the resolution rule as described in the previous section, restricting θ to be a maximally general Σ-unifier of the selected literals. We show a sorted derivation in Section 7.3. In order to retain completeness, a resolvent should be generated for each element of a ΣCIU for the selected literals.[2]

Definition 7.6 (SSC-Clauses) *An* SSC-clause*, written as c/C, is a clause (that is, a multiset of literals) c together with an* SSC *C in solved form such that $M_C(X) \neq \emptyset$ for all $X \in Var(c)$. The set of clauses in $M_C(c)$ are the* ground instances *of c/C. Given a set S of* SSC*-clauses, S_{gr} denotes the set of all the ground instances of clauses in S.*

The constraint on an SSC-clause c/C can be interpreted as

$$X_1 \in M_C(X_1) \wedge \ldots \wedge X_n \in M_C(X_n), \text{ where } \{X_1, \ldots, X_n\} = Var(c).$$

By definition, this constraint is always satisfiable, since we require that $M_C(X_i) \neq \emptyset$ for all i. Alternatively, an SSC-clause c/C can be seen as schematic for the set of its ground instances, $M_C(c)$.

[2] If the ΣCIU is infinite then the resolvents will have to be enumerated in some "fair" manner.

Given two SSC-clauses, c_1/C_1 and c_2/C_2, we assume that $C = C_1 \cup C_2$ is also an SSC in solved form, equivalent to C_A and C_B over their respective domains. This can always be satisfied by renaming the variables in one of the constrained clauses.

Recall that an SSC-*unifier* for a set of terms $\{t_1, \ldots, t_n\}/C$ given a solved form SSC C is a pair $\langle \theta_i, C_i \rangle$ output by the unification algorithm of Figure 4.2 when solving the equation set $\{t_1 = t_2, \ldots, t_{n-1} = t_n\}/C$. Resolution for SSC-clauses can now be described as follows:

> SSC-**Resolution**: From SSC-clauses $c1 = (A \cup S_1)/C$ and $c2 = (S_2 \cup B)/C$, derive the SSC-clause $c3 = (A \cup B)\theta_i/C_i$, where $\langle \theta_i, C_i \rangle$ is an SSC-unifier of $(S_1 \cup \overline{S_2})/C$. The clause $c3$ is called an SSC-*resolvent* of $c1$ and $c2$.

To achieve completeness, all of the clauses $\{(A \cup B)\theta_1/C_1, \ \ldots, \ (A \cup B)\theta_n/C_n\}$ should be derived from $c1$ and $c2$ above, where $\{\langle \theta_1, C_1 \rangle, \ \ldots, \ \langle \theta_n, C_n \rangle\}$ is the set of SSC-unifiers for $(S_1 \cup \overline{S_2})/C$ as generated by the unification algorithm of Figure 4.2. Note that by the results in Chapter 4, such a set is always finite. As with ordinary resolution, the variables in the two clauses should be renamed apart, if necessary, so that $(A \cup S_1)$ and $(S_2 \cup B)$ are variable-disjoint.

Definition 7.7 (SSC-Derivations, $\vdash_{\overline{\mathrm{SSC}}}$) *An* SSC-resolution derivation *of an* SSC-*clause* c *from a set of* SSC-*clauses* S *is a finite binary tree whose root is* c*, whose leaves are* SSC-*clauses in* S*, and where all internal nodes are* SSC-*resolvents of their children. We write* $S \vdash_{\overline{\mathrm{SSC}}} c$ *if there is an* SSC-*resolution derivation of* c *from* S.

We now prove the soundness of SSC-resolution, by showing that the ground instances of an SSC-resolvent can be obtained by resolving ground instances of the two constrained clauses being resolved:

Lemma 7.8 (SSC-Resolution Soundness Lemma) *Let* c_1 *and* c_2 *be* SSC-*clauses and let* $c3$ *be an* SSC-*resolvent of* $c1$ *and* $c2$. *Then any ground instance of* $c3$ *can be obtained by resolving a ground instance of* $c1$ *and a ground instance of* $c2$.

Proof: Let $c1 = (A \cup S_1)/C$, $c2 = (S_2 \cup B)/C$, and $c3 = (A \cup B)\theta_i/C_i$, where $\langle \theta_i, C_i \rangle$ is an SSC-unifier of $S_1 \cup \overline{S_2}$. Let $c3'$ be a ground instance of $c3$; then there exists a ground substitution μ such that $c3' = (A \cup B)\theta_i\mu$ and $X\mu \in M_{C_i}(X)$ for all $X \in Var(A \cup B)\theta_i$. Let $\rho = \theta_i \cdot \mu$. Since $\langle \theta_i, C_i \rangle$ is an SSC-unifier of $(S_1 \cup \overline{S_2})/C$ and $X\rho \in M_{C_i}(X\theta_i)$ for all $X \in Var(S_1 \cup \overline{S_2})$, then ρ unifies $(S_1 \cup \overline{S_2})/C$, by the correctness of the SSC unification algorithm (Theorem 4.17). Since we assume that $M_C(X) \neq \emptyset$ for all $X \in Var(A \cup S_1 \cup S_2 \cup B)$, then ρ can be extended so that $c1' = (A \cup S_1)\rho \in M_C(A \cup S_1)$ and $c2' = (S_2 \cup B)\rho \in M_C(S_2 \cup B)$. Since ρ unifies $S_1 \cup \overline{S_2}$, then $c3' = (A \cup B)\rho$ is a ground resolvent of $c1'$ and $c2'$, which are ground instances of $c1$ and $c2$ respectively (note that clauses are multisets). ■

The converse of the above lemma is the basis of the proof for the Lifting Lemma for SSC-resolution, as follows:

Lemma 7.9 (SSC Lifting Lemma) *If* S *is a set of* SSC-*clauses and* g *is a ground clause such that* $S_{gr} \vdash_{\overline{\mathrm{RES}}} g$*, then there is an* SSC-*clause* c/C' *such that* $S \vdash_{\overline{\mathrm{SSC}}} c/C'$ *and* g *is a ground instance of* c/C'.

Proof: As in the unconstrained case, it suffices to show that a single resolution step in a derivation "lifts:" Let $c1/C$ and $c2/C$ be SSC-clauses such that $c1$ and $c2$ are variable-disjoint.

We now show that if $c1/C$ and $c2/C$ have ground instances $c1'$ and $c2'$ that resolve to obtain $c3'$, then there is an SSC-resolvent $c3/C'$ of $c1/C$ and $c2/C$ such that $c3'$ is a ground instance of $c3/C'$:

Let $c1 = (A \cup S_1)$, $c2 = (S_2 \cup B)$. Since $c1$ and $c2$ are variable disjoint, then there is a ground substitution ρ such that $c1' = c_1\rho$, $c2' = c2\rho$, $c3' = (A \cup B)\rho$, and $X\rho \in M_C(X)$ for all $X \in Var(c1 \cup c2)$. But then ρ unifies the set $S_1 \cup \overline{S}$ and $X\rho \in M_C(X)$ for all $X \in Var(c1 \cup c2)$. Therefore, by the completeness of the SSC unification algorithm (Theorem 4.17) there exists an SSC-unifier $\langle \theta_i, C_i \rangle$ for $(S_1 \cup \overline{S_2})/C$ such that $X\rho \in M_{C_i}(X\theta_i)$ for all X in $Var(S_1 \cup S_2)$. Also, $X\theta = X$ if $X \notin Var(S_1 \cup \overline{S_2})$. Thus, $c3 = (A \cup B)\theta_i/C_i$ is an SSC-resolvent of $c1/C$ and $c2/C$, and $c3'$ is a ground instance of $c3$. ■

Therefore, we have:

Theorem 7.10 (SSC-Resolution Theorem) *Let S be a set of* SSC*-clauses. Then $S \vdash_{\overline{\mathrm{SSC}}} \Box$ if, and only if, $S_{gr} \vdash_{\overline{\mathrm{RES}}} \Box$.*

The "if" direction follows from the soundness lemma for SSC-resolution (Lemma 7.8), given that clauses are multisets. For any SSC-derivation of $\Box$ from S there will be a corresponding ground derivation.[3] The "only if" direction follows directly from the lifting lemma for SSC-resolution (Lemma 7.9).

The above theorem does not mention the satisfiability of a set of SSC-clauses. We can easily relate SSC-derivations with the semantics of sorted clauses by considering SSC's that represent the least models of sort theories. To do this we need the following theorem, adapted from [Frisch, 1991],[4] which is a version of the Herbrand Theorem as applied to sorted axiomatizations:

Theorem 7.11 (Sorted Herbrand Theorem) *Let Σ be a sort theory that has a least Herbrand model and let S be a set of sorted clauses. Then $S \cup \Sigma$ is unsatisfiable if, and only if, the set of all the Σ-ground instances of clauses in S ($S_{\Sigma,gr}$) is unsatisfiable.*

Let Σ be a sort theory and let C_Σ be an SSC that represents the least model of Σ. Recall from Theorem 6.6 that if t is any sorted term and t/C_t is the SSC-relativization of t (see Definition 6.5), then $[t/(C_t \cup C_\Sigma)]$ is the set of all Σ-ground instances of t. Therefore, the set of all the Σ-ground instances of the clauses in S, $S_{\Sigma,gr}$, is equal to $(S/(C_S \cup C_\Sigma))_{gr}$, where S/C_S is the SSC-relativization of S. We can now prove the last theorem in this thesis:

Theorem 7.12 *Let Σ be a sort theory and let C_Σ represent its least Herbrand model. Let S be a finite set of sorted clauses and let S/C_S be its* SSC*-relativization. Then $S \cup \Sigma$ is unsatisfiable if, and only if, $S/(C_S \cup C_\Sigma) \vdash_{\overline{\mathrm{SSC}}} \Box$.*

Proof: By the sorted Herbrand theorem, $S \cup \Sigma$ is unsatisfiable iff $S_{\Sigma,gr}$ is unsatisfiable. By the soundness and completeness of ground resolution, $S_{\Sigma,gr} \models \Box$ iff $S_{\Sigma,gr} \vdash_{\overline{\mathrm{RES}}} \Box$. Since C_Σ represents the least Herbrand model of Σ, $S_{\Sigma,gr} = (S/(C_S \cup C_\Sigma))_{gr}$. By the lifting lemma for SSC-resolution (Lemma 7.9) and its converse (Lemma 7.8), $S_{\Sigma,gr} \vdash_{\overline{\mathrm{RES}}} \Box$ iff $S/(C_S \cup C_\Sigma) \vdash_{\overline{\mathrm{SSC}}} \Box$. ■

[3] A formal proof uses structural induction on the size of derivations.

[4] This is a weak version of a stronger result proved in [Frisch, 1991].

$$\begin{aligned}\Sigma = \{\ & \forall x\ \mathrm{T1}(x) \rightarrow \mathrm{T1}(f(x)),\\ & \forall x\ \mathrm{T2}(x) \rightarrow \mathrm{T2}(f(x)),\\ & \forall x\ \mathrm{T3}(x) \rightarrow \mathrm{T1}(x) \wedge \mathrm{T2}(x),\\ & \mathrm{T3}(a)\ \}\end{aligned}$$

(1):	$\neg P(x{:}\mathrm{T1}) \vee Q(x{:}\mathrm{T1})$	input
(2):	$P(y{:}\mathrm{T2})$	input
(3):	$\neg Q(f(z{:}\mathrm{T1})) \vee Ans(z{:}\mathrm{T1})$	input
(4):	$Q(u{:}\mathrm{T3})$	from (1) and (2)
(5):	$Q(f(w{:}\mathrm{T3}))$	from (1) and (2)
(6):	$Ans(v{:}\mathrm{T3})$	from (3) and (5)

Figure 7.1: A sorted resolution derivation

7.3 A Simple Derivation Example

Frisch [1991] presents the sorted resolution derivation shown in Figure 7.1. We have added the answer literal Ans to keep track of the bindings and obtain answers when clause (3) is seen as the query $Q(f(z{:}\mathrm{T1}))$.

Note that clauses (4) and (5) are only two of an infinite number of resolvents of clauses (1) and (2); Σ tells us that if x is of sort T3, it is also of sorts T1 and T2. But then $f(x)$ also has sorts T1 and T2, and thus $f(f(x))$ also does, and so on. Therefore, a ΣCIU for $\{P(x{:}\mathrm{T1}), P(y{:}\mathrm{T2})\}$ is:

$$\left\{ \begin{array}{l} \{x{:}t1 \mapsto z{:}t3,\ y{:}t2 \mapsto z{:}t3\}\,, \\ \{x{:}t1 \mapsto f(z{:}t3),\ y{:}t2 \mapsto f(z{:}t3)\}\,, \ldots \end{array} \right\}$$

The set of answer clauses that can be obtained is thus infinite:

$$\{Ans(z{:}\mathrm{T3}),\ Ans(f(z{:}\mathrm{T3})),\ \ldots,\ Ans(f^n(z{:}\mathrm{T3})),\ \ldots\}\,.$$

Figure 7.2 an SSC-resolution derivation that can be thought to subsume that in Figure 7.1. The solved form SSC C_Σ represents Σ, and clauses $(1')$, $(2')$ and $(3')$ are the the SSC-relativization of the initial set of clauses, constrained by C_Σ.

Consider now applying SSC-resolution to clauses $(1')$-$(3')$.

Resolving $(1')$ and $(2')$, we unify

$$\{P(X), P(Y)\}\,/\,\{\langle X \equiv \Upsilon_1\rangle,\ \langle Y \equiv \Upsilon_2\rangle\} \cup C_\Sigma$$

Step 1: Renaming variables with $\pi = \{X \mapsto M,\ Y \mapsto N\}$, we solve the set of equations and membership constraints

$$\{P(M) = P(N),\ M \underline{\in} X,\ N \underline{\in} Y\}\,,$$

$$C_\Sigma = \left\{ \begin{array}{c} \langle \Upsilon_1 \equiv \Upsilon_3 \sqcup f(\Upsilon_1) \sqcup k_1 \rangle \\ \langle \Upsilon_2 \equiv \Upsilon_3 \sqcup f(\Upsilon_2) \sqcup k_2 \rangle \\ \langle \Upsilon_3 \equiv a \rangle \end{array} \right\}$$

$(1')$:	$\neg P(X) \vee Q(X)$	$/\ \{\langle X \equiv \Upsilon_1 \rangle\} \cup C_\Sigma$	input
$(2')$:	$P(Y)$	$/\ \{\langle Y \equiv \Upsilon_2 \rangle\} \cup C_\Sigma$	input
$(3')$:	$\neg Q(f(Z)) \vee A(Z)$	$/\ \{\langle Z \equiv \Upsilon_1 \rangle\} \cup C_\Sigma$	input
$(4')$:	$Q(N)$	$/\ \{\langle N \equiv \Upsilon_3 \sqcup f(N) \rangle\} \cup C_\Sigma$	from $(1')$ and $(2')$
$(5')$	$Ans(W)$	$/\ \{\langle N \equiv \Upsilon_3 \sqcup f(N) \rangle$, $\langle W \equiv \Upsilon_3 \sqcup f(Z) \rangle$, $\langle Z \equiv \Upsilon_3 \sqcup f(Z) \rangle\ \} \cup C_\Sigma$	from $(3')$ and $(4')$

Figure 7.2: An SSC-resolution derivation

whose normal form is

$$\{M = N,\ N \in X,\ N \in Y\}\,.$$

Thus we obtain a single pair $\langle \theta_1, C_1 \rangle$, where $\theta_1 = \{M \mapsto N\}$ and $C_1 = \{\langle N \equiv \Upsilon_1 \sqcap \Upsilon_2 \rangle\} \cup C_\Sigma$. Note that this is not yet an SSC-unifier, since C_1 is not in solved form.

Step 2: The SSC C_1 is rewritten into solved form. $\langle N \equiv \Upsilon_1 \sqcap \Upsilon_2 \rangle$ is rewritten into $\langle N \equiv \Upsilon_3 \sqcup (f(\Upsilon_1) \sqcap f(\Upsilon_2)) \rangle$. Using a slightly optimized version of the simplification algorithm of Chapter 3, we could have initialized Int to map N into $\{\Upsilon_1, \Upsilon_2\}$ at the start of the algorithm; $f(\Upsilon_1) \sqcap f(\Upsilon_2)$ is then transformed into $f(N)$, so we finally obtain the solved form SSC $C_1' = \{\langle N \equiv \Upsilon_3 \sqcup f(N) \rangle\} \cup C_\Sigma$.

Step 3: After composing π and θ, the SSC-unifier is, finally, $\langle \{X \mapsto N,\ Y \mapsto N\}\,,\ C_1' \rangle$.

Thus, the new clause is:

$$(4'):\ \ Q(N) \qquad /\, \{\langle N \equiv \Upsilon_3 \sqcup f(N) \rangle\} \cup C_\Sigma \qquad \text{from } (1') \text{ and } (2')$$

There is only one SSC-unifier, and thus a single resolvent; clause $(4')$ stands for all possible "ordinary" resolvents, including clauses (4) and (5) above. We now resolve clauses $(3')$ and $(4')$, unifying $\{Q(f(Z)),\ Q(N)\}$ given the SSC

$$\{\langle z \equiv \Upsilon_1 \rangle,\ \langle N \equiv \Upsilon_3 \sqcup f(N) \rangle\} \cup C_\Sigma$$

Step 1: $\pi = \{N \mapsto V,\ Z \mapsto W\}$. The initial set of equations and membership constraints is

$$\{f(W) = V,\ W \in Z,\ V \in N\}\,.$$

The solving process is:

$$\begin{array}{ll}
\{f(W) = V,\ W \Subset Z,\ V \Subset N\} & \Longrightarrow \\
\{V = f(W),\ W \Subset Z,\ V \Subset N\} & \Longrightarrow \\
\{V = f(W),\ W \Subset Z,\ f(W) \Subset N\} & \Longrightarrow \\
\{V = f(W),\ W \Subset Z,\ f(W) \Subset \Upsilon_3 \sqcup f(N)\} & \Longrightarrow \\
\{V = f(W),\ W \Subset Z,\ f(W) \Subset a \sqcup f(N)\} & \Longrightarrow \\
\{V = f(W),\ W \Subset Z,\ W \Subset Z\}\,. &
\end{array}$$

Step 2: We now rewrite the SSC $\{\langle W \equiv N \sqcap Z\rangle \cup C_\Sigma\}$ into solved form:
Let $Int = \emptyset$; $\{W = N \sqcap Z\} \Longrightarrow \{W = (\Upsilon_3 \sqcup f(N)) \sqcap \Upsilon_1\}$; with $Int = \{W, \{N, Z\}\}$; [5] this is then transformed to $\{W = (\Upsilon_3 \sqcap \Upsilon_1) \sqcup (f(N) \sqcap \Upsilon_1)\}$.

$$\begin{array}{ll}
\Longrightarrow & \{W = (\Upsilon_3 \sqcap \Upsilon_3) \sqcup (\Upsilon_3 \sqcap f(\Upsilon_1)) \sqcup (f(N) \sqcap f(\Upsilon_1))\} \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup (a \sqcap f(\Upsilon_1)) \sqcup f(Z),\ Z = N \sqcap \Upsilon_1\}\,, \\
 & \text{where } Int = \{\langle X, \{N, Z\}\rangle,\ \langle Z, \{N, \Upsilon_1\}\rangle\}\,. \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup \bot \sqcup f(Z),\ Z = (\Upsilon_3 \sqcup f(N)) \sqcap (\Upsilon_3 \sqcup f(\Upsilon_1))\} \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup f(Z),\ Z = (\Upsilon_3 \sqcap \Upsilon_3) \sqcup (\Upsilon_3 \sqcap f(\Upsilon_1)) \sqcup (f(N) \sqcap \Upsilon_3) \sqcup (f(N) \sqcap f(\Upsilon_1))\} \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup f(Z),\ Z = \Upsilon_3 \sqcup (a \sqcap f(\Upsilon_1)) \sqcup (f(N) \sqcap a) \sqcup f(Z)\}\,. \\
 & \text{Note that } f(N) \sqcap f(\Upsilon_1) \text{ was replaced by } Z, \text{ since } Int = \{\langle Z, \{N, \Upsilon_1\}\rangle\}\,. \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup f(Z),\ Z = \Upsilon_3 \sqcup \bot \sqcup \bot \sqcup f(Z)\} \\
\Longrightarrow & \{W = \Upsilon_3 \sqcup f(Z),\ Z = \Upsilon_3 \sqcup f(Z)\}\,.
\end{array}$$

The solving process can be made more efficient by detecting that W itself is equal to the intersection of N and Υ_1. In this case, we would obtain the SSC $\{W = \Upsilon_3 \sqcup f(W)\} \cup C_\Sigma$, equivalent to the one above over the original domain.

The SSC-unifier is $\langle\{N \mapsto f(W), Z \mapsto W, \}, C'\rangle$, where

$$C' = \{\langle N \equiv \Upsilon_3 \sqcup f(N)\rangle\} \cup \{\langle W \equiv \Upsilon_3 \sqcup f(Z)\rangle,\ \langle Z \equiv \Upsilon_3 \sqcup f(Z)\rangle\} \cup C_\Sigma.$$

The answer clause is now:

$(5')$ $Ans(W)$ $\quad / \{\langle N \equiv \Upsilon_3 \sqcup f(N)\rangle, \langle W \equiv \Upsilon_3 \sqcup f(Z)\rangle,\ \langle Z \equiv \Upsilon_3 \sqcup f(Z)\rangle\} \cup C_\Sigma$
from $(3')$ and $(4')$

One can read the answer as being the set of terms of the form $f^n(\Upsilon_3)$, or, in the least Herbrand model of Σ, as the set $\{a,\ f(a),\ f(f(a)),\ \ldots\}$. Note that this infinite set is represented *finitely*.

In general, the SSC-unification algorithm will always generate a finite set of SSC-unifiers, as opposed to traditional sorted unification algorithms.

[5] We will only show Int when it is changed. Also, *Rep* will be exactly the "inverse" of Int in this case, since we never have to compute the intersection of the same two variables more than once.

Chapter 8

Conclusions

We can now review the main results presented in this thesis.

Chapter 2 presented SSC's, an in particular *shallow* ones, as an extension of ordinary set constraints to include inter-variable dependencies, resulting in a formalism that describes a class of tree languages strictly more general than the regular ones. Chapter 3 presents a simplification algorithm that rewrites shallow SSC's with no cycles into solved form SSC's. Chapter 4 presents a solving algorithm that rewrites systems of equations and membership constraints into a normal form, where the membership constraints are described by shallow SSC's. This procedure and the SSC simplification algorithm are then used to construct a unification algorithm for sets of equations constrained by shallow SSC's.

The remaining chapters relate the above results to the problem of sorted unification: Chapter 5 reviews two of the most general presentations of sorted logic, sort theories and sorted signatures, and shows that sorted unification under a sorted signature is equivalent to sorted unification under a corresponding sort theory. Chapter 6 describes how certain classes of logic programs, sort theories and sorted signatures can be represented using shallow SSC's, presenting an algorithm that translates a *semi-linear* sorted signature into an equivalent shallow SSC. Thus, sorted unification is shown to be decidable for this general class of sorted signatures. All of these results are used in Chapter 7 to show that integrating the SSC-unification algorithm into a resolution-based deductive system results in a sound and complete hybrid system where shallow SSC's are used to represent sortal constraints.

8.1 Open Problems

This thesis leaves many questions unanswered. First of all, Conjecture 6.11, which implies that SSC-unification can be used to extract most general answers (as defined in the normal sorted framework) remains to be proved. In fact, it might have to be partially reformulated, since it is unclear what the best definition of generality between SSC-substitutions should be, particularly when SSC's are used to represent sorts: Definition 6.9 can be used as a starting point, but this exact formulation might not be the most appropriate one.

Many of the other open questions are of a practical nature. Cohn [1989b, page 95] writes:

> There is a trade-off between the expressiveness of the allowable sortal descriptions and the complexity of the machinery which is required to implement it ... Moreover, although the dynamic sort checking overheads may increase as the expressiveness

of the many-sorted[1] logic increases, the search space may be reduced, so finding an optimal point in the space is not trivial, and will of course depend on the nature of the domain being axiomatized and whether the increased expressiveness can be exploited in the domain.

How can the SSC-unification algorithm be most efficiently implemented? Is it possible to achieve polynomial-time unification without sacrificing too much of the expressive power of SSC's? The corresponding theoretical question is to identify what restrictions ensure that SSC-unification is in $\mathcal{P}$ or $\mathcal{NP}$. As a particular instance of this, we can ask if there are any particular efficient implementations in the case of linear SSC's (which correspond to linear sorted signatures, which in turn are equivalent to regular tree grammars).

Some other questions are:

- One of the most attractive features of SSC's is the ability to represent infinite sets of terms structured in a relatively "non-trivial" way. What problem domains would most benefit by the introduction of SSC-unification?

- The *unification over the weakest range* presented by Irani and Shin [1985] can be thought of as an instance of sorted unification using SSC's in the case of monomorphic sorted signatures, where new sorts that correspond to boolean combinations of previous ones are added to the sort hierarchy during the unification process (see Section 6.4). The new variables that are added to the SSC's by the unification procedure are equivalent to new sorts. Schmidt-Schauß [1989] writes that this approach to sorted unification "*...may be too heavy a machinery for most practical purposes...*" [1989, page 5]. This criticism might be partially true of SSC-unification; however, we have shown that our algorithm can solve sorted unification problems that standard sorted unification procedures cannot. SSC's in solved form are equivalent to term declarations; we use them dynamically, adding new sorts as necessary. Would it be possible, in some restricted cases, to use the algorithms in this thesis to pre-process a sort theory, computing all (or most) of the necessary sorts ahead of time? Note that in general an infinite number of non-equivalent variables can be added to the constraints on SSC-clauses as an SSC-derivation proceeds.

- If we take the view of sorted logic as imposing control on a corresponding unsorted deduction, as in [Cohn, 1986], what is the relationship between a proof carried out using sorted unification under SSC's and the corresponding unsorted proof? What control and search strategies would achieve the same effect?

  There's also the question of how complex should the constraint-handling mechanism and the sort theory be, in order to achieve the best overall efficiency.

- The set of SSC-unifiers output by the SSC-unification algorithm of Figure 4.2 is not guaranteed to be minimal. Doing so is probably of considerable practical importance. The work of Bogaert and Tison [1991] shows that the problem of subsumption between terms constrained by shallow SSC's is decidable. Briefly, the reason for this is that the tree languages describable by shallow SSC's are embedded in a larger class of tree languages, closed under complementation, for which the emptyness problem is decidable. These results could be used to obtain a procedure that identifies redundant SSC-unifiers. Again,

[1] The same as order-sorted.

the question of an efficient implementation of such a procedure is left open. In general, it appears that there is no direct method for obtaining sets of incomparable well-sorted unifiers; all known general sorted unification procedures that obtain ΣCIU's for complex signatures and theories (as those in [Schmidt-Schauß, 1989] and [Cohn and Frisch, 1990]) do so by "filtering out" redundant unifiers at the end, explicitly comparing the elements of a set of Σ-unifiers that is guaranteed to be complete, but not minimal.

8.1.1 Possible Extensions

We close this thesis with some comments of a rather speculative nature concerning possible ways in which the SSC representation of membership constraints might be extended:

- Whether SSC's could be extended to include inequality constraints while still retaining the set expression formalism is unclear; if this were the case, though, the resulting algorithms should correspond to the decision procedures for tree automata with equality tests as presented by Bogaert and Tison [1991].

- Similar questions are: How would SSC-unification be extended to take into account restricted classes of background equational theories, while preserving the decidability properties necessary to carry out unification? How can SSC's be extended to allow infinite terms (along the lines of Colmerauer's Prolog II [1982; 1984])?[2]

- There is always the question of whether there is a still more general class of sorted signatures for which unification is decidable. We have seen how the work of Comon and Delor [1991] and of Cohn and Frisch [1990] shows that this question is analogous to answering questions about tree languages and logic programs, respectively. Semi-linearity seems to be one of the most general "reasonable" such restrictions, as far as sorted signatures themselves are concerned. In general, if we allow function declarations of the form $S_0(f(x{:}S_1, g(x{:}S_1)))$, sorted unification becomes undecidable, as shown in [Schmidt-Schauß, 1989]. Similarly, Bogaert and Tison [1991] point out that the emptyness problem for tree automata with equality tests becomes undecidable if dependencies between variables nested at depth 2 are allowed in the state transition rules of the automata.

- What classes of logic programs have least models that can be exactly represented using shallow SSC's? Can Heintze and Jaffar's [1990] program approximation algorithm be made more accurate by using a set-constraint representation that includes inter-variable dependencies? This would extend the class of unsorted axiomatizations from which we could extract a decidable sort theory, as in the "inverse relativization" discussed in Section 6.3.4. In general, such an algorithm would have many potential applications in program analysis, as well as type inference for programming languages.

- The SSC-unification algorithm solves membership constraints on terms by reducing them to constraints on variables, and then testing the satisfiability of those. This process might not be strictly necessary, in practice, if the definitions of "solved form" unification problems are changed. Prehofer [1992] presents a method for carrying out efficient sortal

[2] We would then be able to represent structured infinite sets containing infinite terms.

deduction for regular, polymorphic signatures, by associating sortal constraints with general terms in addition to the usual constraints on variables. Whether this idea can be extended to semi-linear signatures, possibly in conjunction with the algorithms in this thesis, is also left open.

- There are other approaches to sorts and unification that raise issues not considered in this thesis: Cohn [1983; 1989a] has developed a deductive system that uses a "more expressive" sorted logic, in the sense that the sort theory is able to express *and use* more than the usual "subset and membership" taxonomic information. Among other features, the sort predicates can appear in the main logic as *characteristic literals*, which allows using disjointedness information about sorts in a proof; also, truth values can be used as sorts, so that literals can be evaluated according to the sorts of the predicate arguments. Achieving all of this requires new special-purpose inference rules, since not everything can be handled only by a specialized unifier, and so this system cannot be classified as "substitutional." However, it might still be possible to integrate many of these ideas with the SSC representation and unification algorithm.

Bibliography

[Abrams and Frisch, 1989] Charlene Bloch Abrams and Alan M. Frisch. An examination of the efficiency of sorted deduction. Unpublished manuscript, 1989.

[Aiken and Murphy, 1991] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, January 1991.

[Bogaert and Tison, 1991] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. 1991. To appear in *Symposium on Theoretical Aspects of Computer Science*, 1992.

[Bogaert, 1990] Bruno Bogaert. *Automates d'arbres avec tests d'égalités*. Ph.D. Dissertation, Université des Sciences et Techniques de Lille Flanders-Artois, December 1990.

[Burckert, 1990] Hans-Jürgen Burckert. A resolution principle for clauses with constraints. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 178–192. Springer-Verlag, July 1990. Volume 449 of *Lecture Notes in Artificial Intelligence*.

[Chang and Lee, 1973] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[Chen and Hsiang, 1991a] Hong Chen and Jieh Hsiang. Order-sorted equational specification and completion. Technical report, State University of New York at Stony Brook, 1991.

[Chen and Hsiang, 1991b] Hong Chen and Jieh Hsiang. Recurrence domains: Their unification and application to logic programming. Technical report, State University of New York at Stony Brook, July 1991. Short version appears as *Logic Programming with Recurrence Domains*, Proceedings ICALP 91.

[Chomicki and Imielinski, 1989] Jan Chomicki and Tomasz Imielinski. Relational specification of infinite query answers. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 174–183, Portland, Oregon, June 1989.

[Cohen, 1990] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Cohn and Frisch, 1990] Anthony G. Cohn and Alan M. Frisch. A universal algorithm for sorted unification. Unpublished manuscript, July 1990.

[Cohn and Frisch, 1991] Anthony G. Cohn and Alan M. Frisch. Computing sorted unifiers. Unpublished manuscript, November 1991.

[Cohn, 1983] Anthony G. Cohn. *Mechanising a Particularly Expressive Many-Sorted Logic*. Ph.D. Dissertation, Department of Computer Science, University of Essex, 1983.

[Cohn, 1985] Anthony G. Cohn. On the solution of Schubert's Steamroller in many-sorted logic. In *Proceedings of the 9^{th} International Joint Conference on Artificial Intelligence*, pages 1169–1174, August 1985.

[Cohn, 1986] Anthony G. Cohn. Many-sorted logic = unsorted logic + control? In M. Bramer, editor, *Research and Development in Expert Systems III*, pages 184–194. Cambridge University Press, Cambridge, England, 1986.

[Cohn, 1989a] Anthony G. Cohn. On the appearance of sortal literals: A non substitutional framework for hybrid reasoning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 55–66, May 1989.

[Cohn, 1989b] Anthony G. Cohn. Taxonomic reasoning with many-sorted logics. *Artificial Intelligence Review*, 3:89–128, 1989.

[Colmerauer, 1982] Alain Colmerauer. Prolog and infinite trees. In K. A. Clark and S.A. Tarnlund, editors, *Logic Programming*. Academic Press, London, 1982.

[Colmerauer, 1984] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems, 1984*, pages 85–99, 1984.

[Colmerauer, 1990] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[Comon and Delor, 1991] Hubert Comon and Catherine Delor. Equational formulae with membership constraints. Rapport de Recherche 649, LRI, Université de Paris Sud, Orsay, France, March 1991. Revised version of *Equational Formulas in Order-Sorted Algebras*, proc. ICALP 90.

[Comon, 1989] Hubert Comon. Inductive proofs by specifications transformation. In *Proceedings Rewriting Techniques and Applications*, pages 76–91, Chapel Hill, April 1989. Springer-Verlag. Volume 355 of LNCS.

[Comon, 1990] Hubert Comon. Equational formulas in order-sorted algebras. In *Proceedings of the 17^{th} International Colloquium on Automata, Languages and Programming*, pages 674–688, July 1990.

[Comon, 1991] Hubert Comon. Completion of rewrite systems with membership constraints. Rapport de Recherche 699, LRI, Université de Paris Sud, Orsay, France, September 1991.

[Dershowitz and Manna, 1979] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[Eder, 1985] Elmar Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, 1985.

[Frisch and Cohn, 1991] Alan M. Frisch and Anthony G. Cohn. Thoughts and afterthoughts on the 1988 workshop on principles of hybrid reasoning. *AI Magazine*, 11(5):77–83, January 1991.

[Frisch and Page, 1990] Alan M. Frisch and C. David Page. Generalization with taxonomic information. In *Proceedings of the* 8^{th} *National Conference on Artificial Intelligence*, pages 755–761, July 1990.

[Frisch, 1986a] Alan M. Frisch. *Knowledge Retrieval as Specialized Inference.* Ph.D. Dissertation, Computer Science Department, University of Rochester, August 1986. Also appears as Report No. UIUCDCS-R-88-1404, University of Illinois at Urbana-Champaign, February 1988.

[Frisch, 1986b] Alan M. Frisch. Parsing with restricted quantification: An initial demonstration. *Computational Intelligence*, 2(3):142–150, 1986. Also appears in A. G. Cohn and J. R. Thomas, editors, *Artificial Intelligence and its Applications*, pages 5–22. Wiley, Chichester, U.K., 1986.

[Frisch, 1989] Alan M. Frisch. A general framework for sorted deduction: Fundamental results on hybrid reasoning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 126–136, May 1989.

[Frisch, 1991] Alan M. Frisch. The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. *Artificial Intelligence*, 49:161–198, 1991.

[Gallier and Snyder, 1990] Jean Gallier and Wayne Snyder. Designing unification procedures using transformations: A survey. Technical Report BU-CS TR 90-005, Boston University, September 1990.

[Gècseg and Steinby, 1984] Ferenc Gècseg and Magnus Steinby. *Tree Automata.* Akadèmiai Kiadò, Budapest, Hungary, 1984.

[Heintze and Jaffar, 1990] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. Research Report RC 16089 (#71415), IBM T.J. Watson Research Center, September 1990. A preliminary version appears in *Proceedings ACM Symposium on Principles of Programming Languages*, 1990.

[Heintze and Jaffar, 1991] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of herbrand set constraints. Technical Report CMU-CS-91-110, Carnegie Mellon University, February 1991. Abstract appears in *Proceedings of the* 5^{th} *Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Herbrand, 1930] Jacques Herbrand. Recherches sur la théorie de la démonstration. 1930. Appears as *Investigations in Proof Theory*, in W. Golfarb, editor, *Logical Writings*, Harvard University Press, 1971.

[Höhfeld and Smolka, 1988] Markus Höhfeld and G. Smolka. Definite relations over constraint languages. Lilog-Report 53, IBM Deutschland, October 1988.

[Irani and Shin, 1985] K. B. Irani and D. G. Shin. A many-sorted resolution based on an extension of a first-order language. In *Proceedings of the 9^{th} International Joint Conference on Artificial Intelligence*, pages 1175–1177, Los Angeles, California, August 1985.

[Jaffar and Lassez, 1987] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, pages 111–119, January 1987.

[Jaffar *et al.*, 1990] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\Re$) language and system. Research Report RC 16292 (#72336), IBM T.J. Watson Research Center, November 1990.

[Jouannaud and Kirchner, 1990] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. Research Report 561, Université de Paris Sud - Centre d'Orsay – Laboratoire de Recherche en Informatique, March 1990.

[Knight, 1989] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[Lassez *et al.*, 1988] J-L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In Jack Minker, editor, *Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.

[Martelli and Montanari, 1982] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[McAllester, 1983] David A. McAllester. Solving uninterpreted equations with context free expression grammars. Technical Report AI Memo No. 708, M.I.T., May 1983.

[Mishra and Reddy, 1985] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, New Orleans, 1985.

[Mitchell *et al.*, 1991] Michael K. Mitchell, Tomás E. Uribe, and Alan M. Frisch. *A Framework for Resolution-based Automated Proof Procedure Systems –* FRAPPS *2.0 User's Guide.* University of Illinois at Urbana-Champaign, Urbana, Illinois, September 1991.

[Mitchell, 1989] Michael K. Mitchell. The anatomy of a framework for resolution-based automated proof procedure systems. Master's thesis, University of Illinois at Urbana-Champaign, July 1989.

[Nebel and Smolka, 1990] Bernard Nebel and Gert Smolka. Representation and reasoning with attributive descriptions. In K.H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, pages 112–139. Springer-Verlag, 1990. Volume 418 of *Lecture Notes in Artificial Intelligence*.

[Ohlbach and Schmidt-Schauß, 1985] Hans J. Ohlbach and Manfred Schmidt-Schauß. The lion and the unicorn. *Journal of Automated Reasoning*, (1):327–332, 1985.

[Prehofer, 1992] Christian F. Prehofer. A constraint language for order-sorted polymorphic resolution. Master's thesis, University of Illinois at Urbana-Champaign, January 1992.

[Reddy and Lakshman, 1991] Uday S. Reddy and T.K. Lakshman. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Proceedings of the International Logic Programming Symposium*, pages 202–220, San Diego, California, October 1991.

[Robinson, 1965] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965. Also in: *Automation of Reasoning — Classical Papers on Computational Logic*, Volume 1. J. Siekmann and G. Wrightson, editors, Springer-Verlag, 1983.

[Schmidt-Schauß, 1989] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer-Verlag, New York, NY, 1989. Volume 395 of *Lecture Notes in Artificial Intelligence*.

[Siekmann, 1989] Jörg H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3):207–274, 1989.

[Siekmann, 1990] Jörg H. Siekmann. An introduction to unification theory. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 369–424. North Holland, 1990.

[Smolka, 1989] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* Ph.D. Dissertation, Universität Kaiserslautern, W. Germany, May 1989.

[Stickel, 1985] Mark E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

[Stickel, 1986] Mark E. Stickel. Schubert's Steamroller problem: Formulations and solutions. *Journal of Automated Reasoning*, 2(1):89–101, 1986.

[Uribe, 1991] Tomás E. Uribe. *Constraints in* FRAPPS. University of Illinois at Urbana-Champaign, Urbana, Illinois, September 1991.

[Van Hentenryk, 1989] Pascal Van Hentenryk. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[Veronis, 1987] Jean Veronis. Discourse consistency and many-sorted logic. In *Proceedings of the* 11^{th} *International Joint Conference on Artificial Intelligence*, pages 633–635, Milan, August 1987.

[Walther, 1985a] Christoph Walther. A mechanical solution of Schubert's Steamroller by many-sorted resolution. *Artificial Intelligence*, 26(2):217–224, 1985.

[Walther, 1985b] Cristoph Walther. A classification of unification problems in many-sorted theories. Technical Report Nr. 10/85, Universität Karlsruhe, Fakultät für Informatik, 1985.

[Walther, 1987] Christoph Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation.* Morgan Kaufmann, Los Altos, CA, 1987.