Semantics of Subset-Logic Languages

by

Devashis Jana

A dissertation

submitted to the Faculty of the Graduate School

of State University of New York at Buffalo

in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

September 1994

To

my parents

and to

our family

Preface

We give here an overview of the various chapters.

Chapter 1 introduces the subset-logic languages through program examples. It also gives the scope of the dissertation and an overview of the research. Chapter 2 is about relevant background and related work.

Chapters 3, 4 and 5 describe the set-theoretic foundations needed to discuss the subset-logic languages. Chapter 3 gives a theory for axiomatising the set constructors used for representing finite sets. Chapter 4 gives algorithms for unification of terms containing set constructors. Chapter 5 establishes the standard or Herbrand structure arising from the set constructors.

Chapter 6 lays out the basic syntax and conventions common to all the subset-logic languages. Chapter 7 gives the declarative and operational semantics of the first level of subset-logic languages, namely the subset-equational language. Chapter 8 enhances this semantics further to deal additionally with database-like programs. Chapter 9 discusses the declarative and operational semantics of the next level of subset-logic languages, namely, the subset-relational language.

Finally, we close with Conclusions and Further Work that discusses the contributions of this work and the work needed beyond the dissertation.

Here, I also acknowledge the help I have received in the course of this dissertation. I thank Prof. Stuart Shapiro and Prof. Kenneth Regan who together with my advisor Prof. Bharat Jayaraman formed my dissertation committee. I thank Prof. Shapiro for his review of the dissertation and his lively input at my dissertation defense. I thank Prof. Regan for his careful, crisp comments on the dissertation and for his ever-friendliness. I thank my advisor Prof. Jayaraman for his intellectual, emotional, and financial support throughout the dissertation. I also thank him for all his encouragements that helped my efforts on the dissertation. I thank Prof. Stott Parker for being the external reader for my dissertation. I also thank Prof. Jeffery Zucker, from whom I learnt much of my semantics.

Contents

Preface . . . iii

1 Introduction . . . 1

1.1 A Broader Basis for Logic Programming . . . 1

1.2 The SuRE Language . . . 2

1.3 Semantics of Subset-Logic Languages . . . 5

1.4 Overview . . . 6

2 Background and Related Work . . . 10

2.1 Background . . . 10

2.2 Related Work . . . 11

3 Set Constructors and Finite Sets . . . 13

3.1 Set Constructors . . . 13

3.2 Disjointness Transformation . . . 15

3.3 The Axioms SetAx . . . 17

3.4 Consistency of SetAx . . . 21

3.5 Relating SetAx to ZF . . . 25

4 Set Constraints and Unification . . . 33

4.1 Constraints and Solved Forms . . . 33

4.2 Terminology of Unification . . . 37

4.3 Unification Algorithm . . . 39

4.4 Correctness of Unification . . . 45

4.5 Properties of SetAx from Unification 47

5 Herbrand $\cong$-Interpretations 50

5.1 Herbrand $\cong$-Structure 50

5.2 Set-Predicate Interpretations 51

5.3 Herbrand $\cong$-Interpretations model SetAx 53

5.4 Ground Terms & Assignments 54

6 The Class of Subset-Logic Languages 57

6.1 Syntactic Elements 57

6.2 Programs as Logical Formulae 59

6.3 The Class of Subset-Logic Languages 60

6.4 The Semantic Framework 61

6.5 Flattening 62

6.6 Collect-All Assumption 65

7 Subset-Equational Language 66

7.1 Syntax 66

7.2 Completion 67

7.3 Declarative Semantics 72

7.4 Matching 76

7.5 Operational Semantics 83

7.6 Fixpoint Semantics 90

7.7 Soundness and Completeness 96

8 Subset-Equational Language: An Enhanced Semantics 105

8.1 Syntax 105

8.2 Completion 106

8.3 Declarative Semantics 109

8.4 Operational Semantics 110

8.5 Soundness and Completeness 114

9 Subset-Relational Language 116

9.1 Syntax 116

9.2 Completion 118

9.3 Declarative Semantics 120

9.4 Unification and Disunification 122

9.5 Operational Semantics 126

9.6 Soundness and Completeness 128

10 Conclusion and Further Work 130

10.1 Contributions of the Dissertation 130

10.2 Further Work 131

References 133

Semantics of Subset-Logic Languages

by

Devashis Jana

Advisor: Bharat Jayaraman

Abstract

This dissertation examines the semantics of a new paradigm of logic programming called *subset-logic programming.* The most novel construct of a subset-logic program is the *subset* assertion, which in conjuction with the more conventional *relational* and *equational* assertions provide a declarative alternative to many common uses of Prolog's extra-logical features: *cut*, *assert*, *retract*, *setof*, and *mode declarations.* These three forms effectively allow to specify function and relation definitions involving (finite) sets as first-class data objects.

Our study of the semantics of subset-logic languages begins by laying the logical foundations for these languages. Because of the need for new set constructors in representating finite sets, we first axiomatize their meaning through suitable axioms, and then use these axioms in a variety of ways: (i) to show that the set constructors indeed behave like finite sets; (ii) to provide a framework for establishing the correctness of set-unification; (iii) to define a Herbrand structure; and (iv) to provide a basis for discussing logical-consequence semantics for the subset-logic languages.

The semantics of subset-logic languages are then studied in two stages: The first stage has only equality and subset program clauses. Here we show that their declarative semantics can be expressed in terms of logical consequence, and their operational semantics in terms of set-matching and a modification of SLD resolution. We then establish the soundness and completeness of the declarative and operational semantics through fixpoint semantics. We further give an enhancement

of this semantics that additionally handles database programs with incomplete information in a sound but incomplete manner. In the second stage, we also include relations, and examine logical-consequence semantics. On the operational side, we use set-unification with resolution.

The contribution of this dissertation is that we have provided rigorous foundations for incorporating finite sets into logic programming. Using this foundation, we have shown that semantics for the initial levels of the subset-logic languages can be given through the logically impeccable means of logical-consequence semantics.

1 Introduction

In this chapter, we introduce the motivations for and the character of the subset-logic languages. We then describe our semantic approach and the scope of this dissertation. Finally, we give a detailed overview of the research, discussing the issues and results accomplished.

1.1 A Broader Basis for Logic Programming

Logic programs are statements expressed in the language of mathematical logic about relations between objects in a given problem situation ([Kow79]). These logical statements have a certain clausal form which are a subset of logic capable of being executed, thereby providing answers to the problem. Logic programming is one of the leading varieties of declarative programming. That is, logic languages have the property of *separation of concerns* — they allow the programmer to describe only *what* is to be computed without being distracted by *how* it is to be computed. Their programs therefore can reflect a purely mathematical and logical understanding of the problem. Programs are usually specifications themselves, thereby facilitating rapid prototyping and increased programmer productivity. The inherent abstract nature of declarative programs makes them naturally amenable to formal analysis and manipulation such as program proving and program transformation. Their mathematical nature and the absence of control specification facilitates parallel evaluation. Logic programs also have other features that possess the advantages of simplicity and expressivity. However, all these benefits have not yet been realised in practice, and one of the aims of current research in logic languages is to bring us closer to achieving these ideals.

The most well-known example of logic languages is Prolog, which, in the interests of practicality and efficiency, includes *extra-logical* features that cause it to depart from pure declarative programming. Extra-logical features — and we include meta-logical and second-order constructs under this category — are those constructs that have no clear logical meaning and are usually understood in terms of the execution model. The presence of these features in programs makes it hard to reason about them, and difficult to run them in parallel. Also, they become less amenable to formal analysis. In order to overcome many uses of the extra-logical *cut*, *assert*, *retract* and *setof* features of Prolog, as well as to give a declarative treatment of *mode* annotations, a broader basis for logic programming incorporating *equational* and *subset assertions* has been suggested in [Jay90]. A substantial portion of most Prolog programs is deterministic; hence these portions can be formulated more clearly using equations without the cut to specify determinacy. Equations are also preferable when relations can be meaningfully run in only a certain mode — as a function from certain input arguments to certain output arguments, e.g., sorting a list of numbers or summing the leaves of a tree. Many uses of assert and retract correspond to implementations of transitive closure operations, memoization, or collecting results from alternative search paths, as in the *setof* construct. Subset assertions allow a more declarative formulation of such uses.

1.2 The SuRE Language

In this dissertation, we investigate the integration of the three logical forms — equations, relations, and subset assertions — by defining a class of *subset-logic languages*, progressively increasing in expressivity, with each language being designed to capture some additional significant aspect of the integration. At the first level, we study the interaction of equations and subset assertions alone, and we call this *subset-equational programming.* At the next level, we also include relations, and we call it *subset-relational programming.* At further levels we consider other aspects such as negative relations and infinite objects (but not in this dissertation). The results arising from these investigations forms the basis of a specific subset-logic language called **SuRE** (derived from **Su**bsets, **R**elations, and **E**quations), which is an implemented language. We therefore loosely call any subset-logic language as a SuRE language and their programs as SuRE programs.

We give some examples of SuRE programs below, to convey an idea of programming in the language. First we roughly describe the clausal forms of SuRE. The program clauses take the general form

$$A \leftarrow L_1, \ldots, L_m$$

where A is the head of the clause and $L_1, \ldots, L_m$ its body. We have $m \geq 0$ and for each $1 \leq i \leq m$, L_i is an atom or the negation of an atom. A is also an atom and atoms can take the forms $f(\vec{t}) \approx e$, $f(\vec{t}) \supseteqsim e$, or $p(\vec{t})$. When the head atom takes these forms it is called, accordingly, an equational, subset, or relational assertion. In addition, the body atoms can involve set constraints such as $t_1 = t_2$ and $t_1 \in t_2$. Here f and p represent user-defined functions and predicates respectively. The $\vec{t}$ represent a tuple of terms and t_1, t_2 represents terms, i.e. data objects formed from variables, constructors, and constants; and e represents an expression, formed from terms and other user-defined function symbols.

We distinguish between $=$, $\supseteq$ and $\approx$, $\supseteqsim$ on technical grounds, i.e., the user-defined function symbols may be undefined for certain inputs, whereas the former symbols are meant for total definedness. Intuitively, however, their meanings can be taken to be the same.

As is usual in logic programming, the reading of the program clause is that the atom A is asserted provided the literals L_1 through L_m hold. A subset-logic program is a collection of program clauses. A goal clause is the same as the body of a program clause, i.e., of the form $\leftarrow L_1, \ldots, L_m$, with the proviso that literals of the form $f(\vec{t}) \approx t'$, $f(\vec{t}) \supseteqsim t'$ can be invoked only when its tuple of terms $\vec{t}$ has become instantiated to ground terms.

Equational assertions enable the user to specify function definitions. For example, $f(\vec{t}) \approx e \leftarrow$ asserts that the user-defined function f, acting on the data arguments $\vec{t}$, has the value e. Subset assertions enable the user to specify functions in terms of set containment. Thus, $f(\vec{t}) \supseteqsim e \leftarrow$ asserts that f acting on $\vec{t}$ contains all possible instances of the set e. Evaluating $f(\vec{t})$ involves a *collect-all* operation, which is the union of all the instances of e. If there are multiple subset assertions for $f(\vec{t})$, then the collect-all is over the right-hand-sides of all these definitions. Relational assertions $p(\vec{t})$ are as usual with the difference that sets are treated as first-class objects in the data tuple of terms $\vec{t}$.

To represent finite sets, we use two constructors, namely, $\{x/t\}$ and $\{x\backslash t\}$, to denote nonempty sets, and the constructor $\emptyset$ to denote the empty set. The constructors $\{x/t\}$ and $\{x\backslash t\}$ both represent the set $\{x\} \cup t$, and are analogous to list representation. They differ in that $x \in t$ is permitted in the former while $x \notin t$ must hold in the latter. Thus $\{x/t\}$ means that x is some element of the set and t is a subset of the set such that $\{x\}$ and t covers the whole set; while $\{x\backslash t\}$ means that x is some element of the set and t is the corresponding remainder of the set. Taking an example, $\{1, 2\}$ can be represented by $\{1/\{2/\emptyset\}\}$ or $\{1/\{1/\{2/\emptyset\}\}\}$ or $\{1\backslash\{2\backslash\emptyset\}\}$ but not by $\{1\backslash\{1\backslash\{2\backslash\emptyset\}\}\}$. As with lists, whenever convenient, we will use the usual notation of $\{a_1, a_2, \ldots, a_n\}$ for sets.

In all of the SuRE programs below, identifiers beginning in `x`, `y`, `z`, `u`, `v`, `w` are considered as variables, and all other identifiers as constructors, functions and predicates. However, we use Prolog syntax when describing Prolog programs (e.g., recall that variables begin in an uppercase letter). The next few program examples show the use of set terms, equations and subset assertions.

In the program `P1`, `intersect` defines the intersection of two sets.

```
P1:  intersect({x\_},{x\_}) ⊇ {x}
```

We have used the anonymous variable (_) as is usual in Prolog. Declaratively, the assertion states that if `x` is any element of the first set and is also an element of the second set, then it is an element in their intersection. Then, by the collect-all assumption, the intersection is defined as the union over all these common singletons. Operationally, suppose we have the goal $\leftarrow$ `intersect({1,2,3}, {2,3,4})` $\approx$ `z`. Now, set-matching is used to solve the constraint $\{1,2,3\} = \{x\backslash y1\}$ and $\{2,3,4\} = \{x\backslash y2\}$. The first equality leads to the multiple matches $\{x \mapsto 1, y1 \mapsto \{2,3\}\}$, $\{x \mapsto 2, y1 \mapsto \{1,3\}\}$, and

$\{x \mapsto 3, y1 \mapsto \{1,2\}\}$. However, in satisfying the second equality, we obtain the only two matches of x as 2 and 3. Now a collect-all operation gives the value of z as $\{2\} \cup \{3\} = \{2,3\}$.

Similarly, the goal $\leftarrow$ `intersect(`$\emptyset$`, {2,3,4})` $\approx$ **z** leads to no matches, and the collect-all, being over an empty set of matches, leads vacuously to the value of $z = \emptyset$.

We see that we have a very succinct and readable program compared to an equivalent Prolog version that simulates sets with lists. The set constructor `{x\_}`, through the matching operation, affords a natural means of iterating over the elements of a set; and this helps to avoid the use of recursion. The use of the set constructor $\{x/y\}$ is shown by program `P2` which converts a list into a set. Here, in the second clause, **x** may well be in the set `list_to_set(y)`.

```
P2:  list_to_set([]) ≈ ∅

     list_to_set([x|y]) ≈ {x/list_to_set(y)}
```

The next program **size** computes the cardinality of a finite set. Here **succ** means successor.

```
P3:  size(∅) ≈ 0

     size({x\y}) ≈ succ(size(y))
```

Of note here is that multiple matches can arise in evaluating equational assertions too, but only one (any) match is considered for further evaluation. Thus, for the goal $\leftarrow$ `size({1,2})` $\approx$ **z** the matches of $\{x \mapsto 1, y \mapsto \{2\}\}$ and $\{x \mapsto 2, y \mapsto \{1\}\}$ arise, but only, say, the former is used in the further evaluation of `succ(size(y))`. Thus, it is assumed that all matches with an equational assertion leads to the same value.

The next two programs, both of which define the powerset of a set through **powerset**, shows a useful aspect of equations. Program `P4` gives the naive version while `P4`$'$ gives the more efficient one.

```
P4:  powerset(∅) ≈ {∅}

     powerset({x\y}) ⊇≈ powerset(y)
```

$\texttt{powerset(\{x\textbackslash y\})} \gtrsim \texttt{power1(x, powerset(y))}$

$\texttt{power1(x, \{y\textbackslash z\})} \gtrsim \texttt{\{\{x/y\}\}}$

P4$'$: $\texttt{powerset(}\emptyset\texttt{)} \approx \{\emptyset\}$

$\texttt{powerset(\{x\textbackslash y\})} \approx \texttt{power2(x, powerset(y))}$

$\texttt{power2(x, y)} \gtrsim \texttt{y}$

$\texttt{power2(x, \{y\textbackslash z\})} \gtrsim \texttt{\{\{x/y\}\}}$

In P4, there is much overcomputation due to the second clause, since the collect-all operation collects over many duplicate elements. For example, $\leftarrow$ `powerset({1,2,3})` $\approx$ `z` would evaluate `powerset({1,2})`, `powerset({2,3})`, and `powerset({1,3})` arising from the multiple matches in the second clause. These lead respectively to the values $\{\emptyset, \{1\}, \{2\}, \{1,2\}\}$, $\{\emptyset, \{2\}, \{3\}, \{2,3\}\}$, & $\{\emptyset, \{1\}, \{3\}, \{1,3\}\}$, and we can see that in the collect-all, $\emptyset$, $\{1\}$ among others are duplicated. Such overcomputation is avoided in P4$'$ by the use of the second equational clause.

SuRE programs can be used to simulate Prolog's `setof` feature in a more declarative manner. For example, given the usual `append` relation in Prolog,

```
append([], X, X).
append([X|U], Y, [X|W]) :- append(U, Y, W).
```

the Prolog goal for defining the different partitions of the list `[1,2,3]` is expressed by `?- setof([X|Y], append(X,Y, [1,2,3]), U)`. A more readable and semantically clearer SuRE version is the program P5 followed by the goal $\leftarrow$ `parts([1,2,3])` $\approx$ `u`.

P5: `append([], x, x)`

`append([x|u], y, [x|w])` $\leftarrow$ `append(u, y, w)`

`parts(z)` $\gtrsim$ `{[x|y]}` $\leftarrow$ `append(x, y, z)`

The `parts` clause also gives useful mode information for the compiler in that `parts` will be invoked with `z` a ground term. Hence, the compilation of `append` relation can make simplifications in the compiled code using this information.

The use of set terms in relations necessitates the use of set-unification rather than set-matching, and in a more general setting, the solving of set constraints involving $=$, $\in$, $\neq$, and $\notin$ between terms. These can be used in some succinct ways. For example, the $\in$ predicate can be used to insert an element into a set `y`, if not already present, via a goal such as $\leftarrow$ `10` $\in$ `y`. Here, there is only *one solution* to the above goal, namely, `y = {10\_}`. Similarly, the $\in$ predicate may be successively invoked to add an element to `y` if not already present. This is unlike the Prolog `member` predicate below, with lists representing sets.

```
member(X, [X|_]).
member(X, [Y|Z]) :- member(X, Z).
```

Here, for the goal `?- member(10, Y)` one obtains the infinite sequence of answers `Y = [10|Z]`, `Y = [X1,10|Z]`, `Y = [X1,X2,10|Z]`, ...

Some other SuRE program examples with relations are `P6` and `P7` below. In `P6`, `permute_set` intends to convert a set of n elements into a list of some permutation of the elements, with no duplications.

```
P6:  permute_set(∅, [])
     permute_set({x\y}, [x|z]) ← permute_set(y,z)
```

Program `P7` shows the use of negative set constraints in the definition of `setdiff` denoting set difference.

```
P7:  setdiff(x, y) ⊇ {z} ← z ∈ x, z ∉ y
```

1.3 Semantics of Subset-Logic Languages

Logic programs can be given *logical semantics*, that is, what relations 'follow' from the program, based upon well understood concepts of mathematical logic. Such semantics is declarative, being independent of any inference rules used to compute in the language. Logical semantics also entails giving a separate *procedural (operational) semantics* that captures the computation strategy, and showing its equivalence to the declarative semantics by means of soundness and completeness results.

Logical semantics differs from other traditional styles of semantics such as *denotational*, *axiomatic*, or *operational* semantics in that logic programs possess a declarative meaning given purely *by logic*, by notions such as truth and models. This is a level more abstract than the other styles permit. Denotational semantics usually ascribes mathematical functions expressed through least fixed points as the denotations of programs. As such, this is also independent of the computational model used to execute the programs. However, these mathematical descriptions become very involved for even small programs.

The axiomatic method of semantics provides inference rules involving formulae of the form $\{p\}\mathcal{S}\{q\}$ for deducing correctness statements about programs. Here p, q are correctness statements and $\mathcal{S}$ is a program fragment. The correctness statements are in some formal logical language related to the specification language, and is different from the programming language itself. In the case of logic programming, the specification and programming languages are so close that it is easy to be assured of the correctness of the program without the machinery of axiomatic semantics.

In operational semantics, the meanings of programs are understood primarily in terms of execution in some abstract machine model. This method of semantics is more intuitive and also provides guidance for the implementation of the language. However, while such a semantics is a necessary component of any programming language, it is not abstract enough to be the sole semantic component. For the abstract machine can be described in a variety of ways and too many details of the machine inessential to the outcome of the program have to be considered.

In this dissertation, we investigate the semantics of the combination of three logical forms —

equational, subset, and relational assertions. Such an investigation not only gives semantics for the programs of the subset-logic languages but also influences the design of the syntax of these languages. Among the subset-logic languages, the subset-equational and the subset-relational languages alone comprise a significant problem for study. As a result, *the goal of this dissertation is to formulate the declarative and operational semantics for the subset-equational and subset-relational languages.*

In declarative semantics we investigate *logical-consequence* and *model-theoretic* semantics. In both cases, one identifies, for a given program, a collection D of ground literals that, in the former case, are logical consequences of the program or of some natural modification of it; and, in the latter case, is a canonical model of the program that is natural in some sense. We examine inductive ways of obtaining D in terms of fixpoint operators. In operational semantics, we examine top-down strategies of deduction since such strategies have well-understood implementation techniques in the literature (e.g., the Warren Abstract Machine [Aït91]). Also, we look for soundness and completeness results.

1.4 Overview

We give below a more detailed description of the kinds of issues and problems encountered in the subset-equational and subset-relational languages, and thereby present an overview of the dissertation.

A treatment of the subset-equational language has been outlined in [JP89] in terms of a sketch of main ideas. What is lacking there is a *formalisation* of these ideas. Formalisation is vital to logical semantics; only it can bear out whether the ideas are supported or not. Secondly, a semantic aspect in there called *emptiness-as-failure (eaf)* has not been given a clean logical treatment. In our approach, we provide a rigorous formalisation as well as an improvement with respect to the emptiness as failure semantics. We elaborate on these aspects below.

• Foundations of Set Constructors

In [JP89], a set constructor $\{x \mid y\}$ was used to represent finite sets in the sense of element and remainder, i.e., $\{x \mid y\}$ was exactly our $\{x \backslash y\}$. However, $\{x \mid y\}$ was also being used in places where $\{x / y\}$ was appropriate. Thus a distinction between the two kinds of set constructors was not made. Also, while the intended meaning of $\{x \mid y\}$ was $x \notin y$, this could not be enforced in the absence of appropriate axioms. For example, while we mean $\{1 \mid \{2\}\}$ to be valid usage, we cannot ensure this without some axioms such as $1 \notin \{2\}$ or equivalently $1 \neq 2$. Further, in clausal forms involving $\{x \mid y\}$, the logical reading of the clause was based on, as is usual, a universal closure of the variables in the clause. Thus, all instantiations for x, y, including $x \in y$, were used in asserting the clause, thereby allowing unintended meanings for $\{x \mid y\}$ to creep in.

The emptiness as failure assumption involved denoting an undefined object by '?' and using equalities such as $\{?\} = \emptyset$ and $\{? \mid y\} = y$. That is, when a collection of objects was made into a set, any undefined object was cast out, considered as not contributing to the set. However, this is not standard usage for sets, and brings into sharper focus the need to justify that our set constructors do indeed behave like finite sets.

Lastly, in constructing the Herbrand domain from the set of ground terms, a quotient structure has to be formed based on the equality relations between set terms. In [JP89] and elsewhere in the literature, this equality relation has been taken to be $\{x \mid \{y \mid z\}\} = \{y \mid \{x \mid z\}\}$ and $\{x \mid \{x \mid z\}\} = \{x \mid z\}$, namely, the 'commutativity' and 'idempotency' properties of the set constructor. While these are certainly intuitive identities, it leaves open the question of whether they are a sufficient set of identities. For example, should extensionality or some other of the axioms of classical set theory play a part?

The above issues indicate the necessity of developing a foundation for our set constructors. For example, we need to be clear about exactly what assumptions are to be made about them, and show that they indeed have the behaviour of finite sets. Accordingly, we give such a development in chapters 3, 4, and 5.

In chapter 3, we give an intuitive collection of axioms called $SetAx$ for the set constructors. These include a set of *freeness axioms* so that inequality and hence nonmembership relations can be deduced. We also give a *disjointness transformation* for clauses containing terms like $\{x \backslash y\}$, to enforce their intended meaning. A natural concern for $SetAx$ is its consistency, especially given the presence of inequality assertions in its freeness axioms. We therefore build a model for $SetAx$, which is through an inductive construction. Finally, we show that our set constructors indeed have the behaviour expected of finite sets by relating them to classical set theory.

In chapter 4, we broaden unification to solving positive set constraints involving set type, equality and membership. We give a rewrite-rule based description of a unification algorithm that is based on logical consequence in $SetAx$. We discuss the common concerns of unification such as completeness and minimality of its unifiers. We deduce properties of $SetAx$ from the algorithm and use them to justify that the freeness axioms of $SetAx$ are adequate. We also touch upon properties arising from matching.

In chapter 5, we establish the details of the standard or Herbrand structure. In addition, we show that the 'commutativity' and 'idempotency' of the set constructors is indeed adequate for deriving the quotient structure. Finally, we show that the Herbrand interpretations model $SetAx$, something desirable when considering the Herbrand models of logic programs in the presence of $SetAx$.

While no single component of this development of the foundation of the set constructors can be said to be very difficult, the overall development was unobvious and unexpectedly long. It represents a significant part of the dissertation. The same approach can be followed to justify any non-standard assumptions about set constructors that may be needed such as in the *eaf* assumption.

The development also gives guidance for the foundations of other finite constructors having equality theories, such as multisets.

- **Subset-Equational Language**

Coming now to the subset-equational language, which we introduce in chapter 7, we indicate the differences with [JP89] and describe our semantic components. We have chosen to treat the semantics in an untyped first-order framework, leaving a full-fledged typed system to be adopted at a later time, after the essential issues relating to sets and equations have been better understood. Thus, we do not make the weak type distinctions of set- and nonset-valued functions made earlier.

The status of user-defined function symbols in first-order logic would be that of total functions. This necessitates that the Herbrand domain be composed of data terms as well as expressions formed from the user-defined function symbols. However, wishing to keep the Herbrand domain consisting purely of elemental data terms, and from the fact that user-defined function definitions may not be total (e.g., $f(x) = f([x])$), we clarify that the user-defined functions be treated as relations. To make this distinction, we use $f(\vec{t}) \approx e$ and $f(\vec{t}) \underset{\sim}{\supseteq} e$ in place of $f(\vec{t}) = e$ and $f(\vec{t}) \supseteq e$. These are actually to be viewed as the relations $f_{\approx}(\vec{t}, e)$ and $f_{\underset{\sim}{\supseteq}}(\vec{t}, e)$. We add suitable axioms to enforce that the $f_{\approx}$ behave like partial functions. (Corresponding axioms for $f_{\underset{\sim}{\supseteq}}$ are not needed.)

Subset-equational programs undergo a transformation, called the completion, that more accurately reflects its intended meaning. In particular, the completion captures the collect-all assumption. We have modified this completion a little so as to replace the *eaf* assumption with standard logical means and avoid the use of the undefined symbol '?'. We have formalised the declarative semantics of a program in terms of logical consequences of its completion, and have given the proofs of the properties outlined in [JP89] such as the existence of a least Herbrand model.

We then give a treatment of the operational semantics of subset-equational programs. Earlier, a rewriting relation was used to describe this semantics that is aided by the use of the undefined symbol '?'. In our case, we formalise it in terms of inductively defined computation trees that is a modification of the usual SLD resolution of logic programming. Such a style would permit a uniform treatment when relations are also considered. Included in the above is the matching process together with its proof of correctness, completeness, and minimality.

Finally, we prove the soundness and completeness result between the declarative and operational semantics, by using an intermediary fixpoint semantics. The latter involves an immediate consequence operator T_P whose domain is not the usual powerset of the Herbrand Base and which unexpectedly is not a total operator as is usual in logic programming. This problem arises from the multiple matches that can apply to a single equational assertion. We give a modification of the standard theorems that enables the fixpoint semantics to go through cleanly, such as assuring the existence of the least fixed point of T_P.

The above treatment is unable to deal with database-like programs which typically have incomplete information and have to provide answers to queries on that basis. To treat such programs, we give an enhanced semantics in chapter 8 that is like the semantics of the *Clark completion* in logic programming. Here, the syntax of functional clauses permits a simplification in the syntactic statement of the completion. This makes the completion much more readable and usable than it otherwise would have been. The declarative semantics is defined in terms of the logical consequences from such a completion.

On the operational side, the additional factor is finite failure. As with finite failure for definite clause programs in logic programming, we need fair derivations. Such derivations are sound, but unlike the definite clause case, it is unable to be complete. We show this to arise from the use of matching rather than unification. Nevertheless, the operational semantics computes more than provided by the semantics in [JP89].

The overall formalisation for the subset-equational language was technically difficult and extended, mainly complicated by the disjointness transformation and the collect-all assumption.

- **Subset-Relational Language**

In chapter 9, we begin the investigation of the interaction of all three forms of assertions — equational, subset, and relational. We choose a syntax that excludes negations of these assertions since negation forms a whole additional dimension in the study of these three forms. We continue with the Clark completion style of logical-consequence semantics, using which we give a sketch of the declarative and operational semantics.

The new factors here are that the declarative semantics has to handle infinite sets that are easily generated by relations. Also, the operational semantics has to be based on unification and disunification rather than matching.

2 Background and Related Work

We give here a brief description of the background and related work. A standard introduction to logic programming semantics is [Llo87] and a very readable account of logic programming through Prolog is in [Bra90]. Research accounts on semantics in logic programming usually appear in the *Journal of Logic Programming (JLP)* and in the proceedings of two conferences on logic programming sponsored by the *Association for Logic Programming (ALP)*. These are the *International Logic Programming Symposium (ILPS)* and the *International Conference on Logic Programming (ICLP)*. There are also a number of other conferences, to which logic programming is pertinent, that publishes papers on logic programming semantics, such as the ACM sponsored *Symposium on Principles of Database Systems (PODS)*. Further pointers to other conferences and literature can be obtained from *Logic Programming*, the newsletter of the ALP.

2.1 Background

It is generally believed that computing with full first-order logic is infeasible. Hence, most attention has focused on a subset of first-order logic called the Horn clause forms for which elegant declarative semantics and quick computation procedures are known. Subsequent work has sought to broaden this base of logic programming in various directions. We cite below ideas and papers from this background that is relevant to our work. These ideas do not directly translate to the SuRE framework because sets and collect-all are fundamentally different from the usual treatment of data structures in logic programming.

- **Definite Clauses**

There is much work in declarative semantics, even for the semantically neat case of definite clause programs, (i.e., programs formed from Horn clauses of the form $A \leftarrow A_1, \ldots, A_n$ with A, A_1, $\ldots$, A_n being positive atoms). The original Kowalski-van Emden scheme (see [Llo87]) for declarative semantics in terms of ground atomic logical consequences of a program has been further extended in some directions. In [GM87], greater abstraction has been sought by using *initial* models; in [JL87], generalizations to non-Herbrand domains yet retaining the usual characteristics of logical semantics, in terms of logic programming schemes and constraint languages, has been studied; in [FLMP88], a new declarative semantics (which has been called *s-semantics* in [Tur89]), more faithful to SLD resolution has been given; and in [GS88], fully abstract and compositional semantics have been explored.

- **Negation**

A great number of approaches have been suggested for including negation in logic programs. These can be largely divided into logical-consequence and model-theoretic approaches. In the former category, well-known semantics are the Clark completion (see [Llo87]), and the Chan constructive negation ([Cha88]), further extended by Przymusinski in [Prz89]. In the latter category, a canonical model is taken as the meaning of a program. This approach was initiated by the idea of *stratification* (see [Min88]), and extended in a number of ways such as *stable* models ([GL88]), *well-founded* models ([vGRS91]), *well-founded-by-case* models ([Sch90]), and *valid* semantics ([BRSS92]). These models do not necessarily coincide always. Approaches through other systems of logic such as linear logic or modal logic have also been considered. A survey of most of these semantics in the context of negation as failure can be found in [She92].

- **Unification**

A survey of unification is given in [JK91]. The original Robinson's unification algorithm, usually called *syntactic unification*, has been studied in other domains having special properties described by equations. Such unification has been called *semantic unification*, and in particular, when the equations are those of associativity, commutativity, and idempotency, it is called ACI unification. Set unification is considered as a special case of ACI unification through the use of the union constructor which has these properties. However, this does not directly apply to SuRE where the set constructor is scons whose meaning is axiomatised by axioms most of which are not equations. An unification algorithm more in tune with our requirements is in [DOPR93], more about which we discuss in §4.3. We also require the theory of disunification, a survey of which can be found in [Com91] and an account on set-disunification can be found in [DR93].

- **Computation Strategies**

The above declarative semantics have their operational counterparts which are not necessarily complete inference strategies. Examples of refutation procedures are OLDT resolution [TS86], SLDNF resolution [Llo87], SLDCNF resolution [Cha88], SLS resolution [Prz88], and global SLS resolution [Ros89]. Restrictions on programs have also been explored to achieve completeness.

- **Equations**

There is much literature on combining functions and relations especially through equality ([DL86, GLMP91, GM87, HO82, Höl89, ODon85]). In [Jay90], the definition of (first order) functions through equational assertions in an operational model of reduction has been argued to be desirable. This approach has been taken in SEL language [JP89] and we will pursue it in the case of the SuRE language too.

2.2 Related Work

A number of declarative language proposals have been made to extend logic programming with sets. We briefly describe them below relative to the SuRE language proposal. We should also mention the language SETL ([SDDS86]) which is, however, an imperative language that includes sets.

- **LDL**

The language LDL (Logical Data Language) is described in [NT89] and its semantics is presented in [BNST91]. Its design is motivated by database considerations. It treats sets through enumeration and the *grouping* construct, and the latter is similar to our collect-all but in the context of relations rather than user-defined functions. It gives a model-theoretic semantics in terms of a minimal model based on a notion of preference between models. It uses bottom-up evaluation techniques in its operational model. It does not combine user-defined functions through equations.

- **LPS**

This language has been proposed in [Kup90] and is called LPS (Logic Programming with Sets). It is developed in a database context and is given a model-theoretic semantics in terms of minimal models. Its main feature is that it allows restricted universal quantifiers like $\forall x \in X$ in the bodies of clauses. It shows how such clauses can simulate the grouping construct of LDL, and how together with negation it is equivalent to LDL. However, the language does not allow definition by set abstraction to be expressed naturally and has not been given a practical operational semantics. Further, it does not integrate equations as SuRE does.

- **COL**

The language COL (Complex Object Language) ([AG91]) is a logic-based language for manipulating complex objects constructed from set and tuple constructors. Its design is very similar to ours, given that it includes functions and relations, but is designed in a database context. It gives model-theoretic semantics through minimal models for stratified programs. It uses bottom-up evaluation techniques for computation.

- **{log}**

The language {log} (read set-log) ([DOPR91, DOPR93]) is an extended logic programming language embodying a *scons*-like set constructor. It uses an axiomatic approach to define such a constructor, but does not give a rigorous justification of the axioms. It does not include functions and the collect-all, but provides restricted universal quantifiers and restricted forms of definition by set abstraction. It provides a model-theoretic semantics based on a least model. Unlike the above languages its design is not explicitly motivated by database considerations. Its operational semantics is influenced by top-down computation and constraint solving involving $=$, $\in$, $\neq$, and $\notin$ relations.

Apart from the difference between language constructs, the principal differences between SuRE and the above languages are in that we examine logical-consequence semantics, give a rigorous development for the set constructors, and explore top-down computation.

3 Set Constructors and Finite Sets

In studying the inclusion of sets in logic programs, it is natural to study finite sets at first. In this chapter, we give the constructors for representing these finite sets that are suitable for logic programming. We then give the logical theory of the set constructors through an axiomatic approach. Such an approach gives a framework for discussing the correctness of set-unification algorithms, defining the Herbrand structure, and establishing logical-consequence semantics for logic programs.

In the following sections, we develop the logical theory by giving axioms, showing their consistency, and establishing that the constructors indeed behave like finite sets. Together, these provide a rigorous foundation for the set constructors, a necessary characteristic when giving precise semantics for logic programs. The theory is not particular to the subset-logic languages, and can form the basis of other logic programming languages involving such set constructors.

We have not taken classical set theory, such as *Zermelo-Fraenkel* set theory, directly as our axiom system because their axioms are formulated for a general setting, dealing with both finite and infinite sets, and not making any assumptions about particular set constructors. Hence their axioms take a more complicated form such as in asserting the existence of particular sets like the union or powerset. With our use of set constructors, appropriate sets are guaranteed to exist by virtue of being nameable by terms formed from the set constructors. Thus our axioms take a simpler form and are more in tune with logic programming. We have, assuredly, adopted and adapted the classical theory wherever possible.

3.1 Set Constructors

A representation often chosen for finite sets is that of *scons*, parallel to the list constructor *cons*. We distinguish between two kinds of such set constructors: $scons(x, y)$ and $dscons(x, y)$. In both cases, the set represented is $\{x\} \cup y$. However, in the former, $x \in y$ is possible, whereas in the latter, $x \notin y$ is required to hold. The *dscons* is called the disjoint set constructor (from $\{x\} \cap y = \emptyset$). Both constructors find natural uses in specifying sets in logic programs. The *scons* is used for specifying sets in terms of parts that may well overlap with each other. The *dscons* is used for specifying sets in terms of an element and remainder.

We give a simple and natural axiomatisation called $SetAx$, designed around the *scons* constructor, to ascribe semantics to the set constructors. We also give the subset of $SetAx$ that is enough to axiomatise the *scons* constructor alone. The extra axioms of $SetAx$ are a set of freeness assertions for deducing nonmembership relations required in the use of *dscons*. After giving $SetAx$, we justify it as a suitable theory for finite sets in logic programming.

The logical framework we consider is a first-order language $\mathbf{L}$ with equality, having an alphabet Σ possessing a set of variables Σ_V, a set of constructor symbols Σ_C, a set of predicate symbols Σ_P, and auxiliary symbols. Constructor symbols are used to build data objects, i.e., terms. Typical symbols are: $u, v, w, x, y, z, u_0, v', \ldots$ for variables, $a, b, c, d, a_0, b', \ldots$ for constructor symbols, $p, q, r, p_0, q', \ldots$ for predicate symbols, and $s, t, s_0, t', \ldots$ for terms.

We require that Σ_C contain the symbols $\{scons, \emptyset\}$ to represent set data, and Σ_P contain the symbols $\{=, set, \in\}$ to represent set predicates (we include equality among the set predicates). As a result, we will need the usual axioms of equality, viz., reflexivity and substitutivity, which we call $EqAx$ (and which we lay out in §3.3). Let $\Sigma_{\overline{C}}$ denote $\Sigma_C \backslash \{scons, \emptyset\}$ to refer to the non-set constructor symbols. The constant $\emptyset$ is used to denote the empty set, and the term $scons(s, t)$ intends to denote $\{s\} \cup t$ if t denotes a set. We do not take *dscons* as a primitive constructor, but will introduce it as a conditionally defined symbol since it is very close in meaning to *scons*. Terms of the form $scons(s, t)$ and $dscons(s, t)$ are called scons terms and dscons terms respectively. All terms that are not sets will be considered as individuals. More suggestive notation for $scons(x, y)$ and $dscons(x, y)$ are $\{x/y\}$ and $\{x\backslash y\}$ respectively. The symbol $\{x\backslash y\}$ evokes the idea of set-difference, which is about what is different or apart about two sets. Similarly, in $\{x\backslash y\}$ we mean that x and y are apart, i.e.x may not be in y. The symbol $\{x/y\}$ is chosen to be opposite to $\{x\backslash y\}$ since x may be in y in $\{x/y\}$. Thus $\{1, 2\}$ can be represented by $\{1/\{2/\emptyset\}\}$, $\{1\backslash\{2/\emptyset\}\}$, $\{2/\{1/\{2/\emptyset\}\}\}$, $\{1\backslash\{2\backslash\emptyset\}\}$, $\{2\backslash\{1\backslash\emptyset\}\}$, but not by $\{2\backslash\{1\backslash\{2\backslash\emptyset\}\}\}$.

Some conventions are as follows. Analogous to list notation, $\{t_1, t_2, \ldots, t_n/t_{n+1}\}$ denotes $\{t_1/\{t_2/\cdots/\{t_n/t_{n+1}\}\cdots\}$ for $n \geq 0$, and similarly for dscons terms, i.e., $\{t_1, t_2, \ldots, t_n\backslash t_{n+1}\}$ denotes $\{t_1\backslash\{t_2\backslash\cdots\backslash\{t_n\backslash t_{n+1}\}\cdots\}$ for $n \geq 0$. Thus, in this notation, we have $\{s_1, \ldots, s_m/\{t_1, \ldots, t_n/t\}\} = \{s_1, \ldots, s_m, t_1, \ldots, t_n/t\}$. When we know that a scons or dscons term is an individual, we will usually use *scons* or *dscons* instead of the brace notation, e.g., $scons(1, 2)$. The set of variables occurring in a syntactic object X will be denoted by $Var(X)$. An arrow over a symbol will denote a tuple of objects. Thus, $\vec{x}$, $\vec{x_0}$, $\vec{x'}$, ..., and $\vec{t}$, $\vec{t_0}$, $\vec{t'}$, ...will denote tuples of variables and terms respectively. We will assume $\vec{x_0}$ represents the tuple $(x_0)_1, (x_0)_2, \ldots, (x_0)_n$ for some $n \geq 0$ and n will be apparent from the context. Similarly, $\vec{x} = \vec{y}$ will be notation for $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$ for some $n \geq 0$.

Further conventions are as follows. In stating logical formulae we will usually omit parentheses with the understanding that the usual precedence rules of the logical operators apply. These are, in order from highest to lowest: $\exists$, $\forall$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. $\forall\varphi$ represents the universal closure of a formula φ, and $\exists\varphi$ represents its existential closure. $(\forall\psi)\varphi$ and $(\exists\psi)\varphi$ represent closure by relativised quantifiers, denoting, respectively, $\forall(\psi \rightarrow \varphi)$ and $\exists(\psi \wedge \varphi)$. Also, $\equiv$ will be used for syntactic identity.

We will often need to do induction on terms and so define a function $size(t)$ as the number of symbols in t, i.e., the number of nodes in the tree representation of t. We will also loosely refer to 'induction on $size(t)$' by 'induction on structure of t' or more simply by 'induction on t'. The possible syntactic forms a term may have are, as usual, x or $c(t_1, \ldots, t_n)$, for some $n \geq 0$. For a term t, let $init(t)$ be its initial symbol, i.e., $init(x) = x$ and $init(c(t_1, \ldots, t_n)) = c$.

An alternate useful set of syntactic forms for terms is as follows. Every term has one of the forms:

x,

$\emptyset$,

$c(t_1, \ldots, t_n)$ for some n-ary $c \in \Sigma_{\overline{C}}$,

$\{t_1, t_2, \ldots, t_m/x\}$ for some $m \geq 1$,

$\{t_1, t_2, \ldots, t_m/\emptyset\}$ for some $m \geq 1$, or

$\{t_1, t_2, \ldots, t_m/c(\vec{t'})\}$ for some $m \geq 1$ and some $c \in \Sigma_{\overline{C}}$.

In the last three forms, we may also consider $m = 0$ and take the terms to be, respectively, x, $\emptyset$, and $c(\vec{t'})$. In this case, the first three forms fold into the last three. Thus any term t has the form $\{t_1, \ldots, t_m/t'\}$, for some $m \geq 0$ and some t' whose initial symbol is not *scons*, and this form is unique. So we define a function $last(t)$ to be $last(t) = t'$.

Examples of the use of set constructors are programs `P1` and `P2` in chapter 1, §1.2.

We will take recourse to introducing new symbols in our first-order language as defined symbols. These defined symbols should be *eliminable* and *non-creative*, i.e., it should be possible to replace them by equivalent logical formulae involving only the primitive symbols, and any consequence of the defined symbols should be equivalently derivable from the primitive symbols alone.

The definitions we give will usually be conditional, i.e., of the form $\delta \rightarrow (\rho \leftrightarrow \varphi)$, where δ is the interesting condition, ρ is the atom introducing the defined symbol, and φ is the definition given in terms of primitive and previously defined symbols. A conditional definition can be made unconditional by giving some arbitrary *don't care* definition for the case when the condition does not hold, i.e., by specifying $\neg\delta \rightarrow (\rho \leftrightarrow \psi)$, where ψ is the don't care definition. Then the unconditional definition becomes $\rho \leftrightarrow ((\delta \rightarrow \varphi) \wedge (\neg\delta \rightarrow \psi))$. (For more on the theory of definitions see [Sup57], [Sup72], or [End72].)

3.2 Disjointness Transformation

We take *dscons* to be conditionally defined as follows. (All our logical definitions are labelled by Dn.) When $x \in y$, we regard it as a 'don't care' situation.

(D1): $x \notin y \rightarrow \{x \backslash y\} = \{x/y\}$

We assume that programs in **L** are in clausal form. Clauses that contain dscons terms have to undergo a transformation, called the *disjointness transformation*, because of the way we view the *dscons* constructor. The transformation adds appropriate nonmembership atoms to the bodies of these clauses. For example, if $\{x \backslash \{y \backslash z\}\}$ is used in a clause C, then it is changed to $\{x/\{y/z\}\}$, and $y \notin z \wedge x \notin \{y/z\}$ is added to the body of C. The transform, applied to `P6` in section §1.2 would yield

```
permute_set({x/y}, [x|z]) ← permute_set(y,z), x∉y
```

Such a transformation says that the original clause is asserted provided the relevant conditions in the conditional definition of *dscons* hold. When the conditions do not hold, the head of the clause is not asserted. Without such a transform, since clauses are universally quantified, an instance of the clause C could well have $y \in z$ or $x \in \{y/z\}$. The meaning of the clause would then depend upon some arbitrary don't care case of the definition of *dscons*, something that is not desirable.

Formally, the transformation is given by means of the operations $\tilde{t}$ and $nonmem(t)$. The operation $nonmem(t)$ explicates all the relevant nonmembership conditions arising from t, and the operation $\tilde{t}$ converts all the *dscons* in t to *scons*. It is convenient to denote the application of these operations to tuples of terms as follows. Let $\vec{t} = (t_1, \ldots, t_m)$. Then, $\tilde{\vec{t}} = (\tilde{t_1}, \ldots, \tilde{t_m})$, and $nonmem(\vec{t}) = nonmem(t_1) \wedge \cdots \wedge nonmem(t_m)$.

Let t be a term possibly containing dscons symbols. Then $\tilde{t}$ is given by:

(i) $\tilde{x} = x$

(ii) $\widetilde{c(\vec{t})} = c(\tilde{\vec{t}}), \quad c \in \Sigma_C$

(iii) $\widetilde{\{t_1 \backslash t_2\}} = \{\tilde{t_1}/\tilde{t_2}\}$.

Let t be a term possibly containing dscons symbols. Then $nonmem(t)$ is given by:

(i) $nonmem(x) = true$

(ii) $nonmem(c(\vec{t})) = nonmem(\vec{t}), \qquad \text{for } c \in \Sigma_C$

(iii) $nonmem(\{t_1 \backslash t_2\}) = nonmem(t_1) \wedge nonmem(t_2) \wedge \tilde{t_1} \notin \tilde{t_2}$.

It is easy to see the following by induction on t. We have $\tilde{t}$ is simply the term t with all *dscons* replaced by *scons*, and $nonmem(t)$ is a conjunction of nonmembership literals, i.e., each literal is of the form $s \notin t$, with no *dscons* appearing in it. Also, if t does not contain *dscons*, then $\tilde{t} \equiv t$ and $nonmem(t) = true$.

The next proposition, in the first part, says that for a term t possibly containing dscons symbols, if $nonmem(t)$ holds then the dscons symbols in t can be understood in the intended sense.

Proposition 3.2.1 Let t be a term possibly containing dscons symbols. Then

(i) $EqAx \models nonmem(t) \rightarrow t = \tilde{t}$

(ii) $EqAx \models nonmem(t) \rightarrow nonmem(s)$, for any subterm s of t.

Proof: Straightforward, by induction on $size(t)$. We take cases on the structure of t as used in the definitions of $\tilde{t}$ and $nonmem(t)$. We show a representative case for (i).

Case: $t \equiv \{t_1 \backslash t_2\}$. So $\tilde{t} = \{\tilde{t_1}/\tilde{t_2}\}$ and $nonmem(t) = nonmem(t_1) \wedge nonmem(t_2) \wedge \tilde{t_1} \notin \tilde{t_2}$. Now assume $nonmem(t)$. By induction hypothesis, $EqAx \models nonmem(t_i) \rightarrow t_i = \tilde{t_i}$, for $i = 1, 2$. Therefore $EqAx \models t_i = \tilde{t_i}$. By D1, we have $\{\tilde{t_1} \backslash \tilde{t_2}\} = \{\tilde{t_1}/\tilde{t_2}\}$. Thus $\tilde{t} = \{\tilde{t_1}/\tilde{t_2}\} = \{\tilde{t_1} \backslash \tilde{t_2}\} = \{t_1 \backslash t_2\} = t$. ■

The above proposition is easily extended to tuples of terms. Thus, $EqAx \models nonmem(\vec{t}) \rightarrow \vec{t} = \tilde{\vec{t}}$, for $\vec{t}$ a tuple of terms possibly containing dscons symbols.

Let $\varphi(\vec{s})$ be a quantifier-free formula with $\vec{s}$ being the tuple of terms occuring in φ, and possibly containing dscons subterms. Then, $(\forall nonmem(\vec{s}))\varphi(\tilde{\vec{s}})$ expresses that φ is universally closed relative to the intended usage of *dscons*. Similarly, $(\exists nonmem(\vec{s}))\varphi(\tilde{\vec{s}})$ expresses that φ is existentially closed relative to the intended usage of *dscons*. In particular, when φ is a clause, we have the following definition.

Definition 3.2.2 Let C be the clause $A(\vec{s}) \leftarrow B(\vec{t})$, where A is the head and B is the body of the clause, and $\vec{s}$ and $\vec{t}$ are the lists of terms appearing in the head and body. Then, the **disjoint transform** of the clause C, $disj(C)$, is given by: $A(\tilde{\vec{s}}) \leftarrow B(\tilde{\vec{t}}), nonmem(\vec{s}), nonmem(\vec{t})$.

Note that in the above definition, $A(\tilde{\vec{s}}) \leftarrow B(\tilde{\vec{t}}), nonmem(\vec{s}), nonmem(\vec{t})$ is logically equivalent under $EqAx$ to $A(\vec{s}) \leftarrow B(\vec{t}), nonmem(\vec{s}), nonmem(\vec{t})$ by virtue of Prop. 3.2.1(i).

3.3 The Axioms SetAx

When using set-theoretic predicates and constructors, we have certain intended interpretations in mind — for example, we only want to consider from amongst those interpretations in which $\forall x(x \in \{x/\emptyset\})$ holds. The logical consequences of programs P are to be those φ that hold in all such intended set-theoretic interpretations that are models of P. We will characterise the desired interpretations by means of axioms. We will seek these axioms for structures that have only individuals and finite sets in their domains.

As in axiomatic set theory, such as *Zermelo-Fraenkel* (ZF) set theory ([FBLD73], [Sup72]), we need defined symbols such as $\subseteq$, $\{x \mid \varphi(x)\}$, and $\bigcup s$ (see [Sup72]). These defined symbols augment the set of terms and formulae. We make the following convention. In the rest of this chapter, and in Chapters 4 and 5, unless otherwise stated, a term will ordinarily contain only primitive symbols, while a formula may contain defined predicate symbols as well.

Since some of the definitions will need some of the axioms or their consequences for their justification, and since some of the axioms are better stated using some of the definitions, we will give the definitions, axioms and theorems in an interleaved manner. We adopt all the relevant definitions of ZF in $SetAx$, and additionally have some new ones such as $indiv$, denoting an individual. We repeat here the definition D1, for the sake of completeness.

(D1): $x \notin y \rightarrow \{x \backslash y\} = \{x/y\}$

(D2): $(y \subseteq z \leftrightarrow \forall x(x \in y \rightarrow x \in z)) \leftarrow set(y) \wedge set(z)$

(D3): $indiv(x) \leftrightarrow \neg set(x)$

Let $SetAx$ be the following axioms and axiom schemas:

Finite Sets ($FinSetAx$):

(FS1): $set(\emptyset)$

(FS2): $set(\{x/y\}) \leftarrow set(y)$

(FS3): $indiv(c(\vec{z}))$ for all $c \in \Sigma_{\overline{C}}$

(FS4): $indiv(\{x/y\}) \leftarrow indiv(y)$

(FS5): $induction(z, \vec{u}) \leftarrow set(z)$ where $induction(z, \vec{u}) \equiv$

$(\psi(\emptyset, \vec{u}) \wedge \forall x \forall y(set(y) \wedge x \in z \wedge y \subseteq z \wedge \psi(y, \vec{u}) \rightarrow \psi(\{x/y\}, \vec{u})) \rightarrow \psi(z, \vec{u}))$

and ψ involves only set predicates and definitions, and $\vec{u}$ are its extra free variables.

Membership ($MemAx$):

(M1): $x \notin \emptyset$

(M2): $x \notin y \leftarrow indiv(y)$

(M3): $(x \in \{y/z\} \leftrightarrow x = y \vee x \in z) \leftarrow set(z)$

(M4): $set(z) \wedge z \neq \emptyset \rightarrow \exists x(x \in z \wedge \forall y(y \in x \rightarrow y \notin z))$

Commutativity and Idempotency ($CoIdAx$):

(CI1): $\{x/\{y/z\}\} = \{y/\{x/z\}\} \leftarrow set(z)$

(CI2): $\{x/\{x/z\}\} = \{x/z\} \leftarrow set(z)$

Equality ($EqAx$):

(I1): $x = x$

(I2): $c(\vec{x}) = c(\vec{y}) \leftarrow \vec{x} = \vec{y}$ for all $c \in \Sigma_C$

(I3): $(p(\vec{x}) \leftrightarrow p(\vec{y})) \leftarrow \vec{x} = \vec{y}$ for all $p \in \Sigma_P$

Freeness ($FreeAx$):

(F1): $c(\vec{x}) \neq d(\vec{y})$ for all $c, d \in \Sigma_C$ such that $c \not\equiv d$

(F2): $c(\vec{x}) \neq c(\vec{y}) \leftarrow \neg(\vec{x} = \vec{y})$ for all $c \in \Sigma_{\overline{C}}$

(F3): $\big(scons(x_1, x_2) \neq scons(y_1, y_2) \leftarrow \neg(x_1 = y_1 \wedge x_2 = y_2)\big) \leftarrow indiv(x_2) \wedge indiv(y_2)$

(F4): $t[x] \neq x \leftarrow indiv(t[x])$ for all terms t containing x and different from x

(F5): $\{s[x]/y\} \neq x$ for all terms s containing x

We briefly motivate the above axioms. A formal justification is given in a later section. Axioms FS1–4 state which terms denote sets and which denote individuals. The induction axiom FS5 is needed to capture the notion of finiteness of sets, as is well known in set theory. While at first it may not appear natural, the need for it becomes apparent in proving even simple properties such as Prop. 3.3.2 below.

We note that the induction axiom could equivalently have been replaced by an axiom like $set(z) \rightarrow finite(z)$ expressing that all sets are finite. We have chosen the induction axiom only because it is more simply stated than definitions of $finite(z)$. Specifically, the latter involves the definition of powerset which is conveniently introduced after the axioms have been stated.

The membership axioms M1–3 state when two terms are in the member relation and when they are not. The axiom M4 is just the axiom of regularity of ZF and is included here since it does not seem to be deducible from the other axioms of $SetAx$. The axiom states that every non-empty set z has some member x whose elements are not in z. (Strictly, this axiom is not essential to ZF, but we include it so that we may facilely borrow from the results of ZF.)

Axioms CI1–2 express that finite sets do not differ on account of order or repetition of their elements in their enumerations. The equality axioms $EqAx$ are the usual rules of reflexivity and substitutivity of the equality predicate.

The freeness axioms $FreeAx$ are needed to assert certain nonmembership relations, for use with *dscons* constructor. Statements about (non)membership can be reduced to statements about (in)equalities and vice-versa. For example, the dscons term $\{1\backslash\{2/\emptyset\}\}$ is well-defined as a set if $1 \notin \{2/\emptyset\}$, i.e., if $1 \neq 2$. More generally, we want $FreeAx$ to be such that terms that are not provably equal from the rest of the axioms, can be proved unequal with the aid of $FreeAx$. A collection of freeness axioms have been given by Clark ([Llo87], §14) in a somewhat different context, for the case when no particular equality property within the data constructors is involved. By extending these axioms we obtain the axioms F1–5. Axiom F1 expresses that different constructor symbols lead to distinct terms. Axioms F2–3 express injectivity, while axioms F4–5 are like the occurs check axiom. The axiom F5 imitates axiom M4 to a certain extent. For example $x \notin x$ can be deduced from either.

There is a close resemblance between the freeness axioms and unification. The former is concerned with when terms are never equal, while the latter is concerned with both when terms can be equal and when never equal. Thus the failure or non-unifiable cases of unification have a correspondence with the freeness axioms.

Let $SetAx^- = SetAx \backslash FreeAx$. Then $SetAx^-$ is the appropriate theory when the *scons* constructor alone is considered, while $SetAx$ is the appropriate theory when both *scons* and *dscons* are considered. Also $SetAx$ is the appropriate theory when *scons* alone together with nonmembership relations are considered. We now list some simple and intuitively apparent properties of $SetAx$ and $SetAx^-$, which find use in subsequent proofs.

Proposition 3.3.1 $SetAx^- \models \varphi$, where φ is any of:

(i) $x \in y \rightarrow set(y)$

(ii) $x \in \{y/z\} \rightarrow set(z)$

(iii) $set(z) \rightarrow x \in \{x/z\}$

(iv) $x \in z \rightarrow x \in \{y/z\}$

(v) $set(z) \leftrightarrow set(\{x_1, \ldots, x_n/z\})$ for any $n \geq 0$

(vi) $set(z) \rightarrow (x \in \{x_1, \ldots, x_n/z\} \leftrightarrow x = x_1 \vee \cdots \vee x = x_n \vee x \in z)$ for any $n \geq 0$

(vii) $set(z) \rightarrow \{x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n/z\} =$

$$\{x_i, x_2, \ldots, x_{i-1}, x_1, x_{i+1}, \ldots, x_n/z\} \quad \text{for } i > 1, n \geq 2$$

(viii) $set(z) \wedge x_1 = x_i \rightarrow (\{x_1, x_2, \ldots, x_n/z\} = \{x_2, \ldots, x_n/z\} \wedge$

$$\{x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n/z\} = \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n/z\}) \quad \text{for } i > 1, n \geq 2$$

(ix) $indiv(z) \leftrightarrow indiv(scons(x_1, \ldots, scons(x_n, z) \ldots))$ for any $n \geq 0$

(x) $set(\{x_1, \ldots, x_n/\emptyset\})$ for any $n \geq 0$

(xi) $x \in \{x_1, \ldots, x_n/\emptyset\} \leftrightarrow x = x_1 \vee \cdots \vee x = x_n$ for any $n \geq 0$

Proof: Trivial. (i): This is the contrapositive of M2. (ii): Assume $x \in \{y/z\}$. So $set(\{y/z\})$ by (i). Hence $set(z)$ by contrapositive of FS4. (iii): Assume $set(z)$. Then $x \in \{x/z\}$ by M3. (iv): Assume $x \in z$. So $set(z)$. Then $x \in \{y/z\}$ by M3. (v): From FS2 and FS4 we get $set(\{x/y\}) \leftrightarrow set(y)$. Now use induction on n (in the metalanguage). (vi): By induction on n and using M3. (vii): Use induction on i and CI1. (viii): Follows from (vii) and CI2. (ix): This is just (v). (x): This is a corollary of (v). (xi): This is a corollary of (vi). ■

The next proposition, again intuitively apparent, is interesting for the use of the induction axiom in its proof. The properties do not seem easy to prove without its use.

Proposition 3.3.2 $SetAx^- \models \varphi$, where φ is any of:

(i) $x \in y \rightarrow \exists z(set(z) \wedge y = \{x/z\} \wedge x \notin z)$

(ii) $set(z) \rightarrow z = \emptyset \vee \exists x(x \in z)$

Proof: (i): Let $\psi(y,x) \equiv x \in y \rightarrow \exists z(set(z) \wedge y = \{x/z\} \wedge x \notin z)$.

Case (1): $\neg set(y)$. Then, $x \notin y$ by M2. So $\psi(y,x)$ (vacuously).

Case (2): $set(y)$. Then, $induction(y,x) \equiv$

$\psi(\emptyset, x) \wedge \forall x', y'(set(y'), x' \in y, y' \subseteq y, \psi(y', x) \rightarrow \psi(\{x'/y'\}, x)) \rightarrow \psi(y,x)$

Basis: We have $x \notin \emptyset$ by M1. So $\psi(y,x)$ (vacuously).

Induction step: Assume $set(y'), x' \in y, y' \subseteq y, \psi(y', x)$. Now,

$\psi(y', x) \equiv x \in y' \rightarrow \exists z(set(z) \wedge y' = \{x/z\} \wedge x \notin z)$, and

$\psi(\{x'/y'\}, x) \equiv x \in \{x'/y'\} \rightarrow \exists z(set(z) \wedge \{x'/y'\} = \{x/z\} \wedge x \notin z)$.

Assume $x \in \{x'/y'\}$. Then $x = x' \vee x \in y'$.

Case (2.1): $x \in y'$. Then $\exists z(set(z) \wedge y' = \{x/z\} \wedge x \notin z)$, by induction hypothesis. So $set(z_1) \wedge y' = \{x/z_1\} \wedge x \notin z_1$. Case (2.1.1): $x' = x$. Then $\{x'/y'\} = \{x'/\{x/z_1\}\} \stackrel{CI2}{=} \{x/z_1\}$. So, $set(z_1) \wedge \{x'/y'\} = \{x/z_1\} \wedge x \notin z_1$, i.e., $\exists z(set(z) \wedge \{x'/y'\} = \{x/z\} \wedge x \notin z)$. Case (2.1.2): $x' \neq x$. Then $\{x'/y'\} = \{x'/\{x/z_1\}\} \stackrel{CI1}{=} \{x/\{x'/z_1\}\}$. Now, $x \notin \{x'/z_1\}$ by M3, and $set(\{x'/z_1\})$. Thus, $\exists z(set(z) \wedge \{x'/y'\} = \{x/z\} \wedge x \notin z)$ by using $\{x'/z_1\}$ as witness for z.

Case (2.2): $x \notin y'$. So $x = x'$. Then, $\exists z(set(z) \wedge \{x'/y'\} = \{x/z\} \wedge x \notin z)$ by using y' as witness for z.

So, in all cases, we have $\exists z(set(z) \wedge \{x'/y'\} = \{x/z\} \wedge x \notin z)$, i.e., we have $\psi(\{x'/y'\}, x)$. Therefore $\psi(y,x)$.

(ii): Assume $set(z)$. Let $\psi(z) \equiv z = \emptyset \vee \exists x(x \in z)$. Then, $induction(z) \equiv \psi(\emptyset) \wedge \forall x', y'(set(y'), x' \in z, y' \subseteq z, \psi(y') \rightarrow \psi(\{x'/y'\})) \rightarrow \psi(z)$. Basis: $\psi(\emptyset)$ is immediate. Induction step: Assume $set(y'), x' \in z, y' \subseteq z, \psi(y')$. From Prop. 3.3.1(iii), $x' \in \{x'/y'\}$, i.e., $\exists x(x \in \{x'/y'\})$. Therefore, $\{x'/y'\} = \emptyset \vee \exists x(x \in \{x'/y'\})$, i.e., $\psi(\{x'/y'\})$. Note that we didn't need the induction hypothesis $\psi(y')$ here. ■

Some properties based on the above proposition are given next.

Proposition 3.3.3 $SetAx^- \models \varphi$, where φ is any of:

(i) $set(y) \rightarrow y = \emptyset \vee \exists x \exists z(set(z) \wedge y = \{x/z\} \wedge x \notin z)$

(ii) $x \in y \rightarrow \{x/y\} = y$

(iii) $set(y) \rightarrow (x_1 \in y \wedge \cdots \wedge x_n \in y \leftrightarrow \exists z(y = \{x_1, \ldots, x_n/z\})) \quad n \geq 0$

Proof: Trivial. (i): From previous proposition. (ii): From previous proposition and CI2. (iii): From the previous two propositions. ■

From Prop. 3.3.1(vi) and Prop. 3.3.3(iii) (with $n = 1$) we get that (non)membership is reducible to (in)equalities; and the reverse holds from Prop. 3.3.1(xi) (with $n = 1$).

In models of ZF, non-well-founded sets do not exist, such as an infinite descending sequence of sets $\cdots s_{k+1} \in s_k \in \cdots \in s_2 \in s_1$. Such sequences appear possible in models of $SetAx$ and $SetAx^-$ — for e.g., consider the sequence $s_i = \{s_{i+1}/\{i/\emptyset\}\}$ for all $i \geq 1$. (In ZF, infinite sets are used to show that such sequences cannot exist.) However, infinite sequences in which the sets repeat, i.e., $s_i = s_j$ for some $i \neq j$, do not exist, as shown by the next proposition. Such sequences are excluded by axiom F5 or by axiom M4.

Proposition 3.3.4 There is no sequence of sets $x_1, \ldots, x_n$ in $SetAx$, for any $n \geq 1$, such that

$$x_1 \in x_n \in \cdots \in x_2 \in x_1,$$

i.e., $SetAx \models \neg \exists x_1, \ldots, x_n(set(x_1) \wedge \cdots \wedge set(x_n) \wedge x_1 \in x_n \wedge \cdots \wedge x_2 \in x_1)$.

Proof: Suppose there were such a sequence. We will show a contradiction. We have $x_1 \in x_n$, and $x_{i+1} \in x_i$, for $1 \leq i \leq n-1$. So, $\exists z(x_n = \{x_1/z\})$, and $\exists z(x_i = \{x_{i+1}/z\})$, for $1 \leq i \leq n-1$. Therefore, $x_n = \{x_1/z_n\}$, and $x_i = \{x_{i+1}/z_i\}$ for $1 \leq i \leq n-1$ and new variables z_1 through z_n. We have, $x_1 = \{x_2/z_1\} = \{\{x_3/z_2\}/z_1\} = \cdots = \{\cdots\{x_n/z_{n-1}\}/\cdots/z_1\} = \{\cdots\{x_1/z_n\}/z_{n-1}\}/\cdots/z_1\} = \{s[x_1]/z_1\}$, where $s[x_1]$ is the term $\{\cdots\{x_1/z_n\}/z_{n-1}\}/\cdots/z_2\}$ and contains x_1. But $x_1 \neq \{s[x_1]/z_1\}$, by F5. Contradiction.

This property also holds in $SetAx^-$ where it is deducible from M4, in the same way it is in ZF, but this has to await the section §3.5 relating $SetAx^-$ to ZF. ■

Another property of $SetAx$, similar to the above, is given next.

Proposition 3.3.5 For any term s containing x, $SetAx \models set(x) \rightarrow s \notin x$.

Proof: From Prop. 3.3.2 and axiom F5. ■

3.4 Consistency of SetAx

Since the freeness axioms involve negation, the consistency of $SetAx$ becomes an added concern. Below, we exhibit a model of the set axioms $SetAx$. We define a structure $\mathcal{F} = \langle U, \Sigma_P, \Sigma_C \rangle$ and show that $\mathcal{F} \models SetAx$.

We inductively construct the universe U as the union of a collection I of individuals and a collection S of finite sets. Below, $\mathcal{P}_{fin}$ stands for the finite-powerset operator, i.e., it gives the collection of all *finite* subsets of a set. We represent all finite sets in the universe by enumerating their elements within braces.

$I_0 = \emptyset$

$S_0 = \emptyset$

$U_0 = I_0 \cup S_0$

and for $i \geq 1$

$I_i = \{c(\vec{x}) \mid c \in \Sigma_{\overline{C}}, c \text{ } n\text{-ary}, n \geq 0, \vec{x} \in U_{i-1}^n\} \cup \{scons(x_1, x_2) \mid x_1 \in U_{i-1}, x_2 \in I_{i-1}\}$

$S_i = \mathcal{P}_{fin}(U_{i-1})$

$U_i = I_i \cup S_i$

$$I = \bigcup_{i<\omega} I_i, \quad S = \bigcup_{i<\omega} S_i, \quad U = \bigcup_{i<\omega} U_i$$

Here, for $i = 1$, I_i and S_i simplify to $I_1 = \{c \mid c \text{ a constant in } \Sigma_C, c \not\equiv \emptyset\}$, and $S_1 = \{\{\}\}$. The following are implied in the construction. The objects in I_i, $i \geq 1$, are not sets. If $x, y \in S$, then $x = y$ iff they have the same elements. If $x, y \in I$ and $x \equiv a(\vec{u})$, $y \equiv b(\vec{v})$ for some $a, b \in \Sigma_{\overline{C}} \cup \{scons\}$ and some $\vec{u}, \vec{v}$ of the appropriate kind, then $x = y$ iff $a \equiv b$ and $\vec{u} = \vec{v}$. Some properties based on the above construction follow.

Proposition 3.4.1

(i) I_i, S_i, and U_i are monotonic in i, i.e., for $0 \leq i \leq j$, $I_i \subseteq I_j$, $S_i \subseteq S_j$, and $U_i \subseteq U_j$.

(ii) $I \cap S = \emptyset$

(iii) $I \cup S = U$

Proof: (i): Enough to show $I_i \subseteq I_{i+1}$, $S_i \subseteq S_{i+1}$, and $U_i \subseteq U_{i+1}$ for every $i \geq 0$. Use simultaneous induction on i. (ii): By construction. (iii): Trivial. ■

From this, it follows that if $\Sigma_{\overline{C}} = \emptyset$ then $I = \emptyset$ and U is a universe of pure sets. Also, if $i < j$ and $n \geq 1$ then $I_i^n \subseteq I_j^n$, $S_i^n \subseteq S_j^n$, $U_i^n \subseteq U_j^n$, and $U_i \times I_i \subseteq U_j \times I_j$.

Proposition 3.4.2 If $x \in I$ then there is a least $i > 0$ such that $x \in I_i$, and it is the unique i such that $x \in I_i \backslash I_{i-1}$. Similarly for $x \in S$ and $x \in U$.

Proof: Trivial. ■

It follows that if $x \in I_j$ and $x \in I_i \backslash I_{i-1}$, then $i \leq j$; and that $x \in U_i \backslash U_{i-1}$ iff $x \in I_i \backslash I_{i-1}$ or $x \in S_i \backslash S_{i-1}$. Also, if $\vec{x} \in U^n$, $n \geq 1$, then there is a least $i > 0$ such that $\vec{x} \in U_i^n$, and it is the unique i such that $\vec{x} \in U_i^n \backslash U_{i-1}^n$. Here i is the maximum of the least levels of occurrences of the components of the tuple, i.e., if for each j, $1 \leq j \leq n$, $x_j \in U_{i_j} \backslash U_{i_j - 1}$, then $i = \max(i_1, \ldots, i_n)$. The same holds for $\vec{x} \in I^n, S^n$, $n \geq 1$, and for $\vec{x} \in U \times I$.

Lemma 3.4.3

(i) Suppose $\vec{x} \in U_i^n \backslash U_{i-1}^n$, for some $n \geq 1$, $i \geq 1$, and $c \in \Sigma_{\overline{C}}$. Then $c(\vec{x}) \in I_{i+1} \backslash I_i$.

(ii) Suppose $(x_1, x_2) \in (U_i \times I_i)\backslash(U_{i-1} \times I_{i-1})$, for some $i \geq 1$. Then $scons(x_1, x_2) \in I_{i+1}\backslash I_i$.

(iii) Suppose $(x_1, \ldots, x_n) \in U_i^n\backslash U_{i-1}^n$, for some $n \geq 1$, $i \geq 1$. Then $\{x_1, \ldots, x_n\} \in S_{i+1}\backslash S_i$.

Proof: (i): Assume the hypothesis. Suppose $c(\vec{x}) \in I_i$. So $c(\vec{x}) = d(\vec{y})$, for some $d \in \Sigma_{\overline{C}} \cup \{scons\}$, $\vec{y} \in U_{i-1}^m$, (and $\vec{y}$ of the appropriate kind in case $d \equiv scons$). Then $c \equiv d$, $n = m$, $\vec{x} = \vec{y}$ and $\vec{x} \in U_{i-1}^n$. Contradiction. Therefore $c(\vec{x}) \notin I_i$, and $c(\vec{x}) \in I_{i+1}\backslash I_i$.

(ii), (iii): Similar to (i). ■

Definition 3.4.4 The constructor interpretations in $\mathcal{F}$ are given by:

(i) $c^{\mathcal{F}}(\vec{x}) = c(\vec{x})$ for $c \in \Sigma_{\overline{C}}$ and $\vec{x} \in U^n$,

(ii) $\emptyset^{\mathcal{F}} = \{\}$, and

(iii) $scons^{\mathcal{F}}(x, y) = \begin{cases} scons(x, y) & \text{if } x \in U, y \in I \\ \{x\} \cup y & \text{if } x \in U, y \in S \end{cases}$

Here, $c^{\mathcal{F}}(\vec{x}) \in I$ for $c \in \Sigma_{\overline{C}}$, $\emptyset^{\mathcal{F}} \in S$, and $scons^{\mathcal{F}}(x, y) \in S$ iff $y \in S$.

Proposition 3.4.5 Let $x, \vec{y}$ be free variables of a term t. Suppose x is assigned such that $x \in U_i\backslash U_{i-1}$, $i \geq 1$. Then, for any assignment $\vec{y} \in U^n$, $n \geq 0$, we have $t^{\mathcal{F}} \in U_j\backslash U_{j-1}$ and $j \geq i$.

Proof: Straightforward. Use induction on t and above lemma. Here, if $t \equiv \{s_1, \ldots, s_m/x\}$, $m \geq 0$ and x does not occur in $s_1, \ldots, s_m$, then $j = i$ is possible. In all other cases of t, we have $j > i$. ■

For predicate interpretations in $\mathcal{F}$, we need only specify interpretations for the set predicate symbols, and can leave the others arbitrary.

Definition 3.4.6 Let $x, y \in U$. Then

(i) $set^{\mathcal{F}}(x) \Leftrightarrow x \in S$

(ii) $x \in^{\mathcal{F}} y \Leftrightarrow y \in S$ and $x \in y$

(iii) $x =^{\mathcal{F}} y \Leftrightarrow x = y$

Here, $x \in y$ and $x = y$ have their usual meanings, viz., membership and identity.

Theorem 3.4.7 $\mathcal{F} \models SetAx$

Proof: By considering each set axiom in turn and using the above construction and its properties.

$FinSetAx$:

(FS1): $\{\} \in S$.

(FS2): Let $x, y \in U$. Assume $set^{\mathcal{F}}(y)$, i.e., $y \in S$. Then $\{x\} \cup y \in S$.

(FS3): Let $c \in \Sigma_{\overline{C}}$ with c n-ary, and $\vec{z} \in U^n$. So $\vec{z} \in U_i^n$ for some $i > 0$ and $c(\vec{z}) \in I_{i+1}$, i.e., $c(\vec{z}) \notin S$.

(FS4): Let $x, y \in U$. Assume $set^{\mathcal{F}}(scons^{\mathcal{F}}(x, y))$, i.e., $scons^{\mathcal{F}}(x, y) \in S$. So $x \in U, y \in S$, i.e., $set^{\mathcal{F}}(y)$.

(FS5): Let $z \in U$, and $\vec{u} \in U^n$, for some $n \geq 0$. Assume $set^{\mathcal{F}}(z)$, i.e., $z \in S$. Assume $\psi(\emptyset, \vec{u})$ and $\forall x \forall y(set(y), x \in z, y \subseteq z, \psi(y, \vec{u}) \rightarrow \psi(\{x/y\}, \vec{u}))$ are satisfied in $\mathcal{F}$. To show $(\psi(z, \vec{u}))^{\mathcal{F}}$. Case (1): $z = \{\}$. So $(\psi(z, \vec{u}))^{\mathcal{F}}$, from the assumption. Case (2): $z \neq \{\}$. Let $z = \{x_1, \ldots, x_k\}$, $k \geq 1$. Consider the sets $v_i \in U$, defined by: $v_i = \{x_1, \ldots, x_i\}$, for $0 \leq i \leq k$. Claim: For all $i \geq 0$, if $i \leq k$, then $(\psi(v_i, \vec{u}))^{\mathcal{F}}$ holds. Proof of the claim is by induction on i, and is trivial. Applying the claim to $i = k$, we get $v_k = z$ and $(\psi(z, \vec{u}))^{\mathcal{F}}$.

$MemAx$:

(M1): Let $x \in U$. Now $x \notin \{\}$. So $(x \notin \emptyset)^{\mathcal{F}}$ holds.

(M2): Let $x, y \in U$. Assume $(\neg set(y))^{\mathcal{F}}$, i.e., $y \notin S$. So $(x \in^{\mathcal{F}} y)$ is false, i.e., $(x \notin y)^{\mathcal{F}}$ holds.

(M3): Let $x, y \in U$ and assume $z \in S$. ($\rightarrow$): Assume $(x \in scons(y, z))^{\mathcal{F}}$. So $x \in \{y\} \cup z$. Therefore $x = y$ or $x \in z$. Hence $x = y$ or ($z \in S$ and $x \in z$), i.e., $(x = y \vee x \in z)^{\mathcal{F}}$ holds. ($\leftarrow$): Assume $(x = y \vee x \in z)^{\mathcal{F}}$, i.e., $x = y$ or ($z \in S$ and $x \in z$). We have $scons^{\mathcal{F}}(y, z) = \{y\} \cup z$, and $\{y\} \cup z \in S$. In either case of $x = y$ or $x \in z$, we get $x \in \{y\} \cup z$. So $(x \in \{y/z\})^{\mathcal{F}}$.

(M4): Assume $z \in S$ and $z \neq \{\}$. So $z = \{x_1, \ldots, x_m\}$ for some $m \geq 1$. If any $x_j \in I$ or $x_j = \{\}$, for $1 \leq j \leq m$, then x_j is a witness to the consequent. When every $x_j \in S$ and $x_j \neq \{\}$, for $1 \le j \le m$, then let i_j be the least level of occurrence of x_j. Choose $i = \min(i_1, \ldots, i_m)$ and let $x_l \in S_i \backslash S_{i-1}$. We have $x_l = \{y_1, \ldots, y_n\}$ for some $n \ge 1$, and $y_k \in U_{i-1}$, for $1 \le k \le n$. So least level of occurrence of any y_k is below i, whereas every element of z occurs at a least level at or above i. Therefore, no element of x_l is in z and x_l is a witness to the consequent.

CoIdAx:

(CI1): Let $x, y, z \in U$. Assume $set^{\mathcal{F}}(z)$, i.e., $z \in S$. We have $(\{x/\{y/z\}\})^{\mathcal{F}} = \{x\} \cup \{y\} \cup z = \{y\} \cup \{x\} \cup z = (\{y/\{x/z\}\})^{\mathcal{F}}$.

(CI2): Let $x, z \in U$. Assume $set^{\mathcal{F}}(z)$, i.e., $z \in S$. We have, $(\{x/\{x/z\}\})^{\mathcal{F}} = \{x\} \cup \{x\} \cup z = \{x\} \cup z = (\{x/z\})^{\mathcal{F}}$.

EqAx:

I1-3 are satisfied because $=^{\mathcal{F}}$ is identity.

FreeAx:

(F1): Let $c, d \in \Sigma_C$ with c m-ary, d n-ary, and $\vec{x} \in U^m$, $\vec{y} \in U^n$. Case (1): $c, d \in \Sigma_{\overline{C}}$. So $c(\vec{x}), d(\vec{y}) \in I$, $c \not\equiv d$. Also $c(\vec{x}) \ne d(\vec{y})$, by construction. Case (2): $c \in \Sigma_{\overline{C}}, d \equiv \emptyset$. So $c(\vec{x}) \in I$, $\{\} \in S$. Hence $c(\vec{x}) \ne \{\}$ by $I \cap S = \emptyset$. Case (3): $c \in \Sigma_{\overline{C}}, d \equiv scons$. Then $\vec{y} = (y_1, y_2)$, and $c(\vec{x}) \in I$. Case (3.1): $y_2 \in I$. So $scons(y_1, y_2) \in I$. Hence $c(\vec{x}) \ne scons(y_1, y_2)$, by construction. Case (3.2): $y_2 \in S$. So $\{y_1\} \cup y_2 \in S$. Hence $c(\vec{x}) \ne \{y_1\} \cup y_2$, by $I \cap S = \emptyset$. Case (4): $c \equiv \emptyset, d \equiv scons$. Then $\vec{y} = (y_1, y_2)$. Case (4.1): $y_2 \in I$. So $\{\} \ne scons(y_1, y_2)$, by $I \cap S = \emptyset$. Case (4.2): $y_2 \in S$. So $\{\} \ne \{y_1\} \cup y_2$.

Other cases of c, d are satisfied by symmetry of $=$.

(F2): We show the contrapositive is satisfied. Let $c \in \Sigma_{\overline{C}}$ with c n-ary, and $\vec{x}, \vec{y} \in U^n$. Assume $(c(\vec{x}) = c(\vec{y}))^{\mathcal{F}}$, i.e., $c(\vec{x}) = c(\vec{y})$. Then $c(\vec{x}), c(\vec{y}) \in I$. So $\vec{x} = \vec{y}$ and hence $(\vec{x} = \vec{y})^{\mathcal{F}}$.

(F3): Let $x_1, x_2, y_1, y_2 \in U$. Assume $indiv^{\mathcal{F}}(x_2), indiv^{\mathcal{F}}(y_2)$, i.e., $x_2 \in I, y_2 \in I$. So $scons^{\mathcal{F}}(x_1, x_2), scons^{\mathcal{F}}(y_1, y_2) \in I$. One can show that the contrapositive holds as in (F2) above.

(F4): Let t be any term containing x and different from x. So t is neither a variable nor a constant. Let $t \equiv c(t_1, \ldots, t_n)$, $n \geq 1$, $c \in \Sigma_C$. Let x occur in t_l, for some $1 \leq l \leq n$. Let $x, \vec{y}$ be the free variables of t and let $x \in U, \vec{y} \in U^m$ be an assignment of these variables. Assume $(indiv(t))^{\mathcal{F}}$, i.e., $t^{\mathcal{F}} \in I$. Case (1): $c \in \Sigma_{\overline{C}}$. We have $t_1{}^{\mathcal{F}}, \ldots, t_n{}^{\mathcal{F}} \in U$. So $(c(t_1, \ldots, t_n))^{\mathcal{F}} \in I$. Let $x \in I_i \backslash I_{i-1}$. Then $t_l{}^{\mathcal{F}} \in I_j \backslash I_{j-1}$, $j \geq i$, by Prop. 3.4.5. Hence $(t_1{}^{\mathcal{F}}, \ldots, t_n{}^{\mathcal{F}}) \in U_k^n \backslash U_{k-1}^n$ and $k \geq j \geq i$. So $t^{\mathcal{F}} \in U_{k+1} \backslash U_k$ and $k+1 > i$, by Lemma 3.4.3(i). Therefore $t^{\mathcal{F}} \neq x$. Case (2): $c \equiv scons$. So $t \equiv scons(t_1, t_2)$. If $t_2{}^{\mathcal{F}} \in S$ then $t^{\mathcal{F}} \in S$. Contradiction. So $t_2{}^{\mathcal{F}} \in I$. Again, similar to case(1), we get $t^{\mathcal{F}} \neq x$.

(F5): Let $x, \vec{z}$ be the free variables of s. So $x, y, \vec{z}$ are the free variables of $scons(s, y)$ with possibly $x \equiv y$. Let $x, y \in U, \vec{z} \in U^n$ be an assignment of these variables. Let $x \in U_i \backslash U_{i-1}$. Case (1): $y \in I$. Then $(indiv(scons(s, y)))^{\mathcal{F}}$ holds with $scons(s, y)$ containing x and different from x. So $(scons(s, y) \neq x)^{\mathcal{F}}$ by F4. Case (2): $y \in S$. Let $s^{\mathcal{F}} \in U_j \backslash U_{j-1}$. Then $j \geq i$ by Prop. 3.4.5. Suppose $y = \{\}$. Then $(scons(s, y))^{\mathcal{F}} = \{s^{\mathcal{F}}\} \in S_{j+1} \backslash S_j$, by Lemma 3.4.3(iii). So $j+1 > i$ and $(scons(s, y) \neq x)^{\mathcal{F}}$. Suppose $y \neq \{\}$, i.e., $y = \{x_1, \ldots, x_m\}, m \geq 1$, and $(x_1, \ldots, x_m) \in U_k^m \backslash U_{k-1}^m$. Let $l = \max(j, k)$. So $(scons(s, y))^{\mathcal{F}} = \{s^{\mathcal{F}}, x_1, \ldots, x_m\} \in S_{l+1} \backslash S_l$, by Lemma 3.4.3(iii). Hence $l + 1 > i$, so that $(scons(s, y) \neq x)^{\mathcal{F}}$. ■

3.5 Relating SetAx to ZF

We now justify $SetAx$ and $SetAx^-$ with respect to finite sets, by relating $SetAx^-$ to ZF. In order to compare the two systems, we consider the following modified subsystem of ZF, which we call $FinZF$. The language of ZF has the primitive symbols $\emptyset$, $\in$, and $=$, while other constructor and predicate symbols are neither assumed nor excluded. The symbol *set* is a defined one in ZF but equivalently may be taken as a primitive and its definition taken as an axiom. Note that the intended meaning of *set* in $SetAx$ and in ZF are different — in the former it represents a finite set, while in the latter a finite or infinite set.

We begin with ZF^- (i.e., ZF without the axiom of infinity), together with $set(x) \rightarrow finite(x)$, which we take as an adequate mathematical theory of finite sets (see [FBLD73] Chapter 2, §3.6). To this we add a definition of *scons*, viz., $set(y) \rightarrow \{x/y\} = \{x\} \cup y$. From the motivations underlying $SetAx$ we see the need to make explicit assumptions about constructor symbols. We take FS3 and FS4 to be those assumptions (FS4 is an assumption about *scons* not covered by its definition). Now let $FinZF$ be the totality of these axioms, which we take to be an adequate theory of finite sets for logic programming. We will show that $SetAx^-$ is equivalent to $FinZF$. The forward direction essentially uses induction. The reverse direction uses well-known properties of ZF.

We do not take $FinZF$ itself in place of $SetAx^-$ because the latter is more natural and simpler than the former. In $SetAx^-$, *scons* is a primitive symbol rather than a defined one, which is natural in logic programming where data constructors are primitive symbols and form the basis of a Herbrand domain. Also, almost all of the axioms in $SetAx^-$ are in definite clausal form, and instead of extensionality axiom of ZF forming the basis of the equational theory, we have the intuitive 'commutativity' and 'idempotency' properties of *scons*.

We list ZF^- below (except for the standard properties of reflexivity and substitutivity of the equality relation).

Set Definition ($SetDef$):

$$set(y) \leftrightarrow \exists x(x \in y) \vee y = \emptyset$$

Axiom of Extensionality ($ExtAx$):

$$set(y) \wedge set(z) \rightarrow (\forall x(x \in y \leftrightarrow x \in z) \rightarrow y = z)$$

Sum Axiom ($SumAx$):

$$set(y) \rightarrow \exists z(set(z) \wedge \forall x(x \in z \leftrightarrow \exists u(set(u) \wedge x \in u \wedge u \in y)))$$

Powerset Axiom ($PowerAx$):

$$set(y) \rightarrow \exists z(set(z) \wedge \forall x(set(x) \rightarrow (x \in z \leftrightarrow x \subseteq y)))$$

Axiom of Regularity ($RegAx$):

$$set(z) \wedge z \neq \emptyset \rightarrow \exists x(x \in z \wedge \forall y(y \in x \rightarrow y \notin z))$$

Axiom Schema of Replacement ($ReplAx$):

$$set(u) \rightarrow \big(\forall x \forall y \forall z(x \in u \wedge \varphi(x, y, \vec{w}) \wedge \varphi(x, z, \vec{w}) \rightarrow y = z)$$
$$\rightarrow \exists v(set(v) \wedge \forall y(y \in v \leftrightarrow \exists x(x \in u \wedge \varphi(x, y, \vec{w}))))\big)$$

Axiom of Choice ($ChoiceAx$):

For any set y there is a function f such that for any non-empty subset z of y, $f(z) \in z$.

The following show that $SetAx^-$ can deduce all the axioms of ZF^-.

Theorem 3.5.1

(i) $SetAx^- \models SetDef$

(ii) $SetAx^- \models ExtAx$

(iii) $SetAx^- \models SumAx$

(iv) $SetAx^- \models PowerAx$

(v) $SetAx^- \models RegAx$

(vi) $SetAx^- \models ReplAx$

Proof: We give a sketch of the proofs.

(i) $SetDef$: Trivial. ($\rightarrow$) By Prop. 3.3.2(ii). ($\leftarrow$) By Prop. 3.3.1(i) and FS1.

(ii) $ExtAx$: Assume $set(y)$ and use induction axiom with

$$\psi(y) \equiv \forall z(set(z) \wedge \forall x(x \in y \leftrightarrow x \in z) \rightarrow y = z)$$

Basis step: $\psi(\emptyset) \equiv \forall z(set(z) \wedge \forall x(x \in \emptyset \leftrightarrow x \in z) \rightarrow \emptyset = z)$. Assume $set(z) \wedge \forall x(x \in \emptyset \leftrightarrow x \in z)$. Hence $\exists x(x \in z) \vee z = \emptyset$, by $SetDef$. Suppose $\exists x(x \in z)$. Then $x \in z$, and from $x \in \emptyset \leftrightarrow x \in z$ we get $x \in \emptyset$. Contradiction to M1. Therefore $\neg\exists x(x \in z)$, i.e., $z = \emptyset$.

Induction step: We show that $\forall x', y'(set(y'), x' \in y, y' \subseteq y, \psi(y') \rightarrow \psi(\{x'/y'\}))$. Assume $set(y')$, $x' \in y$, $y' \subseteq y$ and $\psi(y')$. We may take $x' \notin y'$, by Prop. 3.3.3(ii). Assume $set(z) \wedge \forall x(x \in$

$\{x'/y'\} \leftrightarrow x \in z)$, the antecedent in $\psi(\{x'/y'\})$. By $SetDef$, $z = \emptyset \vee \exists x(x \in z)$. Suppose $z = \emptyset$. Then $x' \in \{x'/y'\} \leftrightarrow x' \in \emptyset$. So $x' \in \emptyset$ by Prop. 3.3.1(iii), which contradicts M1. Hence $z \neq \emptyset$, i.e., $\exists x(x \in z)$. Let $z = \{x''/y''\} \wedge set(y'') \wedge x'' \notin y''$, by Prop. 3.3.2(i). Therefore $\forall x(x \in \{x'/y'\} \leftrightarrow x \in \{x''/y''\})$, i.e., $\forall x(x = x' \vee x \in y' \leftrightarrow x = x'' \vee x \in y'')$ by M3.

Case (1): $x' = x''$. We show $\forall x(x \in y' \leftrightarrow x \in y'')$. If $x \neq x'$, then $x \neq x''$ and so $x \in y' \leftrightarrow x \in y''$. If $x = x'$, then $x = x''$ and so, from $x' \notin y'$ and $x'' \notin y''$, we get $x' \notin y' \leftrightarrow x'' \notin y''$, i.e., $x \in y' \leftrightarrow x \in y''$. Hence $\forall x(x \in y' \leftrightarrow x \in y'')$. From the induction hypothesis $\psi(y')$ we get $y' = y''$. Thus $\{x'/y'\} = \{x''/y''\}$ by $EqAx$. Hence $\{x'/y'\} = z$.

Case (2): $x' \neq x''$. Then $x' \in y''$ and $x'' \in y'$. Let $y'' = \{x'/z''\} \wedge set(z'') \wedge x' \notin z''$ and $y' = \{x''/z'\} \wedge set(z') \wedge x'' \notin z'$, by Prop. 3.3.2(i). Note that $x' \notin \{x''/z''\}$ as $x' \neq x''$ and $x' \notin z''$. We show $\forall x(x \in y' \leftrightarrow x \in \{x''/z''\})$. Now $\forall x(x = x' \vee x \in y' \leftrightarrow x = x'' \vee x = x' \vee x \in z'')$ by M3. So $\forall x(x = x' \vee x \in y' \leftrightarrow x = x' \vee x \in \{x''/z''\})$. If $x \neq x'$, then $x \in y' \leftrightarrow x \in \{x''/z''\}$. If $x = x'$, then from $x' \notin y'$ and $x' \notin \{x''/z''\}$ we get $x' \notin y' \leftrightarrow x' \notin \{x''/z''\}$, i.e., $x \in y' \leftrightarrow x \in \{x''/z''\}$. Hence $\forall x(x \in y' \leftrightarrow x \in \{x''/z''\})$. From the induction hypothesis $\psi(y')$, we get $y' = \{x''/z''\}$. Then $\{x'/y'\} = \{x'/\{x''/z''\}\} = \{x''/\{x'/z''\}\} = \{x''/y''\} = z$ by CI1.

So, in either case, we obtain $\{x'/y'\} = z$, and hence $\psi(\{x'/y'\})$. Therefore $\psi(y)$.

(iii) $SumAx$: Assume $set(y)$ and use induction axiom with

$$\psi(y) \equiv \exists z(set(z) \wedge \forall x(x \in z \leftrightarrow \exists u(set(u) \wedge x \in u \wedge u \in y)))$$

At the induction step, an auxiliary induction is required, to show that a certain set, that cannot be expressed purely as a scons term, exists. We give brief details of the overall proof.

Basis step: It is easy to show $\psi(\emptyset)$. Take $\emptyset$ as witness for z.

Induction step: To show $\forall x', y'(set(y'), x' \in y, y' \subseteq y, \psi(y') \rightarrow \psi(\{x'/y'\}))$. Assume $set(y')$, $x' \in y$, $y' \subseteq y$ and $\psi(y')$. Case(1): $indiv(x')$. (Since x' has no elements, a witness for z in $\psi(y')$ is a witness for z in $\psi(\{x'/y'\})$.) Easy to show that $\psi(\{x'/y'\}) \leftrightarrow \psi(y')$.

Case(2): $set(x')$. (Intuition: Let z_1 be a witness for z in $\psi(y')$, i.e., $z_1 = \bigcup y'$. Then, want $\bigcup\{x'/y'\} = x' \cup (\bigcup y') = x' \cup z_1$ to be a witness for z in $\psi(\{x'/y'\})$. But, we cannot express $x' \cup z_1$ as a scons term since actual elements of x' are not known. So, we do an auxiliary induction on x'. Note that the symbols $\bigcup$ and $\cup$ have not yet been introduced in $SetAx$, and are to be understood here in an intuitive sense.)

We show $induction(x', y')$ holds, where $\rho(x', y') \equiv \psi(\{x'/y'\})$, and $induction(x', y') \equiv$

$$\rho(\emptyset, y'), \forall x'', y''(set(y''), x'' \in x', y'' \subseteq x', \rho(y'', y') \rightarrow \rho(\{x''/y''\}, y')) \rightarrow \rho(x', y')$$

The auxiliary basis step $\rho(\emptyset, y') \equiv \psi(\{\emptyset/y'\})$ is established in the same way as for case (1) since $\emptyset$ has no elements. The auxiliary induction step is established by using $\{x''/z_2\}$ as a witness for z in $\rho(\{x''/y''\}, y')$, where z_2 is a witness for z in $\rho(y'', y')$.

Thus we have $\rho(x'/y')$, i.e., $\psi(\{x'/y'\})$. Hence $\psi(y)$.

(iv) $PowerAx$: Assume $set(y)$ and use induction axiom with

$$\psi(y) \equiv \exists z(set(z) \wedge \forall x(set(x) \rightarrow (x \in z \leftrightarrow x \subseteq y)))$$

Again, at the induction step, an auxiliary induction is required. We give brief details of the overall proof.

Basis step: Straightforward to show $\psi(\emptyset)$. Take $\{\emptyset/\emptyset\}$ as witness for z.

Induction step: To show $\forall x', y'(set(y'), x' \in y, y' \subseteq y, \psi(y') \rightarrow \psi(\{x'/y'\}))$. Assume $set(y')$, $x' \in y$, $y' \subseteq y$ and $\psi(y')$. (Intuition: Let z_1 be a witness for z in $\psi(y')$, i.e., $z_1 = \mathcal{P}(y')$. Then, want $\mathcal{P}(\{x'/y'\}) = z_1 \cup \{\{x'/u\} \mid set(u) \wedge u \in z_1\}$ to be a witness for z in $\psi(\{x'/y'\})$. But, we cannot express this witness as a scons term since actual elements of z_1 are not known. So, we do an auxiliary induction on z_1. Again, note that the symbols $\mathcal{P}$, $\{- \mid \ldots\}$, etc. are to be understood in an intuitive sense.)

We show $induction(z_1, x')$, where $\rho(z_1, x') \equiv$

$$\exists z(set(z) \wedge \forall x(x \in z \leftrightarrow x \in z_1 \vee \exists u(set(u) \wedge x = \{x'/u\} \wedge u \in z_1)))$$

and $induction(z_1, x') \equiv$

$$\rho(\emptyset, x'), \forall x'', y''(set(y''), x'' \in z_1, y'' \subseteq z_1, \rho(y'', x') \to \rho(\{x''/y''\}, x')) \to \rho(z_1, x')$$

Here, $\rho(z_1, x')$ expresses the existence of the required witness in $\psi(\{x'/y'\})$, where z_1 is the witness for z in $\psi(y')$.

The auxiliary basis step $\rho(\emptyset, x')$ is established with $\emptyset$ as witness for z. In the auxiliary induction step, suppose that z_2 is a witness for z in $\rho(y'', x')$. Then, as witness for z in $\rho(\{x''/y''\}, x')$, use $\{x''/z_2\}$ when $indiv(x'')$ holds and use $\{\{x'/x''\}/\{x''/z_2\}\}$ when $set(x'')$ holds.

Thus we have $\rho(z_1, x')$. We then show that a witness for z in $\rho(z_1, x')$ is also a witness for z in $\psi(\{x'/y'\})$. Hence $\psi(y)$.

(v) *RegAx*: Trivially.

(vi) *ReplAx*: Assume $set(u)$ and use induction axiom with

$$\psi(u, \vec{w}) \equiv \forall x \forall y \forall z(x \in u \wedge \varphi(x, y, \vec{w}) \wedge \varphi(x, z, \vec{w}) \to y = z)$$
$$\to \exists v(set(v) \wedge \forall y(y \in v \leftrightarrow \exists x(x \in u \wedge \varphi(x, y, \vec{w}))))$$

Basis step: In $\psi(\emptyset, \vec{w})$, the antecedent holds vacuously while $\emptyset$ is a witness for v in the consequent.

Induction step: To show $\forall x', y'(set(y'), x' \in u, y' \subseteq u, \psi(y', \vec{w}) \to \psi(\{x'/y'\}, \vec{w}))$. Assume $set(y')$, $x' \in u$, $y' \subseteq u$ and $\psi(y', \vec{w})$. The antecedent of $\psi(\{x'/y'\}, \vec{w})$ expresses the antecedent of $\psi(y', \vec{w})$ as well as that $\phi(x', y, \vec{w})$ is functional in its second argument when $\exists y \phi(x', y, \vec{w})$. Let v_1 be a witness for v in the consequent of the induction hypothesis $\psi(y', \vec{w})$. If $\neg \exists y \phi(x', y, \vec{w})$ then v_1 is a witness for v in the consequent of $\psi(\{x'/y'\}, \vec{w})$. If $\exists y \phi(x', y, \vec{w})$ and y_1 is witness to it , then $\{y_1/v_1\}$ is a witness for v in the consequent of $\psi(\{x'/y'\}, \vec{w})$.

Thus we have $\psi(\{x'/y'\}, \vec{w})$ and hence $\psi(u, \vec{w})$. ■

We need to develop the theory further before deducing the remaining axiom *ChoiceAx* of ZF^- from $SetAx^-$. But we can now directly borrow in $SetAx^-$ all the properties of the axioms of

ZF^- deduced in above theorem. The next proposition is useful in subsequent proofs.

Proposition 3.5.2 $SetAx^- \models \varphi$, where φ is any of:

(i) $set(y) \wedge set(z) \rightarrow (y = z \rightarrow \forall x(x \in y \leftrightarrow x \in z))$

(ii) $set(y) \wedge set(z) \rightarrow (\{w/y\} = \{w/z\} \wedge w \notin y \wedge w \notin z \rightarrow y = z)$

Proof: (i): Trivial (from predicate substitutivity of equality axioms). This is just the converse of extensionality $ExtAx$.

(ii): Straightforward. Assume $set(y)$, $set(z)$ and that $\{w/y\} = \{w/z\}$, $w \notin y$, $w \notin z$. From (i), $\forall x(x \in \{w/y\} \leftrightarrow x \in \{w/z\})$, i.e., $\forall x(x = w \vee x \in y \leftrightarrow x = w \vee x \in z)$. Hence, when $x \neq w$, we get $x \in y \leftrightarrow x \in z$. When $x = w$, from $x \notin y$, $x \notin z$ we get $x \notin y \leftrightarrow x \notin z$, i.e., $x \in y \leftrightarrow x \in z$. Therefore, $\forall x(x \in y \leftrightarrow x \in z)$ in either case. Thus $y = z$, from $ExtAx$. ■

In $SetAx$ and $SetAx^-$, we adopt all the defined symbols of ZF associated with ZF^-, except set, which is taken as primitive in $SetAx$. These adopted symbols are given the same definitions in $SetAx$ as in ZF. That the definitions are well-defined in $SetAx$ follows from their well-definedness in ZF^- and from Theorem 3.5.1. We list some of these definitions below.

(D4): $$\big(\{x \mid \varphi(x, \vec{u})\} = y \leftrightarrow set(y) \wedge \forall x(x \in y \leftrightarrow \varphi(x, \vec{u}))\big)$$
$$\leftarrow \exists y\big(set(y) \wedge \forall x(x \in y \leftrightarrow \varphi(x, \vec{u}))\big)$$

Here, $\vec{u}$ is the other free variables in φ. The antecedent expresses the condition of interest, viz., that the intuitively appropriate collection specified by the set abstraction does form a (finite) set. When the set abstraction forms too large a collection to exist as a (finite) set, i.e., when the negation of the antecedent holds, we take $y = \emptyset$ as the don't care definition. Some of the definitions below are specified using set abstraction. In each case, the intuitively appropriate collection does exist as a set, because it is known to do so in ZF^-.

(D5): $(y \supseteq z \leftrightarrow z \subseteq y) \leftarrow set(y) \wedge set(z)$

(D6): $(y \cap z = \{x \mid x \in y \wedge x \in z\}) \leftarrow set(y) \wedge set(z)$

(D7): $(y \cup z = \{x \mid x \in y \vee x \in z\}) \leftarrow set(y) \wedge set(z)$

(D8): $(y \backslash z = \{x \mid x \in y \wedge x \notin z\}) \leftarrow set(y) \wedge set(z)$

(D9): $(\mathcal{P}(y) = \{x \mid set(x) \wedge x \subseteq y\}) \leftarrow set(y)$

(D10): $\bigcup y = \{x \mid \exists z(x \in z \wedge z \in y)\} \leftarrow set(y)$

(D11): $\{y, z\} = w \leftrightarrow set(w) \wedge \forall x(x \in w \leftrightarrow x = y \vee x = z)$

(D12): $\{x\} = \{x, x\}$,

$\{x, y, z\} = \{x, y\} \cup \{z\}$,

$\{x, y, z, w\} = \{x, y\} \cup \{z, w\}$, and so on.

(D13): $\cup_{i=1}^{n} x_i = \bigcup\{x_1, \ldots, x_n\}$

(D14): $\langle x, y\rangle = \{\{x\}, \{x, y\}\}$

(D15): $(xzy \leftrightarrow \langle x, y\rangle \in z) \leftarrow set(z)$

(D16): $\big(relation(u) \leftrightarrow \forall x(x \in u \rightarrow \exists y \exists z(x = \langle y, z\rangle))\big) \leftarrow set(u)$

(D17): $(function(f) \leftrightarrow relation(f) \wedge \forall x \forall y \forall z(xfy \wedge xfz \rightarrow y = z)) \leftarrow set(f)$

(D18): $(f(x) = y \leftrightarrow (\exists! z(xfz) \wedge xfy) \vee (\neg\exists! z(xfz) \wedge y = \emptyset)) \leftarrow set(f)$

Using the above definitions a formal statement of the Axiom of Choice can be given, and it can be deduced using induction, as is well-known in set theory.

Axiom of Choice ($ChoiceAx$):

$set(z) \rightarrow \exists f(set(f) \wedge function(f) \wedge \forall x(set(x) \wedge x \subseteq z \wedge x \neq \emptyset \rightarrow f(x) \in x))$

Theorem 3.5.3 $SetAx^- \models ChoiceAx$

Proof: Assume $set(z)$ and use induction axiom with

$$\psi(z) \equiv \exists f(set(f) \wedge function(f) \wedge \forall x(set(x) \wedge x \subseteq z \wedge x \neq \emptyset \rightarrow f(x) \in x))$$

We sketch the overall proof.

Basis step: Use $\emptyset$ as witness for f in $\psi(\emptyset)$.

Induction step: To show $\forall x', y'(set(y'), x' \in z, y' \subseteq z, \psi(y') \rightarrow \psi(\{x'/y'\}))$. Assume $set(y')$, $x' \in z$, $y' \subseteq z$ and $\psi(y')$. Let f_0 be a witness for f in $\psi(y')$. Then show $f_1 = f_0 \cup \{\langle x, x' \rangle \mid x \subseteq \{x'/y'\}, x' \in x\}$ is a witness for f in $\psi(\{x'/y'\})$. Using the already deduced axioms of ZF^- and their properties, it is straightforward to establish that f_1 exists as a set, is a function, and that it is a witness for f in $\psi(\{x'/y'\})$. ■

The following definition of minimality is used to give the definition of finiteness in sets. It states that a set x is minimal amongst a collection of sets y if it is minimal with respect to the order relation $\subseteq$. For example, if $y = \{\{1,2\}, \{1\}, \{3\}\}$, then $x = \{1\}$ is a minimal element in y, as is $\{3\}$, i.e., $minimal(\{1\}, y)$ and $minimal(\{3\}, y)$ hold. We follow Tarski's definition of finiteness which states that a set z is finite exactly when every non-empty family of subsets of z has a minimal element.

(D19): $(minimal(x, y) \leftrightarrow x \in y, set(x), \forall z(set(z), z \in y, z \subseteq x \rightarrow z = x)) \leftarrow set(y)$

(D20): $\big(finite(z) \leftrightarrow \forall y(set(y), y \neq \emptyset, y \subseteq \mathcal{P}(z) \rightarrow \exists x\, minimal(x, y))\big) \leftarrow set(z)$

Some properties based on the above definitions are given next. They establish that *scons* does indeed have its intended meaning, and that *set* in $SetAx^-$ does refer to finite sets. The more usual definition of finiteness of a set, viz., being equinumerous to a natural number, is known to be equivalent, under ZF^-, to Tarski's definition (see [Sup72] §4.2, §5.2 and [Rub73], where the equivalence is shown apparently under ZF, but the relevant proofs can be worked out without the use of the Axiom of Infinity). Therefore the two definitions of finiteness are now equivalent under $SetAx^-$.

Theorem 3.5.4 $SetAx^- \models \varphi$, where φ is any of:

(i) $set(y) \rightarrow \{x/y\} = \{x\} \cup y$

(ii) $\{x_1/\{x_2/ \cdots \{x_n/\emptyset\} \cdots\} = \{x_1, x_2, \ldots, x_n\}$ for any $n \geq 1$

(iii) $set(z) \rightarrow finite(z)$

Proof: (i): Assume $set(y)$. So $set(\{x/y\})$ by FS2. Also $set(\{x\} \cup y)$ by D12, D11, D7 & D4. Now we show $\forall z(z \in \{x/y\} \leftrightarrow z \in \{x\} \cup y)$, and apply $ExtAx$. We have $z \in \{x/y\} \leftrightarrow z = x \vee z \in y \leftrightarrow z \in \{x\} \vee z \in y \leftrightarrow z \in \{x\} \cup y$, by M3 and from properties of ZF^-, viz., Theorems 43 & 21, Chapter 2 [Sup72].

(ii): By induction on n. Basis: $n = 1$. By (i) above, $\{x_1/\emptyset\} = \{x_1\} \cup \emptyset = \{x_1\}$. Induction step: $n \geq 2$. We have $\{x_1/\{x_2/\cdots\{x_n/\emptyset\}\cdots\} = \{x_1\} \cup \{x_2/\cdots\{x_n/\emptyset\}\cdots\} = \{x_1\} \cup \{x_2, \ldots, x_n\} = \{x_1, \ldots, x_n\}$, by induction hypothesis and from D11 & D12.

(iii): Assume $set(z)$ and use induction axiom with

$$\psi(z) \equiv finite(z) \equiv \forall y(set(y), y \neq \emptyset, y \subseteq \mathcal{P}(z) \rightarrow \exists x\, minimal(x, y))$$

We give brief details of the proof.

Basis step: In $\psi(\emptyset)$ we have $\mathcal{P}(\emptyset) = \{\emptyset\}$, and the only non-empty subset y of $\mathcal{P}(\emptyset)$ is $y = \{\emptyset\}$. Clearly $\emptyset$ is minimal in y, i.e., $minimal(\emptyset, y)$.

Induction step: To show $\forall x', y'(set(y'), x' \in z, y' \subseteq z, \psi(y') \rightarrow \psi(\{x'/y'\}))$. Assume $set(y')$, $x' \in z$, $y' \subseteq z$ and $\psi(y')$. Also, we can assume that $x' \notin y'$. In $\psi(y')$, we have that non-empty subsets y of $\mathcal{P}(y')$ have minimal elements. In $\psi(\{x'/y'\})$, we have to show that non-empty subsets y of $\mathcal{P}(\{x'/y'\})$ have minimal elements. Let $y \subseteq \mathcal{P}(\{x'/y'\})$ and $y \neq \emptyset$. We have $\mathcal{P}(y') \subseteq \mathcal{P}(\{x'/y'\})$. Let $u = y \cap \mathcal{P}(y')$. Now $u \subseteq \mathcal{P}(y')$, and if $u \neq \emptyset$, then from $\psi(y')$ we get that u has a minimal element x. Then it is easy to see that x is a minimal element of y.

If $u = \emptyset$ then every element of y contains x'. Now let $v = \{w \mid \{x'/w\} \in y \wedge x' \notin w\}$. We have $v \subseteq \mathcal{P}(y')$ and $v \neq \emptyset$ (since $y \neq \emptyset$). From $\psi(y')$ we get that v has a minimal element x. Then it is easy to see that $\{x'/x\}$ is a minimal element of y. Thus $\psi(\{x'/y'\})$ and hence $\psi(z)$. ■

We have not gone into the issue of the independence of each of the axioms of $SetAx$ since it is inessential to this work. We note that axiom FS3 can be deduced from the remaining axioms (from F1 and Prop. 3.3.3(i)).

Finally, we have the reverse implication.

Theorem 3.5.5 $FinZF \models SetAx^-$

Proof: Straightforward, using the properties of ZF^-, such as in Chapters 2 and 4 of [Sup72].

$FinSetAx$: (FS1): $set(\emptyset)$ holds from $SetDef$. (FS2): Assume $set(y)$. Hence $set(\{x/y\})$, since $\{x/y\} = \{x\} \cup y$ from definition of $scons$. (FS3) and (FS4): Trivially from $FinZF$. (FS5): We have $set(z) \wedge finite(z) \rightarrow induction(z, \vec{w})$, by Theorem 32, Chapter 4 [Sup72]. Hence $set(z) \rightarrow induction(z, \vec{w})$ since $set(z) \rightarrow finite(z)$ is in $FinZF$.

$MemAx$: (M1): $x \notin \emptyset$, by Theorem 1, Chapter 2 [Sup72]. (M2): We show the contrapositive. Assume $x \in y$. So $\exists z(x \in y)$. Hence $set(y)$, from $SetDef$. (M3): Assume $set(z)$. Now $x = y \vee x \in z \leftrightarrow x \in \{y\} \vee x \in z \leftrightarrow x \in \{y\} \cup z \leftrightarrow x \in \{y/z\}$, by Theorems 43 & 20, Chapter 2 [Sup72], and by definition of $scons$. (M4): Trivially from $FinZF$.

$ColdAx$: (CI1): Assume $set(z)$. So $set(\{y\} \cup z)$ and $set(\{x\} \cup z)$, i.e., $set(\{y/z\})$ and $set(\{x/z\})$. From definition of $scons$, and from Theorems 22 & 21, Chapter 2 [Sup72], we get, $\{x/\{y/z\}\} = \{x\} \cup \{y/z\} = \{x\} \cup \{y\} \cup z = \{y\} \cup \{x\} \cup z = \{y\} \cup \{x/z\} = \{y/\{x/z\}\}$. (CI2): Assume $set(z)$. So $set(\{x\} \cup z)$, i.e., $set(\{x/z\})$. From definition of $scons$, and from Theorems 22 & 23, Chapter 2 [Sup72], we get, $\{x/\{x/z\}\} = \{x\} \cup \{x/z\} = \{x\} \cup \{x\} \cup z = \{x\} \cup z = \{x/z\}$.

$EqAx$: Trivially from $FinZF$. ■

We find that $SetAx^-$ has essentially been described in [MW85], but our development was done without awareness of it. However, only §3.3 of this chapter overlaps with that work (except for the deduction of extensionality from $SetAx$). Also, their work showed that one of the axioms in an earlier version of $SetAx$ was redundant.

It remains to justify the freeness axioms of $SetAx$ as an adequate system to enforce inequalities among objects that are not provably equal in $SetAx^-$. This is done in the next chapter. We note that certain inequalities can be deduced in $SetAx^-$ itself, such as $x \neq y \leftarrow set(x) \wedge indiv(y)$, and $(y \neq z \leftarrow \neg\forall x(x \in y \leftrightarrow x \in z)) \leftarrow set(y) \wedge set(z)$. (The latter is just the contrapositive of Prop. 3.5.2(i).) Also, M4 helps to deduce inequalities like $x \neq \{x\}$ and $y \neq \{\{y\}\}$.

4 Set Constraints and Unification

In this chapter, we state the unification problem in $SetAx$ in the broader context of solving set predicate constraints such as equality and membership constraints. We give rewriting algorithms for solving them and deduce relevant properties of $SetAx$ and $SetAx^-$ from them. Using the properties we justify the freeness axioms of $SetAx$. We also take the opportunity to address the usual concerns of unification and matching such as obtaining correct, complete, and minimal sets of unifiers.

4.1 Constraints and Solved Forms

We first informally motivate the unification process in $SetAx$ through an example before developing the definitional aspects leading up to it. Suppose we want to unify $s = t$ for $s \equiv c(x, z, \{2/y\})$ and $t \equiv c(\{1/x\}, 1, \{2/\emptyset\})$, $c \in \Sigma_{\overline{C}}$. Solving this unification problem amounts to giving a sequence of rewriting steps, with each step seeking to rewrite the previous problem into an equivalent but simpler problem. Each step is represented as a multiset of atomic relations, which is understood as a conjunction of these atoms. The multiset may have zero or more existential variables before it. Each step of the rewriting involves applying a rewrite rule to a selected atom (indicated by an underline) and replacing the atom with the rhs of the rewrite rule. The rewrite rule is chosen from a fixed collection of rules that together with the control on applying them constitutes the unification algorithm.

Finding all possible unifiers to a unification problem amounts to setting up a tree of all possible rewritings. Such a tree is given below for our chosen $s = t$.

(1) $$\{\underline{c(x, z, \{2/y\}) = c(\{1/x\}, 1, \{2/\emptyset\})}\}$$

$$\Downarrow$$

$$(2) \qquad \{\underline{x = \{1/x\}}, z = 1, \{2/y\} = \{2/\emptyset\}\}$$

$$\Downarrow$$

$$(3) \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, \underline{\{2/y\} = \{2/\emptyset\}}\}$$

$$\swarrow \qquad \searrow$$

$$(4) \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, \underline{2 = 2}, y = \emptyset\} \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, \underline{y = \{2/\emptyset\}}, 2 \in y\}$$

$$\Downarrow \qquad \Downarrow$$

$$(5) \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \emptyset\} \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \{2/\emptyset\}, \underline{2 \in \{2/\emptyset\}}\}$$

$$\swarrow \qquad \searrow$$

$$(6) \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \{2/\emptyset\}, \underline{2 = 2}\} \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \{2/\emptyset\}, \underline{2 \in \emptyset}\}$$

$$\Downarrow \qquad \Downarrow$$

$$(7) \qquad \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \{2/\emptyset\}\} \qquad F$$

The nodes of the tree have multiple branches according to the number of rewrite rules that apply to the selected atom at the node. The rewrite rules are framed to be such that each parent node is equivalent to the disjunction of its children nodes. We take equivalence to be logical equivalence. For example, in going from (2) to (3), we have applied the rule

$$K \cup \{x = \{1/x\}\} \Rightarrow \exists z_1 K \cup \{x = \{1/z_1\}, set(z_1)\}$$

where K denotes the rest of the unification problem. This rule arises from the equivalence

$$SetAx \models \forall(x = \{1/x\} \leftrightarrow \exists z_1(x = \{1/z_1\} \wedge set(z_1)))$$

Similarly, in going from (3) to (4) the rules applied were

$$K \cup \{\{x/y\} = \{x/\emptyset\}\} \Rightarrow K \cup \{x = x, y = \emptyset\}$$

$$K \cup \{\{x/y\} = \{x/\emptyset\}\} \Rightarrow K \cup \{y = \{x/\emptyset\}, x \in y\}$$

based on the equivalence

$$SetAx \models \forall(\{x/y\} = \{x/\emptyset\} \leftrightarrow (x = x \wedge y = \emptyset) \vee (y = \{x/\emptyset\} \wedge x \in y)).$$

(This is a 'toy' equivalence used for illustrative purposes.)

The leaves of the tree are are either F, denoting failure of unification of its parent node, or are multisets of atoms in a form called *solved forms*, denoting unifiers to the unification problem at the root of the tree. The solved forms are nodes at which no rewrite rule applies to any of its atoms. The solved forms in the example are at line 5

$$\zeta_1 \equiv \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \emptyset\}$$

and at line 7

$$\zeta_2 \equiv \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \{2/\emptyset\}\}$$

The aim of the unification process is to transform a given unification problem step by step that ultimately ends in simplified forms called solved forms. Solved forms have the desirable property that all solutions can be easily read off from them. Thus the collection of solved forms at the leaves of the rewrite tree expresses *explicitly* the same set of solutions as was *implicit* in the unification problem at the root of the tree. The particular solved forms obtained in a rewrite tree correspond to substitutions that are unifiers of the starting unification problem.

For example, in the solved form $\zeta_1 \equiv \exists z_1\{x = \{1/z_1\}, set(z_1), z = 1, y = \emptyset\}$, all its solutions can be read off by giving all possible set values for z_1, and thereby obtaining the values of x, z, and y. Also, ζ_1 is a unifier of $s = t$ since it leads to the substitution $\theta = \{x \mapsto \{1/z_1\}, z \mapsto 1, y \mapsto \emptyset\}$ for which $\theta s = \theta t$ holds in $SetAx$ (provided we consider z_1 to be a set variable, i.e., can take only instantiations that are sets). An equivalent way of knowing that ζ_1 is a unifier of $s = t$ is by asserting that $SetAx \models \forall(\zeta_1 \rightarrow s = t)$.

When there are no branch points in the tree, i.e., only one rule is applicable at each node, the solved form, if any, is said to be obtained through a *deterministic* rewriting. When there are branch points, then solved forms are said to be obtained through *non-deterministic* rewritings. Thus, in the above tree, the solved form ζ_1 is obtained non-deterministically through the sequence of rewritings that occur along the path from the root to this leaf.

Some notable features in the above example, different from the usual case of 'syntactic' unification, are as follows. In solving the equality predicate, we see that other set predicates such as *set* and $\in$ may enter into the picture. The number of unifiers to a unification problem can be seen to be in general more than one, none of which is redundant. Also, new variables are introduced in the unification process whose roles are that of existential variables.

Below we consider unification as being generalised to solving *constraints*, where a constraint is roughly a conjunction of atoms involving the set predicates *set*, $=$, and $\in$. Henceforth, we will use the term unification in this more general sense. We now give various definitions relevant to constraints, solved forms and substitutions.

Definition 4.1.1 A **constraint** χ is a conjunction of the form $\exists\vec{z}(A_1 \wedge \cdots \wedge A_n)$ where $n \geq 0$ and each A_i is of the form $set(s)$, $s = t$ or $s \in t$. The $\vec{z}$ are the bound variables of χ.

The free variables of χ are denoted by $Var(\chi)$. It is convenient to abbreviate a constraint as an existential multiset $K = \exists\vec{z}\{A_i\}_{i=1}^{n}$ and let $Var(K)$ be its free variables.

Definition 4.1.2 An object such as a term, atom, literal, etc. is called **ground** if it contains no variables.

Definition 4.1.3 A **matching constraint** is a constraint in which there are no existential variables and only atoms of the form $s = t$ or $s \in t$ are present, with t being a ground term.

A *unification problem* varies according to the kind of constraint we want to solve in a given situation. For example, we may want to solve arbitrary constraints or only matching constraints or constraints with no variables. Accordingly, a unification problem will be defined in that situation to be a constraint or matching constraint or a constraint with no variables. Similarly, the syntactic form of a *solved form* that denotes the solutions to a unification problem can vary with the situation. Hence, depending on the context we are interested in, we will fix our definition of a unification problem and solved forms. In the current context, these definitions are as below.

Definition 4.1.4 A **unification problem** is a constraint.

Definition 4.1.5 A **solved form** is a constraint of the form $\exists\vec{z}\{set(y_1), \cdots, set(y_m), x_1 = t_1, \cdots, x_n = t_n\}$ where $m, n \geq 0$, and the x_i's are distinct free variables that occur exactly once. Also, each existential variable occurs at least once in the multiset, and if any y_j is an existential variable, then it occurs among the t_i.

Definition 4.1.6 Let $\zeta \equiv \exists\vec{z}\{set(y_1), \cdots, set(y_m), x_1 = t_1, \cdots, x_n = t_n)$ where $m, n \geq 0$. The y_j are called **set variables** of ζ. The **domain** of ζ, called $Dom(\zeta)$, is $Dom(\zeta) = \{x_1, \ldots, x_n\}$. The **range** of ζ, called $Ran(\zeta)$, is $Ran(\zeta) = \{set(y) \mid set(y) \text{ occurs in } \zeta\} \cup \{t \mid x = t \text{ occurs in } \zeta\}$. The variables in $Ran(\zeta)$ are called $VRan(\zeta)$.

Clearly, for a solved form ζ, $Dom(\zeta) \cap VRan(\zeta) = \emptyset$. The solutions provided by ζ can be obtained by choosing arbitrary values for the range variables $VRan(\zeta)$ (with the values being set values for the set variables). The values of the domain variables are obtained from the equations of ζ. After this, the existential variables can be omitted or forgotten, leaving only the values of the free variables.

Definition 4.1.7 Let $TERM$ be the set of terms. A **substitution** is a mapping $\theta\colon \Sigma_V \rightarrow TERM$ such that it is an identity mapping almost everywhere, i.e., for all but a finite number of variables. It is represented as a finite set of pairs $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $x_i \not\equiv t_i$ for all i, $1 \leq i \leq n$, and all the x_i's are distinct. The identity mapping, i.e., the **empty substitution**, is denoted by ϵ.

Definition 4.1.8 The **domain** of a substitution θ, called $Dom(\theta)$, is $Dom(\theta) = \{x \mid x \text{ is a variable}$

and $\theta(x) \not\equiv x\}$. The **range** of θ, called $Ran(\theta)$, is $Ran(\theta) = \{\theta(x) \mid x \in Dom(\theta)\}$. The variables in $Ran(\theta)$ are called $VRan(\theta)$.

Typical symbols for constraints are χ, χ_0, χ', ..., for solved forms are ζ, ξ, ζ_0, ξ', ..., and for substitutions are θ, σ, γ, θ_0, σ', ...

Substitutions can be applied to objects other than variables by extending them by natural morphisms to sets of terms, atoms, clauses, and so on. Hereon, we will usually omit parentheses in applying a substitution to an object, such as in θx. Substitutions, being maps, can be composed, and we have $(\sigma \circ \theta)(e) = \sigma(\theta(e))$, and ϵ as the identity of composition.

Definition 4.1.9 A substitution θ is a **ground substitution** if $VRan(\theta) = \emptyset$.

Definition 4.1.10 A substitution θ is **idempotent** if $\theta = \theta \circ \theta$.

Definition 4.1.11 A substitution $\theta = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ is a **renaming substitution** for an object X such as a term or atom if $Dom(\theta) \subseteq Var(X)$, the y_i are distinct, and $(Var(X) \backslash Dom(\theta)) \cap VRan(\theta) = \emptyset$.

Definition 4.1.12 Let X and Y be objects such as terms or atoms. X and Y are **variants** if there exists renaming susbtitutions σ and θ such that $X = \sigma Y$ and $Y = \theta X$. We also say that X **is a variant of** Y and Y **is a variant of** X.

If X and Y are variants, $X = \sigma Y$, $Dom(\sigma) = Var(Y)$, and the variables in $VRan(\sigma)$ 'have not occurred before' in a context, then X is a new variant of Y.

Definition 4.1.13 A **ground instance** of an object X, where X is a term, literal, or clause, is the object σX for some ground substitution σ such that $Dom(\sigma) \supseteq Var(X)$.

The next two propositions are well-known in logic programming ([Llo87]).

Proposition 4.1.14 Let $\sigma = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ and $\theta = \{u_1 \mapsto s_1, \ldots, u_m \mapsto s_m\}$ be substitutions. Then $\sigma \circ \theta$ is obtained from the set

$$\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n, u_1 \mapsto \sigma s_1, \ldots, u_m \mapsto \sigma s_m\}$$

by deleting any binding $v_j \mapsto t_j$ for which $v_j \in \{u_1, \ldots, u_m\}$ and deleting any binding $u_i \mapsto \sigma s_i$ for which $u_i \equiv \sigma s_i$.

Note that $Dom(\sigma \circ \theta) = (Dom(\sigma) \cup Dom(\theta)) \setminus \{u \mid u \equiv \sigma s, u \mapsto s \in \theta\}$. For the variable u such that $u \mapsto s \in \theta$ and $u \equiv \sigma s$, we must have s is a variable, and $s \mapsto u \in \sigma$.

Proposition 4.1.15 θ is idempotent iff $Dom(\theta) \cap VRan(\theta) = \emptyset$.

Definition 4.1.16 Let V be a set of variables. The **restriction of a substitution** θ **to** V, denoted by $\theta\lceil V$, is $\{x \mapsto t \mid x \mapsto t \in \theta, x \in V\}$.

Definition 4.1.17 Let V be a set of variables. The **restriction of a solved form** ζ **to** V, denoted by $\zeta\lceil V$, is a solved form $\exists\vec{z}\{set(y_1), \cdots, set(y_m), x_1 = t_1, \cdots, x_n = t_n\}$ derived from ζ, where each $x_i = t_i$ occurs in ζ and $x_i \in V$; and each $set(y_j)$ occurs in ζ and if y_j is a free variable then $y_j \in V$ while if y_j is an existential variable then y_j occurs among the t_i.

For example, if

$$\zeta = \exists y_2, y_3\{set(y_1), set(y_2), set(y_3), set(y_4), x_1 = y_2, x_2 = \{1/y_3\}\}$$

and $V = \{x_1, y_1\}$, then

$$\zeta\lceil V = \exists y_2\{set(y_1), set(y_2), x_1 = y_2\}.$$

That is, $\zeta\lceil V$ is obtained as the submultiset of ζ containing only those $x_i = t_i$ for which $x_i \in V$, and only those $set(y_j)$ for which either y_j is free and $y_j \in V$, or y_j is existential and y_j occurs in the t_i of the submultiset. Finally, only those existential variables of ζ are retained in $\zeta\lceil V$ that occur in the submultiset.

The above definition looks complicated and fortunately we will not have to deal with it. They are needed to form subsequent definitions in a general way, but the following property will help us to ignore it.

It is easy to see that if $Var(\zeta) \subseteq V$, then $\zeta \lceil V = \zeta$.

4.2 Terminology of Unification

Unification can be seen as the problem of solving equations over arbitrary abstract algebras ([JK91]). In the context of $SetAx$, it is seen as solving constraints over arbitrary structures that model $SetAx$. The solutions desired are those arising out of the properties of $SetAx$.

Definition 4.2.1 A **solution** to a constraint χ in a model $\mathcal{J}$ of $SetAx$ is an assignment $\mathbf{A}$ such that $\chi^{\mathbf{A},\mathcal{J}}$ holds.

In general, there may be infinitely many solutions to a constraint χ. The notion of *unifier* is a step towards the construction of a finite representation of their solutions. In the case of $SetAx$, it happens that every solution to a constraint is representable by some unifier, and that, in general, a finite set of unifiers is required to describe all solutions. Thus, we are interested in minimal sets of unifiers representing all solutions, and thereby, in comparisons between unifiers. We give the requisite definitions below, adapted from unification theory to the case of $SetAx$. Our definitions are given a logical basis rather than the usual syntactic one of composition of substitutions, since that is what appears more appropriate within our logical theory $SetAx$.

Definition 4.2.2 Let ζ_1, ζ_2 be solved forms and V be a set of variables. Let $\zeta_1' = \zeta_1 \lceil V$ and $\zeta_2' = \zeta_2 \lceil V$. Then, ζ_1 **is an instance of** ζ_2 **on** V, or ζ_2 **is more general than** ζ_1 **on** V, denoted by $\zeta_1 \geq^V \zeta_2$, if $SetAx \models \forall(\zeta_1' \rightarrow \zeta_2')$.

In the above definition, if $V = \Sigma_V$, i.e., V is all the variables, then we write $\zeta_1 \geq \zeta_2$, and the definition reduces to $SetAx \models \forall(\zeta_1 \rightarrow \zeta_2)$. The logical view of $\zeta_1 \geq^V \zeta_2$ is that every solution of ζ_1 is a solution of ζ_2, when restricted to variables V. For example, $\{z = \emptyset, x = \emptyset\} \geq \{x = z\}$ and $\{x = \emptyset\} \geq^{\{x\}} \exists z\{set(z), x = z\}$.

From the definition, it is easy to see that $\leq^V$ is a pre-order. It is, however, not a partial order, since for $\zeta_1 \equiv set(y) \wedge x = y$ and $\zeta_2 \equiv set(x) \wedge y = x$, we have $\zeta_1 \geq \zeta_2$ and $\zeta_2 \geq \zeta_1$, but not $\zeta_1 = \zeta_2$. The following definition converts it to a partial order in the standard way.

Definition 4.2.3 Let ζ_1, ζ_2 be solved forms and V be a set of variables. Then, ζ_1 **is a variant of** ζ_2 **on** V, denoted by $\zeta_1 \simeq^V \zeta_2$, if $\zeta_1 \leq^V \zeta_2$ and $\zeta_1 \geq^V \zeta_2$.

Logically, ζ_1 is a variant of ζ_2 on V if they have the same solutions when restricted to V, i.e., $SetAx \models \forall(\zeta_1\lceil V \leftrightarrow \zeta_2\lceil V)$. The following are examples of variants: $\{x = a\} \simeq \{x = a\}$, $\exists z\{set(z), x = \{1/z\}\} \simeq \exists z\{set(z), x = \{1/\{1/z\}\}\}$, and $\exists z_2\{set(z_2), u = c(y), z = y, x = \{1/z_2\}\} \simeq^{\{x,z,y,u\}} \exists z_1\{set(z_1), u = c(z), y = z, x = \{1/z_1\}\}$.

Definition 4.2.4 Let ξ be a solved form, $\Omega = \{\zeta_1, \ldots, \zeta_k\}$, $k \geq 0$, be a set of solved forms, and V be a set of variables. Let $\xi' = \xi\lceil V$ and $\zeta_i' = \zeta_i\lceil V$, for $1 \leq i \leq k$. Then ξ **is subsumed by** Ω **on** V, denoted by $\xi \geq^V \Omega$, if $SetAx \models \forall(\xi' \rightarrow \zeta_1' \vee \cdots \vee \zeta_k')$.

Definition 4.2.5 Let χ be a constraint. Then, a solved form ζ is a **unifier** of χ if $Var(\zeta) \subseteq Var(\chi)$ and $SetAx \models \forall(\zeta \rightarrow \chi)$.

Definition 4.2.6 Let Ω be a finite set of unifiers of χ. Then Ω is a **complete set of unifiers of** χ, denoted by $CSU(\chi)$, if for any unifier ξ of χ, ξ is subsumed by Ω on $Var(\chi)$, i.e., $\xi \geq^{Var(\chi)} \Omega$.

Definition 4.2.7 Let ζ be a unifier of χ. Then ζ is a **maximally general unifier (mgu) of** χ if for any unifier ξ of χ that is more general than ζ on $Var(\chi)$, we have ξ is a variant of ζ on $Var(\chi)$.

Definition 4.2.8 Let Ω be a $CSU(\chi)$. Then Ω is a **minimal complete set of unifiers of** χ, denoted by $MCSU(\chi)$, if for every $\zeta \in \Omega$, ζ is not subsumed by $\Omega \backslash \{\zeta\}$ on $Var(\chi)$.

A simplification that occurs in comparing unifiers, say ζ and ξ, of a constraint χ on $Var(\chi)$ is that $\zeta\lceil Var(\chi) = \zeta$ and $\xi\lceil Var(\chi) = \xi$ (since $Var(\zeta) \subseteq Var(\chi)$ and similarly for ξ). Thus, restrictions of unifiers can be ignored in comparing them.

We next look at the issues present in describing unification algorithms through a rule-based approach. This is a style having the advantages of ease of design and ease of correctness proofs which arise from making a clear distinction between the rewrite rules and their actual use (the control) (see [JK91] §3.1). Principally, it views unification as a step by step process of transforming unification problems through rewrite rules until a solved form is obtained. A rule is called deterministic if no other rule applies concurrently on the same selected atom; otherwise it is called non-deterministic. In general, these rules may be non-deterministic, so that the several transformation rules that apply concurrently on the same selected atom of the unification problem obtains an equivalent collection of unification problems. In our case, we take equivalence to be logical equivalence. Thus, a tree is defined in the rewriting process whose leaves yield solved forms.

A systematic guide to giving correctness proofs involves the following four steps of *soundness*, *completeness*, *termination*, and *fairness*. These are terms taken from [JK91] to conform to the practice in the unification literature. We note that the terms soundness and completeness below are used in a somewhat augmented or different sense than usual. (Thus these terms, in particular the word completeness, is being used in several senses in this dissertation, and we expect that the particular sense will always be clear from the context.)

• Soundness: Show that rewriting a selected atom of a unification problem through all possible non-deterministic rules yields a collection of children unification problems whose disjunction is logically equivalent to the parent unification problem. This ensures that the collection of unifiers of the children problem is identical to the collection of unifiers of the parent problem.

• Completeness: Show that the normal (or irreducible) forms (with respect to the rewrite rules) are indeed solved forms.

• Termination: Show that the specific control or class of controls for applying the rewrite rules leads to termination of the step by step rewriting process. Part of the control involves choosing the selected atom from a unification problem.

Finding a termination proof is generally the difficult aspect of unification.

- Fairness: Show that normal forms are indeed reached for the chosen control.

4.3 Unification Algorithm

The following propositions give the basis of the rewrite rules.

Proposition 4.3.1 $SetAx^- \models \varphi$, where φ is any of:

(i) $set(x) \wedge set(y) \rightarrow \big(\{x_1/x\} = \{y_1/y\} \leftrightarrow$

$\big((x_1 \notin x \wedge y_1 \notin y \wedge x_1 = y_1 \wedge x = y)$

$\vee(x_1 \notin x \wedge y_1 \notin y \wedge x_1 \neq y_1 \wedge \exists z(set(z) \wedge x = \{y_1/z\} \wedge y_1 \notin z \wedge y = \{x_1/z\} \wedge x_1 \notin z))$

$\vee(x_1 \notin x \wedge y_1 \in y \wedge y = \{x_1/x\})$

$\vee(x_1 \in x \wedge y_1 \notin y \wedge x = \{y_1/y\})$

$\vee(x_1 \in x \wedge y_1 \in y \wedge x = y)\big)\big)$

(ii) $set(x) \wedge set(y) \rightarrow \big(\{x_1/x\} = \{y_1/y\} \leftrightarrow$

$\big((x_1 = y_1 \wedge x = y)$

$\vee\exists z(set(z) \wedge x = \{y_1/z\} \wedge y = \{x_1/z\})$

$\vee(x_1 = y_1 \wedge y = \{x_1/x\})$

$\vee(x_1 = y_1 \wedge x = \{y_1/y\})\big)\big)$

Proof: (i): Assume $set(x)$, $set(y)$. ($\leftarrow$): Straightforward. ($\rightarrow$): Straightforward. Take combinations of cases of $x_1 \notin x$, $x_1 \in x$, with cases of $y_1 \notin y$, and $y_1 \in y$. In the case of the combination of $x_1 \notin x$ and $y_1 \notin y$ we get two subcases of $x_1 = y_1$ and $x_1 \neq y_1$.

(ii): This is the same as part (i), except that the equivalence holds even when all the negative relations in the disjuctions are dropped. Also, further simplifications can be made in the disjunctions as shown below.

Assume $set(x)$, $set(y)$. ($\leftarrow$): Straightforward. ($\rightarrow$): Take combinations of cases of $x_1 \notin x$, $x_1 \in x$, with cases of $y_1 \notin y$, and $y_1 \in y$. We show three of these cases.

Case (1): $x_1 \notin x$, $y_1 \notin y$. Case (1.1): $x_1 = y_1$. Then $x = y$. So, $x_1 = y_1 \wedge x = y$. Case (1.2): $x_1 \neq y_1$. Then, $x_1 \in y$, i.e., $\exists z(set(z) \wedge y = \{x_1/z\} \wedge x_1 \notin z)$. So, $\{x_1/x\} = \{y_1/\{x_1/z\}\} = \{x_1/\{y_1/z\}\}$. Then, $x_1 \notin \{y_1/z\}$. So, $x = \{y_1/z\}$. Thus, $\exists z(set(z) \wedge x = \{y_1/z\} \wedge y = \{x_1/z\})$.

Case (2): $x_1 \notin x$, $y_1 \in y$. So, $y = \{x_1/x\}$, and $y_1 \in \{x_1/x\}$, i.e., $y_1 = x_1$ or $y_1 \in x$. Case (2.1): $y_1 = x_1$. So, $x_1 = y_1 \wedge y = \{x_1/x\}$. Case (2.2): $y_1 \in x$. So, $\exists z(set(z) \wedge x = \{y_1/z\})$, i.e., $set(z) \wedge x = \{y_1/z\}$. It is easy to see that $y = \{x_1/\{y_1/z\}\} = \{x_1/\{x_1/z\}\} = \{x_1/z\}$. Thus, $\exists z(set(z) \wedge x = \{y_1/z\} \wedge y = \{x_1/z\})$.

Case (4): $x_1 \in x$, $y_1 \in y$. So, $x = y$, and $\exists z(set(z) \wedge x = \{x_1/z\})$, i.e., $x = \{x_1/z\}$. Therefore, $y_1 \in \{x_1/z\}$. Case (4.1): $y_1 = x_1$. So, $x = \{y_1/z\} \wedge y = \{x_1/z\}$, i.e., $\exists z(set(z) \wedge x = \{y_1/z\} \wedge y = \{x_1/z\})$. Case (4.2): $y_1 \in z$. So, $\exists z_1(set(z_1) \wedge z = \{y_1/z_1\})$, i.e., $set(z_1) \wedge z = \{y_1/z_1\}$. It is easy to see that $x = \{y_1/\{x_1, y_1/z_1\}\}$ and $y = \{x_1/\{x_1, y_1/z_1\}\}$. Thus, $\exists z(set(z) \wedge x = \{y_1/z\} \wedge y = \{x_1/z\})$ (take $\{x_1, y_1/z_1\}$ as witness). ∎

From the above proposition, we get that an equality constraint like that occurring on the lhs of these equivalences can be unified by recursively solving the constraints on the rhs. The constraints on the rhs are expected to be simpler according to some complexity measure on terms, so that termination of the rewriting process is ensured.

If we use part (i) of the proposition for rewriting, it means solving negative as well as positive relations in the recursive step. Traditionally, in the unification literature, positive relations have been solved in terms of positive relations alone and we choose to do the same. An advantage is that solved forms for positive relations alone are much nicer than when negative relations are involved too. Also, we intend to deduce properties of $SetAx$ and $SetAx^-$ from the unification algorithms such as in §4.5, where we find that there are properties for positive relations that do not hold when negative relations are included too.

Our unification algorithm was based along the lines of part (ii) of the above proposition, but part (ii) is a more insightful form than we had, a form that has been noticed by [DOPR93] and used as the basis of their set unification algorithm. Thay have also given a termination proof for their algorithm and hence we describe essentially their algorithm here. (We note that a sketch of a set unification algorithm in the style of part (ii) was first given in [JP89] without proof, but it turned out to be an incomplete and non-terminating version.)

In seeking to describe the solving of positive relations without any dependence on negative ones too, we lose on two aspects. For one, the mutual exclusivity of the disjuncts in the recursive step in part (i) is lost. This has implications for the minimality of the collection of unifiers obtained using part (ii).

Secondly, the part (ii) is not capable of solving arbitrary set terms anymore. For example, in solving the problem $\{\underline{\{1/y\} = \{3/y\}}\}$ using the second disjunct of part (ii) leads to the solving of $\{set(z), \underline{y = \{3/z\}}, y = \{1/z\}\}$, i.e., to $\{set(z), y = \{3/z\}, \underline{\{3/z\} = \{1/z\}}\}$. Thus we have a recursive problem that has the same size as the starting problem and hence non-termination. (This does not ocur when using the second disjunct of part (i), for it leads to the solving of $\{1 \notin y, 3 \notin y, 1 \neq 3, set(z), y = \{3/z\}, 3 \notin z, y = \{1/z\}, 1 \notin z\}$. In subsequent solving of $3 \notin y$, i.e., $3 \notin \{3/z\}$, we get failure and hence termination.)

The situation of non-termination occurs more generally in solving set terms $s = t$ for which $last(s)$ and $last(t)$ are identical variables. Note that the defect is not in the control strategy of rewriting, for any order of choosing the selected atom in our above example would lead to a problem of the same size. To take care of such cases, an alternate equivalence has to be used, such as part (i) or (ii) of the next proposition. The part (ii) is used in the [DOPR93] algorithm.

Proposition 4.3.2

(i) $set(w) \rightarrow (\{x_1, \ldots, x_m/w\} = \{y_1, \ldots, y_n/w\} \leftrightarrow$

$x_1 \in \{y_1, \ldots, y_n/w\} \wedge \cdots \wedge x_m \in \{y_1, \ldots, y_n/w\} \wedge$

$y_1 \in \{x_1, \ldots, x_m/w\} \wedge \cdots \wedge y_n \in \{x_1, \ldots, x_m/w\})$

(ii) $set(w) \rightarrow \big(\{x_1, \ldots, x_m/w\} = \{y_1, \ldots, y_n/w\} \leftrightarrow$

$\big(\bigvee_{j=1}^{n}(x_1 = y_j \wedge \{x_2, \ldots, x_m/w\} = \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$

$\vee x_1 = y_j \wedge \{x_1, \ldots, x_m/w\} = \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$

$\vee x_1 = y_j \wedge \{x_2, \ldots, x_m/w\} = \{y_1, \ldots, y_n/w\})$

$\vee \exists z(set(z) \wedge w = \{x_1/z\} \wedge \{x_2, \ldots, x_m/z\} = \{y_1, \ldots, y_n/z\})\big)\big)$

Proof: (i): Straightforward, from extensionality and that $w \subseteq \{y_1, \ldots, y_n/w\}$ and $w \subseteq \{x_1, \ldots, x_m/w\}$ are tautological statements under $SetAx^-$.

(ii): Assume $set(w)$. ($\leftarrow$): Straightforward. ($\rightarrow$): Take cases of $x_1 \in \{y_1, \ldots, y_n\}$ and $x_1 \notin \{y_1, \ldots, y_n\}$.

Case (1): $x_1 \in \{y_1, \ldots, y_n\}$. Let $1 \leq j \leq n$. Case (1.j): $x_1 = y_j$. Take combinations of cases of x_1 being in and not being in $\{x_2, \ldots, x_m/w\}$, with cases of y_j being in and not being in $\{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$. We show two of these cases. Case (1.j.1): $x_1 \notin \{x_2, \ldots, x_m/w\}$, $y_j \notin \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$. So, $\{x_2, \ldots, x_m/w\} = \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$. Case (1.j.2): $x_1 \notin \{x_2, \ldots, x_m/w\}$, $y_j \in \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$. So, $\{x_1, \ldots, x_m/w\} = \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$. The case (1.j.4) of $x_1 \in \{x_2, \ldots, x_m/w\}$, $y_j \in \{y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_n/w\}$ gives the same conclusion as case (1.j.1).

Case (2): $x_1 \notin \{y_1, \ldots, y_n\}$. So, $x_1 \in w$, i.e., $\exists z(set(z) \wedge w = \{x_1/z\} \wedge x_1 \notin z)$. Then, $set(z) \wedge w = \{x_1/z\} \wedge x_1 \notin z$. Case (2.1): $x_1 \notin \{x_2, \ldots, x_m\}$. So, $\{x_2, \ldots, x_m/z\} = \{y_1, \ldots, y_n/z\}$ by removing x_1 from $\{x_1, \ldots, x_m/w\}$ and from $\{y_1, \ldots, y_n/w\}$. Hence $\exists z(set(z) \wedge w = \{x_1/z\} \wedge \{x_2, \ldots, x_m/z\} = \{y_1, \ldots, y_n/z\}$. Case (2.2): $x_1 \in \{x_2, \ldots, x_m\}$. It is easy to see that $\exists z_1(set(z_1) \wedge$

$w = \{x_1/z_1\} \wedge \{x_2, \ldots, x_m/z_1\} = \{y_1, \ldots, y_n/z_1\}$ by using $\{x_1/z\}$ as witness for z_1. ■

Below are the rewrite rules that can be applied to a constraint. In each rule, a selected atom is rewritten to the form on the rhs of the rule. Here, F denotes failure in solving the constraint. An existential variable is made prominent in a rule only if it is involved in the transformation of the selected atom. By 'new variable' we mean new to the lhs of the rule. By applying a substitution to a constraint K we mean that it is applied to the multiset of K. Also, we use the phrase 'x occurs in K' iff x occurs in the multiset of K.

(S1) $K \cup \{set(\emptyset)\} \Rightarrow K$

(S2) $K \cup \{set(x)\} \Rightarrow K$

if $set(x)$ occurs in K.

(S3) $\exists z K \cup \{set(z)\} \Rightarrow K$

if $set(z)$ does not occur in K.

(S4) $K \cup \{set(c(\vec{t})\} \Rightarrow F$

if $c \in \Sigma_{\overline{C}}$.

(S5) $K \cup \{set(\{s_1/s_2\})\} \Rightarrow K \cup \{set(last(s_2))\}$

(E1) $\exists z K \Rightarrow K$

if z does not occur in K.

(E2) $\exists z K \cup \{z = t\} \Rightarrow K$

if $z \notin Var(t)$ and z does not occur in K.

(E3) $K \cup \{x = x\} \Rightarrow K$

(E4) $K \cup \{x = t\} \Rightarrow \{x \mapsto t\} K \cup \{x = t\}$

if $x \notin Var(t)$ and x occurs in K.

(E5) $K \cup \{x = t\} \Rightarrow \exists z K \cup \{set(z), x = t'\}$

if $x \in Var(t)$, $x \not\equiv t$,

$t \equiv \{t_1, \ldots, t_n/x\}$ with $n \geq 1$, and $x \notin Var(t_1, \ldots, t_n)$.

Here $t' \equiv \{t_1, \ldots, t_n/z\}$, z a new variable.

(E6) $K \cup \{x = t\} \Rightarrow F$

if $x \in Var(t)$, $x \not\equiv t$, and it is not the case that

($t \equiv \{t_1, \ldots, t_n/x\}$ with $n \geq 1$ and $x \notin Var(t_1, \ldots, t_n)$).

(E7) $K \cup \{t = x\} \Rightarrow K \cup \{x = t\}$

if t is not a variable.

(E8) $K \cup \{c(\vec{s}) = d(\vec{t})\} \Rightarrow F$

if $c \not\equiv d$.

(E9) $K \cup \{c(s_1, \ldots, s_m) = c(t_1, \ldots, t_m)\} \Rightarrow K \cup \{s_1 = t_1, \ldots, s_m = t_m\}$

if $m \geq 0$.

(E10) $K \cup \{\{s_1, \ldots, s_m/w\} = \{t_1, \ldots, t_n/w\}\} \Rightarrow K \cup \{set(w), s_1 = t_j,$

$\{s_2, \ldots, s_m/w\} = \{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n/w\}\}$

for $1 \leq j \leq n$.

(E11) $K \cup \{\{s_1, \ldots, s_m/w\} = \{t_1, \ldots, t_n/w\}\} \Rightarrow K \cup \{set(w), s_1 = t_j,$

$\{s_1, \ldots, s_m/w\} = \{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n/w\}\}$

for $1 \leq j \leq n$.

(E12) $K \cup \{\{s_1, \ldots, s_m/w\} = \{t_1, \ldots, t_n/w\}\} \Rightarrow K \cup \{set(w), s_1 = t_j,$

$\{s_2, \ldots, s_m/w\} = \{t_1, \ldots, t_n/w\}\}$

for $1 \leq j \leq n$.

(E13) $K \cup \{\{s_1, \ldots, s_m/w\} = \{t_1, \ldots, t_n/w\}\} \Rightarrow \exists z K \cup \{set(z), w = \{s_1/z\},$

$\{s_2, \ldots, s_m/z\} = \{t_1, \ldots, t_n/z\}\}$

Here z is a new variable.

(E14) $K \cup \{\{s_1/s_2\} = \{t_1/t_2\}\} \Rightarrow \exists z K \cup \{\{s_1/z\} = t_2, s_2 = \{t_1/z\}, set(z)\}$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

Here z is a new variable.

(E15) $K \cup \{\{s_1/s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{s_1 = t_1, s_2 = \{t_1/t_2\}, set(t_2)\}$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

(E16) $K \cup \{\{s_1/s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{s_1 = t_1, \{s_1/s_2\} = t_2, set(s_2)\}$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

(M1) $K \cup \{s \in \emptyset\} \Rightarrow F$

(M2) $K \cup \{s \in x\}\} \Rightarrow \exists z K \cup \{set(z), x = \{s/z\}\}$

Here, z is a new variable.

(M3) $K \cup \{s \in c(\vec{t})\} \Rightarrow F$

if $c \in \Sigma_{\overline{C}}$.

(M4) $K \cup \{s \in \{t_1/t_2\}\} \Rightarrow K \cup \{set(last(t_2)), s = t_1\}$

(M5) $K \cup \{s \in \{t_1/t_2\}\} \Rightarrow K \cup \{s \in t_2\}$

As for the control strategy, an unrestricted selection strategy for the selected atom is not possible since it may lead to non-termination. For example, the following derivation sequence would not terminate.

$\{\underline{\{1/x\} = \{2/y\}}, \{3/x\} = \{4/y\}\}$

$\Downarrow$ E14

$\exists z_1\{\{1/z_1\} = y, x = \{2/z_1\}, set(z_1), \underline{\{3/x\} = \{4/y\}}\}$

$\Downarrow$ E14

$\exists z_1, z_2\{\underline{\{1/z_1\} = y}, x = \{2/z_1\}, set(z_1), \{3/z_2\} = y, x = \{4/z_2\}, set(z_2)\}$

$\Downarrow$ E7

$$\exists z_1, z_2\{\underline{y = \{1/z_1\}}, x = \{2/z_1\}, set(z_1), \{3/z_2\} = y, x = \{4/z_2\}, set(z_2)\}$$

$$\Downarrow \text{E4}$$

$$\exists z_1, z_2\{y = \{1/z_1\}, \underline{x = \{2/z_1\}}, set(z_1), \{3/z_2\} = \{1/z_1\}, x = \{4/z_2\}, set(z_2)\}$$

$$\Downarrow \text{E4}$$

$$\exists z_1, z_2\{y = \{1/z_1\}, x = \{2/z_1\}, set(z_1), \underline{\{3/z_2\} = \{1/z_1\}}, \{2/z_1\} = \{4/z_2\}, set(z_2)\}$$

The last two equalities have the same size as the starting problem and the above control could be applied repeatedly to them to obtain non-termination. The difficulty arose in that in the second step the last atom was selected before previous atoms had been solved. Since the *last* terms in the last atom were not identical variables, the rules that would apply to it were E9 and E14 to E16, of which E14 was applied. However, if the prior atoms had been solved first, then the last atom would have its *last* terms as identical variables. This is shown in the derivation below. Hence, the applicable rules now for the last equality would be E9 to E13.

$$\{\underline{\{1/x\} = \{2/y\}}, \{3/x\} = \{4/y\}\}$$

$$\Downarrow \text{E14}$$

$$\exists z_1\{\underline{\{1/z_1\} = y}, x = \{2/z_1\}, set(z_1), \{3/x\} = \{4/y\}\}$$

$$\Downarrow \text{E7}$$

$$\exists z_1\{\underline{y = \{1/z_1\}}, x = \{2/z_1\}, set(z_1), \{3/x\} = \{4/y\}\}$$

$$\Downarrow \text{E4}$$

$$\exists z_1, z_2\{y = \{1/z_1\}, \underline{x = \{2/z_1\}}, set(z_1), \{3/x\} = \{4/\{1/z_1\}\}\}$$

$$\Downarrow \text{E4}$$

$$\exists z_1, z_2\{y = \{1/z_1\}, x = \{2/z_1\}, set(z_1), \underline{\{3/\{2/z_1\}\} = \{4/\{1/z_1\}\}}\}$$

All this suggests that a control strategy that selected, at each step, the leftmost applicable atom for rewriting would ensure termination. The proof of termination in [DOPR93] uses such a strategy.

4.4 Correctness of Unification

The following theorem is the proof of correctness of the unification algorithm based on the control described in the statement of the theorem.

Theorem 4.4.1 Starting with a unification problem K (or χ) and using the rewrite rules above repeatedly in a leftmost selection strategy until no rule is applicable results in a finite tree whose leaves are labelled F or a solved form. The collection of solved forms is a $CSU(\chi)$. More specifically, if $\Omega = \{\zeta_1, \ldots, \zeta_k\}$, $k \geq 0$, is the set of solved forms at the leaves of the tree, then $SetAx^- \models \forall(\zeta_i \rightarrow \chi)$ and $SetAx \models \forall(\chi \rightarrow \zeta_1 \vee \cdots \vee \zeta_k)$.

Proof: We first consider the four steps of soundness, completeness, termination, and fairness of a correctness proof. Then, we draw further conclusions from them.

Soundness: Straightforward. Consider each rewrite rule 'lhs $\Rightarrow$ rhs' in turn and show that $SetAx^- \models \forall(\text{rhs} \rightarrow \text{lhs})$. Notice also that in each rule all new variables added are existential and that the free variables of the rhs of a rule are a subset of those on the lhs.

Next, consider all possible forms of an atom for rewriting. For each atom, suppose there are $n \geq 1$ many applicable non-deterministic rules 'lhs $\Rightarrow$ rhs$_i$', for $1 \leq i \leq n$. Show that $SetAx \models \forall(\text{lhs} \rightarrow \text{rhs}_1 \vee \cdots \vee \text{rhs}_n)$.

From the above it follows that $SetAx \models \forall(\text{lhs} \leftrightarrow \text{rhs}_1 \vee \cdots \vee \text{rhs}_n)$.

Completeness: Let K be a normal or irreducible form in the rewriting. We show that K is a solved form. If K is empty, then it has no existential variables (due to rule E1), and K is a solved form. If K is non-empty, then its elements must take the forms for atoms that are not covered by those appearing on the lhs of rules. A careful examination of the rules shows that the forms not covered are $x = t$ with $x \notin Var(t)$ and $set(x)$, with additional restrictions on them. Hence, we can say K is of the form $\exists\vec{z}\{set(y_1), \cdots, set(y_m), x_1 = t_1, \cdots, x_n = t_n\}$ where $m, n \geq 0$. The x_is are free variables because of rules E2 and E4. For if any x_i is existential and occurs in the rest of K then rule E4 would apply. Or if any x_i is existential and does not occur in the rest of K then rule E2 would apply. Similar reasoning can be used to show that K fulfils all the requirements of a solved form.

Termination: The termination of the *set* relation is trivial — clearly the term size decreases across each rule S1 to S5. The termination for equality relation has been shown in [DOPR93]. From that, the termination for the membership relation follows.

Fairness: Follows from the definition of the control.

Next, using the above, we get the rewriting tree to be finite and hence can apply induction on the depth of this tree.

By transitivity and induction it follows that $SetAx^- \models \forall(\zeta_i \rightarrow \chi)$. Hence, each ζ_i is a unifier. It is easy to show by induction that $SetAx \models \forall(\chi \rightarrow \zeta_1 \vee \cdots \vee \zeta_k)$. Thus, we have $SetAx \models \forall(\chi \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$, i.e., Ω is a $CSU(\chi)$. ■

While the above theorem gives Ω to be a $CSU(\chi)$, it is not necessarily a $MCSU(\chi)$. For example, for $\chi \equiv \{x/y\} = \{1/\{1/\emptyset\}\}$, we have Ω contains $\zeta_1 \equiv x = 1 \wedge y = \{1/\emptyset\}$ and $\zeta_2 \equiv x = 1 \wedge y = \{1/\{1/\emptyset\}\}$, but $\zeta_1 \geq^{\{x,y\}} \Omega\backslash\{\zeta_1\}$ since ζ_1 is a variant of ζ_2 on $\{x, y\}$. It has been generally observed in unification theory that the algorithms that solve a unification problem do not simultaneously produce a minimal complete set of unifiers.

We now look at solving matching constraints. Accordingly, we change our definition of unification problem and solved forms. The solved forms now take a simpler form.

A *unification problem* is a matching constraint. A *solved form* ζ is a matching constraint of the form $\{x_1 = t_1, \cdots, x_n = t_n\}$ where $n \geq 0$, and the x_i's are distinct variables and the t_i's are ground. The *domain* of ζ, $Dom(\zeta) = \{x_1, \ldots, x_n\}$ and the *range* of ζ, $Ran(\zeta) = \{t_1, \ldots, t_n\}$.

A solved form $\zeta = \{x_1 = t_1, \cdots, x_n = t_n\}$ obviously corresponds with the ground substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Hence, we shall use such solved forms as substitutions and apply them to objects as such. We can also represent ζ as $\mathcal{E}(\theta)$, where $\mathcal{E}$ is a transform that converts the maps $x_i \mapsto t_i$ of θ into equations $x_i = t_i$ of ζ.

When matching constraints are considered it is usual to replace the term *unify* with the term *match* in the terminology of unifiers, and we follow this practise here. Thus we have matchers and complete sets of matchers in place of unifiers and complete sets of unifiers. All the definitions about unifiers, as in Defn. 4.2.5 to Defn. 4.2.8, go through for matchers except that we can be more specific about matchers, as given next.

Definition 4.4.2 Let χ be a matching constraint. Then, a solved form ζ is a **matcher** of χ if $Var(\zeta) = Dom(\zeta) = Var(\chi)$ and $SetAx \models \forall(\zeta \rightarrow \chi)$.

We can use the preceeding theorem to solve matching constraints and obtain a complete set of matchers, as shown by the next theorem. (In fact, the rewrite rules were set up so they could be applicable to matching problems too.)

Theorem 4.4.3 Starting with a matching problem K (or χ) and using the rewrite rules above repeatedly in a leftmost selection strategy until no rule is applicable results in a finite tree whose leaves are labelled F or a solved form. The collection of solved forms is a complete set of matchers.

Proof: Since the rules and control are the same as in Thm. 4.4.1, we simply use the results of that theorem. The only additional property we have to prove here is that the leaves are solved forms in the context of matching.

To show this, we consider each rule that can apply to a matching constraint and use induction on the depth of the tree. For example, for the rule E14

$$K \cup \{\{s_1/s_2\} = \{t_1/t_2\}\} \Rightarrow \exists z K \cup \{\{s_1/z\} = t_2, s_2 = \{t_1/z\}, set(z)\}$$

we reason informally as follows.

We have $\{s_1/z\} = t_2$ is a matching constraint and hence has a set of matchers which include an equality for z. This is substituted on the rest of the matching problem, so that $\{t_1/z\}$ and $set(z)$

become ground terms. Additionally, $\exists z$ is eliminated by rule E2. Hence, when $s_2 = \{t_1/z\}$ is selected for solving, it is a matching constraint and has a set of matchers. Finally, when $set(z)$ is selected for solving, it too is ground and hence by rules S1 to S5, it is either eliminated or leads to F. Thus, putting all these matchers together, we obtain solved forms that are matchers of $\{s_1/s_2\} = \{t_1/t_2\}$.

■

We again have that the set of solved forms in the rewrite tree is not a complete set of matchers. The same example of $\chi \equiv \{x/y\} = \{1/\{1/\emptyset\}\}$ used before can be used to show this.

4.5 Properties of SetAx from Unification

The following properties of $SetAx$ and $SetAx^-$ ensue from unification.

Theorem 4.5.1 Let $\Omega = \{\zeta_1, \ldots, \zeta_k\}$ for some $k \geq 0$ be the set of all solved forms obtained from the rewriting procedure for a constraint χ. Then,

(i) $SetAx \models \forall(\chi \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$

(ii) $SetAx^- \models \exists\chi \;\Leftrightarrow\; \Omega \neq \emptyset$

(iii) $SetAx \models \exists\chi \;\Leftrightarrow\; SetAx^- \models \exists\chi$

(iv) $SetAx \models \exists\chi$ or $SetAx \models \neg\exists\chi$

(v) $SetAx^- \not\models \exists\chi \;\Rightarrow\; SetAx \models \neg\exists\chi$

Proof: (i): This is simply a repeat of a result of Thm. 4.4.1, for the sake of easy reference.

(ii): ($\Rightarrow$) Follows from (i) above and consistency of $SetAx$. ($\Leftarrow$) Let $\zeta \in \Omega$ witness $\Omega \neq \emptyset$. Starting from $SetAx^- \models \forall(\zeta \rightarrow \chi)$, we show $SetAx^- \models \exists\chi$. Let $\zeta \equiv \exists\vec{z}(set(y_1)\wedge\cdots\wedge set(y_m)\wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n)$ where $m, n \geq 0$ and let $\vec{u} = Var(\chi)$ be the free variables of χ. We have $Var(\zeta) \subseteq Var(\chi)$. Hence, $SetAx^- \models \forall\vec{u}, \vec{z}(set(y_1) \wedge \cdots \wedge set(y_m) \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n \rightarrow \chi)$.

We next substitute instances for $\vec{u}$ and $\vec{z}$ such that the antecedent holds. Let substitutions θ, σ be $\theta = \{y_1 \mapsto \emptyset, \ldots, y_m \mapsto \emptyset\}$ and $\sigma = \{x_1 \mapsto \theta t_1, \ldots, x_n \mapsto \theta t_n\}$. Clearly $\sigma\theta(set(y_1) \wedge$

$\cdots \wedge set(y_m) \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n)$ holds in $SetAx^-$. Hence $SetAx^- \models \forall \sigma\theta\chi$. This gives $SetAx^- \models \exists\chi$.

(iii): Show $SetAx^- \not\models \exists\chi \Rightarrow SetAx \not\models \exists\chi$, which follows from (ii) and (i) above, and from consistency of $SetAx$.

(iv): Follows from (iii), (ii), and (i) above.

(v): Follows from (ii) and (i) above. ■

The property (v) above justifies the adequacy of $FreeAx$ (by taking $\chi \equiv s = t$ or $\chi \equiv s \in t$).

The property (iv) above shows that $SetAx$ is a complete theory over formulae φ of the form $\exists\chi$ or $\neg\exists\chi$. As a consequence, the following properties of complete theories hold for $SetAx$. Let $\mathcal{J}$ be any structure that models $SetAx$. Then, $\varphi^{\mathcal{J}}$ holds iff $SetAx \models \varphi$, and $\varphi^{\mathcal{J}}$ does not hold iff $SetAx \models \neg\varphi$, for φ of the above form. In other words, satisfaction of φ in a model of $SetAx$ follows the behaviour of logical consequence of φ from $SetAx$, and vice-versa. So, satisfaction of φ behaves identically in all models of $SetAx$.

We also have that $SetAx \models \varphi \vee \psi \Rightarrow SetAx \models \varphi$ or $SetAx \models \psi$, where φ is a formula of the above form and ψ is any closed formula. This leads to the symmetrical form $SetAx \models \varphi \vee \psi \Leftrightarrow SetAx \models \varphi$ or $SetAx \models \psi$, since the converse direction always holds. We note that $SetAx$ is not complete over all closed formulae. For example, when considering a negative relation like $indiv(x) \equiv \neg set(x)$, we have $SetAx \not\models \exists x\, indiv(x)$ and $SetAx \not\models \neg\exists x\, indiv(x)$, when $\Sigma_{\overline{C}} = \emptyset$. These can be shown, respectively, by using a model of $SetAx$ of pure sets and a model of $SetAx$ containing at least one individual. These models can be constructed from the structure $\mathcal{F}$ in §3.4.

We note a property of the special case of χ having no variables, that is used below. A unifier ζ of such χ can only be $true$ since $Var(\zeta) \subseteq Var(\chi) = \emptyset$. Thus, from Thm. 4.5.1, we have that $\Omega = \emptyset \Leftrightarrow SetAx \models \neg\chi$ and $\Omega = \{true\} \Leftrightarrow SetAx \models \chi$. Put another way, χ is equivalent to $true$ or $false$ in $SetAx$.

The following properties hold when matching constraints are involved. Note that the solved forms considered below are in the context of matching.

When χ is a matching constraint, $SetAx$ can be shown to be a complete theory over an additional form of formulae over that considered in Theorem 4.5.1(iv). The broader form of formulae is needed in a later chapter, and is $\exists(\chi \wedge \delta)$ or $\neg\exists(\chi \wedge \delta)$, where δ is as given in the next proposition.

Proposition 4.5.2 Let $\varphi \equiv \exists(\chi \wedge \delta)$, where χ is a matching constraint, δ is a conjunction of negative set predicate literals, and $Var(\delta) \subseteq Var(\chi)$. Let $\delta \equiv L_1 \wedge \cdots \wedge L_m$, $m \geq 0$, with each L_i being of the form $\neg set(s)$, $s \neq t$, or $s \notin t$.

(i) There are solved forms $\zeta_1, \ldots, \zeta_k$, $k \geq 0$, such that $SetAx \models \forall(\chi \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$

(ii) There are solved forms $\zeta_1, \ldots, \zeta_k$, $k \geq 0$, such that $SetAx \models \forall(\chi \wedge \delta \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$

(iii) $SetAx \models \varphi$ or $SetAx \models \neg\varphi$

Proof: (i): This is just a result of Thm. 4.4.3, repeated here for the sake of completeness. Take $\{\zeta_1, \ldots, \zeta_k\}$ to be the set of solved forms obtained in Thm. 4.4.3.

(ii): Let $\Omega' = \{\zeta'_1, \ldots, \zeta'_l\}$ be the set of solved forms obtained in Thm. 4.4.3. Each ζ'_i, when applied to δ, makes $\zeta'_i\delta$ a conjunction of ground literals. Forgetting the negations in $\zeta'_i\delta$ and solving each positive ground atom will show it to be equivalent to *true* or *false*. It follows that $\zeta'_i\delta$ is equivalent to *true* or *false*. Accordingly, ζ'_i is retained or omitted to form the required set $\{\zeta_1, \ldots, \zeta_k\}$. This idea is formalised below.

Form $\Omega = \{\zeta \mid \zeta \in \Omega', SetAx \models \zeta\delta\}$ and let $\Omega = \{\zeta_1, \ldots, \zeta_k\}$. We have $\zeta\delta \equiv \zeta L_1 \wedge \cdots \wedge \zeta L_m$, and each $\neg\zeta L_i$ as *true* or *false* in $SetAx$. Therefore, we get $\zeta\delta$ to be *true* or *false*, i.e., $SetAx \models \zeta\delta$ or $SetAx \models \neg\zeta\delta$.

($\rightarrow$): Assume $\chi \wedge \delta$. Therefore $\zeta'_1 \vee \cdots \vee \zeta'_l$ by part (i). Case (ζ'_i): Hence $SetAx \models \zeta'_i\delta$, by predicate substitutivity. Then, $\zeta'_i \in \Omega$ and $\zeta'_i \equiv \zeta_j$ for some $1 \leq j \leq k$. Hence, over all cases, $\zeta_1 \vee \cdots \vee \zeta_k$.

($\leftarrow$): We show $\zeta_i \rightarrow \chi \wedge \delta$, i.e., $\zeta_i \rightarrow \chi$ and $\zeta_i\delta$, for $1 \leq i \leq n$. Assume ζ_i. Therefore, χ, since $\zeta_i \in \Omega'$ and by Thm. 4.5.1(i). Also, $SetAx \models \zeta_i\delta$, since $\zeta_i \in \Omega$.

(iii): In (ii) above, we have the following cases. Case (1): $k = 0$. Therefore, $SetAx \models \neg\exists(\chi \wedge \delta)$, i.e., $SetAx \models \neg\varphi$. Case (2): $k > 0$. Therefore, $SetAx \models \forall(\zeta_1 \rightarrow \chi \wedge \delta)$, i.e., $\zeta_1(\chi \wedge \delta)$ by the logic of equality. Hence, $SetAx \models \exists(\chi \wedge \delta)$, i.e., $SetAx \models \varphi$. ■

A property about matchers and complete sets of matchers that is readily shown using the completeness property of $SetAx$ (in Prop. 4.5.2(iii) or in Prop. 4.5.1(iv)) is as below.

Proposition 4.5.3 Let χ be a matching constraint, $\Omega = \{\zeta_1, \ldots, \zeta_k\}$, $k \geq 1$, a complete set of matchers of χ, and ξ a matcher of χ. Then ξ is a variant of ζ_i for some $1 \leq i \leq k$, i.e., $SetAx \models \forall(\xi \leftrightarrow \zeta_i)$.

Proof: We have $SetAx \models \forall(\chi \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$ by Prop. 4.5.2(i) and $SetAx \models \forall(\xi \rightarrow \chi)$ by meaning of matchers. Hence, ξ is subsumed by Ω on $Var(\chi)$, i.e., $SetAx \models \forall(\xi \rightarrow \zeta_1 \vee \cdots \vee \zeta_k)$. So $SetAx \models \xi\zeta_1 \vee \cdots \vee \xi\zeta_k$, where each $\xi\zeta_1$ is ground, since $Dom(\xi) = Var(\chi) = Dom(\zeta_i)$, $1 \leq i \leq k$. By completeness property of $SetAx$, $SetAx \models \xi\zeta_1$ or ...or $SetAx \models \xi\zeta_k$. Hence, $SetAx \models \forall(\xi \rightarrow \zeta_i)$. Also, $\zeta_i\xi$ is the same as $\xi\zeta_i$. This gives $SetAx \models \forall(\zeta_i \rightarrow \xi)$. ■

Since the domains of ξ and ζ_i in the above proposition are the same, it follows that the corresponding elements of the ranges of ξ and ζ_i are equal in $SetAx$. Hence ξ can be considered the same as ζ_i.

5 Herbrand $\cong$-Interpretations

As is usual in logic programming, we seek to focus on just one kind of interpretation, the Herbrand-like interpretations. Such interpretations have a fixed domain and fixed functional assignment for the constructor symbols. The domain is based on a quotient structure formed from the set of ground terms modulo a relation that is equality between ground terms. In the context of programming, the domain contains the elemental data objects. The predicate interpretations are allowed to vary, leading thereby to different interpretations. In this chapter, we lay out these details for the theory $SetAx$.

In particular, we define the Herbrand $\cong$-structure, comprising of the domain and the set predicate interpretations. We use the notation $\cong$ to denote that the domain is a quotient structure. We show that the 'commutativity' and 'idempotency' of the set constructors is indeed adequate for deriving the quotient structure, as commonly assumed in the literature. We also fix the set predicate interpretations since they are not arbitrary predicates. Finally, it is of interest to know whether the Herbrand $\cong$-structure models $SetAx$, and we show this to be the case.

5.1 Herbrand $\cong$-Structure

We define the Herbrand $\cong$-structure $\mathcal{H} = \langle U_{\cong}, \Sigma_P, \Sigma_C \rangle$ below. We do not consider the defined symbols because they are given in terms of the primitive symbols, and are therefore not elemental in nature.

Definition 5.1.1 Let s, t be ground terms. The relation $\cong$ is given by:

$$s \cong t \text{ iff } SetAx \models s = t$$

Here, extensionality can cause syntactically different ground terms to be related — for example, $\{1/\{2/\emptyset\}\} \cong \{2/\{1/\emptyset\}\}$. By virtue of $EqAx$, $\cong$ is an equivalence relation. Let $[t]$ denote the equivalence class containing ground term t. A somewhat simpler characterization of the $\cong$-relation is possible, viz., using just $CoIdAx$ and $EqAx$ to define the relation. (This characterization is also what appears in the literature, but without sufficient justification.) In addition to $CoIdAx$ and $EqAx$ we also need some of the $FinSetAx$, since set appears in $CoIdAx$. Let $SetCoIdAx$ be FS1 $\cup$ FS2 $\cup\, CoIdAx \cup EqAx$. The next proposition justifies this more intuitive characterisation of $\cong$.

Proposition 5.1.2 Let s, t be ground terms. Then

$$SetAx \models s = t \;\Leftrightarrow\; SetCoIdAx \models s = t$$

Proof: ($\Leftarrow$) Immediate. ($\Rightarrow$) By induction on s, t. We show just the following case.

Case: $s \equiv \{s_1, \ldots, s_m/\emptyset\}, t \equiv \{t_1, \ldots, t_n/\emptyset\}, m, n \geq 1$.

Let $SetAx \models s_1 = s_{i_1}$, $\ldots$, $SetAx \models s_1 = s_{i_k}$, and $SetAx \not\models s_1 = s_{i_{k+1}}$, $\ldots$, $SetAx \not\models s_1 = s_{i_m}$, where $i_1, \ldots, i_m$ is a permutation of $1, \ldots, m$. We have, $k \geq 1$ since $SetAx \models s_1 = s_1$.

Let $SetAx \models s_1 = t_{j_1}$, $\ldots$, $SetAx \models t_1 = t_{j_l}$, and $SetAx \not\models s_1 = t_{j_{l+1}}$, $\ldots$, $SetAx \not\models s_1 = t_{j_n}$, where $j_1, \ldots, j_n$ is a permutation of $1, \ldots, n$. We have, $l \geq 1$ since $SetAx \models s_1 \in t$, i.e., $SetAx \models s_1 = t_1 \vee \cdots \vee s_1 = t_n$, i.e., $SetAx \models s_1 = t_1$ or $\ldots$or $SetAx \models s_1 = t_n$, by completeness of $SetAx$. We also have $SetAx \models s_1 \notin \{s_{i_{k+1}}, \ldots, s_{i_m}/\emptyset\}$, and $SetAx \models s_1 \notin \{t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\}$. So, $SetAx \models \{s_{i_{k+1}}, \ldots, s_{i_m}/\emptyset\} = \{t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\}$.

By induction hypothesis, we have $SetCoIdAx \models s_1 = s_{i_1}$, $\ldots$, $SetCoIdAx \models s_1 = s_{i_k}$, and $SetCoIdAx \models s_1 = t_{j_1}$, $\ldots$, $SetCoIdAx \models t_1 = t_{j_l}$, and $SetCoIdAx \models \{s_{i_{k+1}}, \ldots, s_{i_m}/\emptyset\} = \{t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\}$.

Therefore, $SetCoIdAx \models \{s_1, \ldots, s_m/\emptyset\} \overset{CI1}{=} \{s_{i_1}, \ldots, s_{i_k}, s_{i_{k+1}}, \ldots, s_{i_m}/\emptyset\}$
$\overset{CI2}{=} \{s_1, s_{i_{k+1}}, \ldots, s_{i_m}/\emptyset\} \overset{I2}{=} \{s_1, t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\} \overset{I2}{=} \{t_{j_1}, t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\}$
$\overset{CI2}{=} \{t_{j_1}, \ldots, t_{j_l}, t_{j_{l+1}}, \ldots, t_{j_n}/\emptyset\} \overset{CI1}{=} \{t_1, \ldots, t_n/\emptyset\}$. ∎

Definition 5.1.3 The domain $U_{\cong}$ of $\mathcal{H}$, called the **Herbrand $\cong$-Universe**, is:

$$U_{\cong} = \{[t] \mid t \text{ a ground term}\}$$

Definition 5.1.4 The constructor interpretations in $\mathcal{H}$ are, for all $c \in \Sigma_C$

$$c^{\mathcal{H}}([t_1], \ldots, [t_n]) = [c(t_1, \ldots, t_n)]$$

It is easily verified that $c^{\mathcal{H}}$ is well-defined.

Proposition 5.1.5 For any ground term t, $t^{\mathcal{H}} = [t]$.

Proof: Trivial, by induction on t. ■

5.2 Set-Predicate Interpretations

In $\mathcal{H}$, we leave the interpretations of the non-set predicate symbols unspecified since we want them to depend upon the logic program at hand. However, the set predicate interpretations are constrained to satisfy $SetAx$, and so, here we focus on them.

Definition 5.2.1 Let p be set, $\in$, or $=$, and $t_1, \ldots, t_n$ be ground terms. Then,

$$p^{\mathcal{H}}([t_1], \ldots, [t_n]) \text{ iff } SetAx \models p(t_1, \ldots, t_n)$$

It is easily verified that $p^{\mathcal{H}}([t_1], \ldots, [t_n])$ is well-defined. The definition is motivated by our seeking the least relations that might model $SetAx$. For example, the quotient universe construction and the above definition gives identity as the interpretation of equality, i.e., $[s] = [t]$ iff $[s] =^{\mathcal{H}} [t]$, for ground terms s, t. Additional justification for the above set-predicate interpretations is provided by the following proposition.

Proposition 5.2.2 Based on the above $U_{\cong}$ and constructor interpretations, the set predicate interpretations that model $SetAx$ are unique.

Proof: This follows from the completeness of $SetAx$ over the ground set predicate atoms. Let $\mathcal{H}_1$, $\mathcal{H}_2$ be two structures with the above domain $U_{\cong}$ and above constructor interpretations, but with different set-predicate interpretations; and let $\mathcal{H}_1$, $\mathcal{H}_2$ both model $SetAx$. Let p be set, $\in$, or $=$, and $t_1, \ldots, t_n$ be ground terms. Now if $SetAx \models p(t_1, \ldots, t_n)$ then $p^{\mathcal{H}_1}([t_1], \ldots, [t_n])$ and $p^{\mathcal{H}_2}([t_1], \ldots, [t_n])$ must both hold. So for $\mathcal{H}_1$ and $\mathcal{H}_2$ to differ in their set-predicate interpretations, there must be a p and $t_1, \ldots, t_n$ such that $SetAx \not\models p(t_1, \ldots, t_n)$ but $p^{\mathcal{H}_1}([t_1], \ldots, [t_n])$ and not $p^{\mathcal{H}_2}([t_1], \ldots, [t_n])$ hold. However, then $SetAx \models \neg p(t_1, \ldots, t_n)$, i.e., not $p^{\mathcal{H}_1}([t_1], \ldots, [t_n])$. Contradiction. ∎

We will subsequently show that $\mathcal{H}$ models $SetAx$. The above proposition shows that the set-predicate interpretations given in $\mathcal{H}$ are the only possible ones that can lead to a model of $SetAx$ (based on the fixed universe and constructor interpretations).

It is convenient to abbreviate $p^{\mathcal{H}}([t_1], \ldots, [t_n])$ as $[p(t_1, \ldots, t_n)]$, for a predicate symbol p and ground terms $t_1, \ldots, t_n$; and, for S a set of ground atoms, to let $[S]$ denote $\{[A] \mid A \in S\}$.

The following propositions are useful in the next section.

Proposition 5.2.3 The interpretation of set in $\mathcal{H}$ is:

(i) not $[set(scons(s_1, \ldots, scons(s_m, c(\vec{s'})) \cdots))]$, $\quad m \geq 0, c \in \Sigma_{\overline{C}}$,

(ii) $[set(\{s_1, \ldots, s_m/\emptyset\})]$, $\quad m \geq 0$.

Proof: Straightforward. Let $p \equiv set$. To show $[p(\vec{t})]$, show $SetAx \models p(\vec{t})$. To show not $[p(\vec{t})]$, show $SetAx \models \neg p(\vec{t})$. If $SetAx \models \neg p(\vec{t})$, by consistency of $SetAx$, $SetAx \not\models p(\vec{t})$. ∎

Proposition 5.2.4 The interpretations of $\in$ and $=$ in $\mathcal{H}$ are given (mutually recursively) below. (Symmetric cases are omitted.)

($\in$.1) not $[s \in \emptyset]$,

($\in$.2) not $[s \in c(\vec{t'})]$, $\quad c \in \Sigma_{\overline{C}}$,

($\in$.3) not $[s \in scons(t_1, \ldots, scons(t_m, c(\vec{t'})) \cdots)]$, $\quad m \geq 1, c \in \Sigma_{\overline{C}}$,

($\in$.4) $[s \in \{t_1, \ldots, t_m/\emptyset\}] \Leftrightarrow [s = t_1]$ or $\cdots$ or $[s = t_m]$, $\quad m \geq 1$.

(=.1) $[\emptyset = \emptyset]$,

(=.2) not $[\emptyset = c(\vec{t'})]$, $\quad c \in \Sigma_{\overline{C}}$,

(=.3) not $[\emptyset = scons(t_1, \ldots, scons(t_m, c(\vec{t'})) \cdots)]$, $\quad m \geq 1, c \in \Sigma_{\overline{C}}$,

(=.4) not $[\emptyset = \{t_1, \ldots, t_m/\emptyset\}]$, $\quad m \geq 1$,

(=.5) not $[c(\vec{s'}) = d(\vec{t'})]$, $\quad c \not\equiv d, c, d \in \Sigma_{\overline{C}}$,

(=.6) $[c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)] \Leftrightarrow [s_1 = t_1]$ and $\cdots$ and $[s_n = t_n]$, $c \in \Sigma_{\overline{C}}$,

(=.7) not $[c(\vec{s'}) = scons(t_1, \ldots, scons(t_m, d(\vec{t'})) \cdots)]$, $\quad m \geq 1, c, d \in \Sigma_{\overline{C}}$,

(=.8) not $[c(\vec{s'}) = \{t_1, \ldots, t_m/\emptyset\}]$, $\quad m \geq 1, c \in \Sigma_{\overline{C}}$

(=.9) not $[scons(s_1, \ldots, scons(s_m, c(\vec{s'})) \cdots) = scons(t_1, \ldots, scons(t_n, d(\vec{t'})) \cdots)]$,

$m, n \geq 1, m \neq n, c, d \in \Sigma_{\overline{C}}$,

(=.10) $[scons(s_1, \ldots, scons(s_m, c(\vec{s'})) \cdots) = scons(t_1, \ldots, scons(t_m, d(\vec{t'})) \cdots)] \Leftrightarrow$

$[s_1 = t_1]$ and $\cdots$ and $[s_m = t_m]$ and $[c(\vec{s'}) = d(\vec{t'})]$, $\quad m \geq 1, c, d \in \Sigma_{\overline{C}}$,

(=.11) not $[scons(s_1, \ldots, scons(s_m, c(\vec{s'})) \cdots) = \{t_1, \ldots, t_n/\emptyset\}]$, $\quad m, n \geq 1, c \in \Sigma_{\overline{C}}$,

(=.12) $[\{s_1, \ldots, s_m/\emptyset\} = \{t_1, \ldots, t_n/\emptyset\}] \Leftrightarrow$

$[s_1 \in \{t_1, \ldots, t_n/\emptyset\}]$ and $\cdots$ and $[s_m \in \{t_1, \ldots, t_n/\emptyset\}]$ and

$[t_1 \in \{s_1, \ldots, s_m/\emptyset\}]$ and $\cdots$ and $[t_n \in \{s_1, \ldots, s_m/\emptyset\}]$, $\quad m, n \geq 1$.

Proof: Straightforward, by induction on s, t. We use the consistency of $SetAx$ and completeness of $SetAx$ on the ground set predicate atoms. We show a representative case.

($\in$.4): $SetAx \models s \in \{t_1, \ldots, t_m/\emptyset\} \leftrightarrow s = t_1 \vee \cdots \vee s = t_m$

So, $SetAx \models s \in \{t_1, \ldots, t_m/\emptyset\} \Leftrightarrow SetAx \models s = t_1 \vee \cdots \vee s = t_m$

$\Leftrightarrow SetAx \models s = t_1$ or $\ldots$or $SetAx \models s = t_m$, by completeness. $\blacksquare$

In $\mathcal{H}$, while the set predicate interpretations are fixed, the non-set predicate interpretations are allowed to vary. This leads to different interpretations, and the following definition gives a suitable way to specify them.

Definition 5.2.5 The **Herbrand $\cong$-Base**, $B_{\cong}$, is,

$$B_{\cong} = \{[A] \mid A \text{ a ground atom with initial symbol not a set predicate}\},$$

and a **Herbrand $\cong$-interpretation** is a subset of $B_{\cong}$.

We do not include set predicate atoms in $B_{\cong}$, because their interpretations in $\mathcal{H}$ are fixed and therefore can be factored out from consideration.

5.3 Herbrand $\cong$-Interpretations model SetAx

In the sequel, we will be interested in Herbrand $\cong$-models of $P \cup SetAx$ for logic programs P. Hence it is useful to show that Herbrand $\cong$-interpretations model $SetAx$. It is not immediate that Herbrand $\cong$-interpretations model $SetAx$, since $SetAx$ is not all in definite clause form. Examples are axioms FS5 and M3.

We first show a correspondence between $\mathcal{H}$ and $\mathcal{F}$, viz., that $\mathcal{H}$ is isomorphic to $\mathcal{F}$, where $\mathcal{F}$ is the model of $SetAx$ constructed previously in §3.4. Let the correspondence be given by $(\bullet)^\circ\colon U_{\cong} \longrightarrow U$, where

$$[\emptyset]^\circ = \{\},$$

$$[c(t_1, \ldots, t_n)]^\circ = c([t_1]^\circ, \ldots, [t_n]^\circ), \quad \text{for } c \in \Sigma_{\overline{C}},$$

$$[scons(t_1, \ldots, scons(t_m, c(\vec{t'})) \cdots)]^\circ = scons([t_1]^\circ, \ldots, scons([t_m]^\circ, [c(\vec{t'})]^\circ) \cdots),$$

$$\text{for } c \in \Sigma_{\overline{C}} \text{ and } m \geq 1,$$

$$[\{t_1, \ldots, t_m/\emptyset\}]^\circ = \{[t_1]^\circ, \ldots, [t_m]^\circ\}, \quad \text{for } m \geq 1.$$

We need to show that $(\bullet)^\circ$ is well-defined. Clearly, by induction on t, we have $[t]^\circ \in U$. That $[s] = [t] \Rightarrow [s]^\circ = [t]^\circ$ follows from the next lemma.

Lemma 5.3.1 $[s] = [t] \Leftrightarrow [s]^\circ = [t]^\circ$.

Proof: Straightforward, by induction on s, t. For each case of s, t, we show $[s] = [t] \Rightarrow [s]^\circ = [t]^\circ$ and $[s] \neq [t] \Rightarrow [s]^\circ \neq [t]^\circ$. We use the fact that $[s] = [t] \Leftrightarrow [s = t]$. We also use Prop. 5.2.4, definition of $(\bullet)^\circ$, and properties of the structure $\mathcal{F}$. ■

Theorem 5.3.2 Excluding the non-set predicate interpretations, $\mathcal{H}$ is isomorphic to $\mathcal{F}$.

Proof: We show $(\bullet)^\circ$ is bijective. Injectivity follows from the above lemma. Surjectivity follows from the following claim: For all $i \geq 0$, for all $z \in U_i$, there is $[t] \in U_{\cong}$ such that $[t]^\circ = z$. Proof of the claim is straightforward, and proceeds by induction on i.

It is straightforward to show that the constructor interpretations correspond, i.e., $(c^{\mathcal{H}}([t_1], \ldots, [t_n]))^\circ = c^{\mathcal{F}}([t_1]^\circ, \ldots, [t_n]^\circ)$. We take cases of $c \in \Sigma_C$.

We need to show that the set-predicate interpretations correspond, i.e., $p^{\mathcal{H}}([t_1], \ldots, [t_n]) \Leftrightarrow p^{\mathcal{F}}([t_1]^\circ, \ldots, [t_n]^\circ)$. The proofs are straightforward. For $p \equiv set$, we show $[set(s)] \Leftrightarrow set^{\mathcal{F}}([s]^\circ)$ by taking cases on the term structure of s and using Prop. 5.2.3. For $p \equiv =$, we show $[s = t] \Leftrightarrow [s]^\circ =^{\mathcal{F}} [t]^\circ$ by using the above lemma. For $p \equiv \in$, we show $[s \in t] \Leftrightarrow [s]^\circ \in^{\mathcal{F}} [t]^\circ$ by taking cases on the term structure of t, and using Prop. 5.2.4. ■

Theorem 5.3.3 Every Herbrand $\cong$-interpretation models $SetAx$.

Proof: Immediate, by Thm. 3.4.7. ■

Since we have used the consistency of $SetAx$ in proving properties that the above theorem depends on, we cannot substitute it for Thm. 3.4.7 for establishing that $SetAx$ is consistent. However, we have not checked alternate ways of leading to the above theorem without using consistency.

The above theorem justifies the following definitions.

Definition 5.3.4 A **Herbrand $\cong$-model** of a sentence φ is a Herbrand $\cong$-interpretation that models φ.

Definition 5.3.5 Let φ and ψ be sentences. Then ψ is a **Herbrand $\cong$-logical consequence** of φ, denoted by $\varphi \models_{\mathcal{H}} \psi$, if ψ is true in all Herbrand $\cong$-models of φ.

5.4 Ground Terms & Assignments

We now note certain simple and well-known facts about ground terms, ground substitutions, and assignments that are useful later on. Let $\mathcal{J}$ be a structure $\mathcal{J} = \langle D, \Sigma_P, \Sigma_C \rangle$ for some domain D and predicate and constructor symbols Σ_P, Σ_C. We will usually be interested in structures $\mathcal{J}$ that model $SetAx$, and of these, we need only consider the *normal* models, i.e., models that interpret equality as identity (see [Fit90] §8.3 or [Men79] §2.8).

Definition 5.4.1 A quadruple $\langle \sigma, V, \mathbf{A}, \mathcal{J} \rangle$ is an **assignment based on** σ if σ is a ground substitution, V is a set of variables, $\mathbf{A}$ is an assignment in the structure $\mathcal{J}$, $Dom(\sigma) = V$, and for all $x \in V$, $\mathbf{A}$ assigns x to $(\sigma x)^{\mathcal{J}}$.

The interest in assignments based on σ stem from the fact that every assignment in any Herbrand structure, when restricted to the variables of interest, is an assignment based on σ.

Lemma 5.4.2 Let t be a term, $\varphi(\vec{x})$ be a formula, both possibly containing defined symbols. Let $\langle \sigma, V, \mathbf{A}, \mathcal{J} \rangle$ be an assignment based on σ, with $V \supseteq Var(t)$ in the case of t and $V \supseteq \vec{x}$ in the case of $\varphi(\vec{x})$. Then,

(i) $t^{\mathbf{A},\mathcal{J}} = (\sigma t)^{\mathcal{J}}$

(ii) $\varphi(\vec{x})^{\mathbf{A},\mathcal{J}} = \varphi(\sigma \vec{x})^{\mathcal{J}}$

Proof: When t, φ do not contain defined symbols, then the above are well-known properties in first-order logic (e.g., [Fit90] §5.3). When they do contain defined symbols, then the defining equivalences can be used to establish the same.

For example, consider $t = y \leftrightarrow \rho$, where y is a new variable not in t and ρ is obtained by eliminating all defined symbols in $t = y$. Thus $\sigma t = y \leftrightarrow \sigma\rho$. Without loss of generality, let $\mathbf{A}$ assign y such that $(t = y)^{\mathbf{A},\mathcal{J}}$ is true. Then, $(t = y)^{\mathbf{A},\mathcal{J}} = \rho^{\mathbf{A},\mathcal{J}} = (\sigma\rho)^{\mathcal{J}} = (\sigma t = y)^{\mathbf{A},\mathcal{J}}$. Hence, $t^{\mathbf{A},\mathcal{J}} = y^{\mathbf{A},\mathcal{J}}$ and $(\sigma t)^{\mathcal{J}} = y^{\mathbf{A},\mathcal{J}}$, i.e., $t^{\mathbf{A},\mathcal{J}} = (\sigma t)^{\mathcal{J}}$. ■

Thus, if $\langle \sigma, V, \mathbf{A}, \mathcal{J} \rangle$ is an assignment based on σ, then $\{x \mid \varphi(x, \vec{v})\}^{\mathbf{A},\mathcal{J}} = \{x \mid \varphi(x, \sigma\vec{v})\}^{\mathcal{J}}$. The advantage of the above lemma is that having σt or $\varphi(\sigma\vec{x})$ in place of t or $\varphi(\vec{x})$ respectively allows us to exploit equivalences about σt or $\varphi(\sigma\vec{x})$ that are logical consequences of $SetAx$.

In the case of the Herbrand $\cong$-structure $\mathcal{H}$, every assignment is based on some ground substitution σ and every ground substitution σ corresponds to an an assignment $\mathbf{A}$ (when restricted to the variables of interest). Hence we have $t^{\mathbf{A},\mathcal{H}} = [\sigma(t)]$, $\varphi(\vec{x})^{\mathbf{A},\mathcal{H}} = \varphi(\sigma\vec{x})^{\mathcal{H}}$, and evaluating $(\forall\vec{x}\varphi(\vec{x}))^{\mathbf{A},\mathcal{H}}$ or $(\exists\vec{x}\varphi(\vec{x}))^{\mathbf{A},\mathcal{H}}$ is the same as evaluating ground instances $\varphi(\sigma\vec{x})^{\mathcal{H}}$. Thus, $(\forall\vec{x}\varphi(\vec{x}))^{\mathbf{A},\mathcal{H}}$ holds iff all ground instances $\varphi(\sigma\vec{x})^{\mathcal{H}}$ hold, and $(\exists\vec{x}\varphi(\vec{x}))^{\mathbf{A},\mathcal{H}}$ holds iff some ground instance $\varphi(\sigma\vec{x})^{\mathcal{H}}$ holds.

We will also need to evaluate objects like $(\bigcup t)^{\mathcal{J}}$, $(\cup_{i=1}^{n} t_i)^{\mathcal{J}}$ for ground terms t, and t_i, $1 \leq i \leq n$. For these cases we note the following obvious lemma.

Lemma 5.4.3 Let t, and t_i, $1 \leq i \leq n$, be ground terms with $GroundIsSet(t)$.

(i) There is a ground term s such that $SetAx^- \models \bigcup t = s$.

(ii) There is a ground term s such that $SetAx^- \models \cup_{i=1}^{n} t_i = s$

Proof: (i): Let $t \equiv \{t'_1, \ldots, t'_m/\emptyset\}$, $m \geq 0$. Let $t''_1, \ldots, t''_l$ be exactly the terms among $t'_1, \ldots, t'_m$ that are sets, i.e., $GroundIsSet(t''_j)$, $1 \leq j \leq l$. Let $t''_j \equiv \{s_{j1}, \ldots, s_{jk_j}/\emptyset\}$, $1 \leq j \leq l$, $k_j \geq 0$. Then take $s = \{s_{11}, \ldots, s_{1k_1}, \ldots, s_{l1}, \ldots, s_{lk_l}/\emptyset\}$. Again, this follows simply from the definition of $\bigcup t$. If $t \equiv \emptyset$, i.e., $m = 0$, then the above s reduces to $s = \emptyset$.

(ii): We have $\cup_{i=1}^{n} t_i = \bigcup\{t_1, \ldots, t_n/\emptyset\}$. The rest follows by (i). ∎

From above we get that $(\bigcup t)^{\mathcal{J}} = s^{\mathcal{J}}$ and $(\cup_{i=1}^{n} t_i)^{\mathcal{J}} = s^{\mathcal{J}}$, for the appropriate ground term s.

6 The Class of Subset-Logic Languages

We will define a sequence of subset-logic languages progressively increasing in expressivity by the consideration of additional syntactic or semantic features. Certain topics common to the discussion of all these languages are collected together and described here in this chapter. These topics include the basic syntactic elements of these languages, their logical and semantic framework including the *flattening transformation* and the *collect-all assumption*. Also, the common conventions followed for all these languages are established here. The conventions of the previous chapter carry over, except when they are in conflict with the present ones, whereupon the latter take precedence.

6.1 Syntactic Elements

Here we give the basic syntactic definitions for the subset-logic programming languages. We begin with defining the alphabet.

Definition 6.1.1 An **alphabet** Σ is the union of the following disjoint sets of symbols.

(1) the countably infinite set Σ_V of **variables**,

(2) the countable set Σ_F of (**user-defined**) **function** symbols,

(3) the countable set Σ_C of (**data**) **constructor** symbols,

(4) the countable set Σ_P of **predicate** symbols,

(5) the set Σ_{dis} of **distinguished predicate symbols** $\{\approx, \gtrsim\}$,

(6) the set of **connectives** $\{\vee, \wedge, \neg, \leftarrow, \leftrightarrow, \exists, \forall\}$, and

(7) the set of **punctuation symbols** $\{(,), ,\}$.

The above alphabet is the same as that of a first-order language, with $\Sigma_F \cup \Sigma_C$ being the usual function symbols, and $\Sigma_P \cup \Sigma_{dis}$ being the usual predicate symbols. Associated with each symbol in Σ_F, Σ_C, Σ_P, and Σ_{dis} is a natural called the **arity** of that symbol, and if the arity of a symbol is n, it is said to be **n-ary**. 0-ary constructors are called **constants**, and 0-ary predicate symbols are just proposition symbols.

The sets in (5)–(7) are the same for each alphabet, while the sets in (1)–(4) can vary from alphabet to alphabet. However, Σ_C must contain the symbols $\{\emptyset, scons, dscons\}$ to represent set data, and Σ_P must contain the symbols $\{=, set, \in\}$ to represent equality and set predicates. Let us denote $\Sigma\backslash\{\emptyset, scons, dscons\}$ by $\Sigma_{\overline{C}}$ (as before) and $\Sigma_P\backslash\{=, set, \in\}$ by $\Sigma_{\overline{P}}$. We distinguish the symbols in Σ_{dis} since we will treat them differently from other predicate symbols. Intuitively, they can be regarded as being close to the equality and superset relations.

We will adopt certain symbols together with their subscripted or primed forms as typical of certain syntactic categories. The typical symbols for the categories of the alphabet are:

Σ_F : $f, g, h, f_0, g', \ldots$

Σ_D : $a, b, c, d, a_0, b', \ldots$

Σ_P : $p, q, r, p_0, q', \ldots$

Σ_V : $u, v, w, x, y, z, u_0, v', \ldots$

The intended meaning of these categories of symbols is clear from logic programming. The only difference is our distinction between (user-defined) function and (data) constructor symbols. (Henceforth we will simply call them function and constructor symbols respectively.) Constructor symbols are used to build structured data objects. Function symbols are used to specify user-desired input-output relations on data objects.

Next, we define several syntactic categories for a given alphabet Σ. Their intended meanings are described in the next subsection.

Definition 6.1.2 The set of **terms** $TERM$ is the least set satisfying the following: Every variable is a term. If $t_1, \ldots, t_n$ are terms and c is a constructor symbol of arity $n \geq 0$, then $c(t_1, \ldots, t_n)$ is a term.

Definition 6.1.3 The set of **expressions** $EXPR$ is the least set satisfying the following: Every variable is an expression. If $e_1, \ldots, e_n$ are expressions and f is a function symbol of arity $n \geq 0$, then $f(e_1, \ldots, e_n)$ is an expression. If $e_1, \ldots, e_n$ are expressions and c is a constructor symbol of arity $n \geq 0$, then $c(e_1, \ldots, e_n)$ is an expression.

Clearly $TERM \subseteq EXPR$, and an expression without any function symbols is a term.

Definition 6.1.4 If $e_1, \ldots, e_n$ are expressions and p is a predicate symbol of arity $n \geq 0$, then $p(e_1, \ldots, e_n)$ is an **atom**.

We will use the infix forms $e_1 \approx e_2$ and $e_1 \supsetsim e_2$ rather than the prefix forms of these atoms. Here, in both cases, e_1 is called the *lhs* and e_2 the *rhs* of the atom. Also 0-ary objects like X() will be abbreviated to X.

Definition 6.1.5 A **literal** is an atom or the negation of an atom. A **positive literal** is an atom, and a **negative literal** is the negation of an atom.

Definition 6.1.6 The **initial symbol** of an expression, an atom, or a literal is the following. The initial symbol of a variable x is x. The initial symbol of an expression or an atom of the form $X(\ldots)$ is X. The initial symbol of a literal is the initial symbol of its atom.

Definition 6.1.7 A **program clause** is of the form:

$$A \leftarrow L_1, \ldots, L_n$$

where A is an atom and $L_1, \ldots, L_n$ are literals, for $n \geq 0$. A is called the **head** of the program clause. If A is $e_1 \approx e_2 \leftarrow$ or $e_1 \supsetsim e_2 \leftarrow$, then the program clause is also called an **equational clause** or a **subset clause** respectively; otherwise it is called a **relational clause**. If $n = 0$, the program clause is called a **fact** or an **assertion** or an **unit clause**. The unit clauses $e_1 \approx e_2 \leftarrow$ and $e_1 \supsetsim e_2 \leftarrow$ are also called an **equational assertion** and a **subset assertion** respectively. All other unit clauses $p(e_1, \ldots, e_n) \leftarrow$ are called **relational assertions**. If $n > 0$, the program clause is also called a **rule** or a **conditional assertion**, with an equational or subset clause being called a **conditional equational assertion** or a **conditional subset assertion** respectively.

Definition 6.1.8 A **goal clause** is of the form

$$\leftarrow L_1, \ldots, L_n$$

for $n \geq 0$. We have the **empty clause** when $n = 0$. Associated with a goal clause $\leftarrow L_1, \ldots, L_n$ is the **query** $L_1, \ldots, L_n$ and vice-versa.

Definition 6.1.9 A **clause** is a program clause or a goal clause. Given a clause $A \leftarrow L_1, \ldots, L_n$, (in which A may be absent), $L_1, \ldots, L_n$ is called the **body** of the clause, and a literal in its body is called a **subgoal**.

Definition 6.1.10 A **program** is a finite set of program clauses.

Definition 6.1.11 A **subset-logic language** is the set of all programs. Programming in this paradigm is called **subset-logic programming**.

The typical symbols for the syntactic categories above are:

term	:	$s, t, s_0, t', \ldots$	query	:	$Q, Q_0, Q', \ldots$
expression	:	$e, e_0, e', \ldots$	goal clause	:	$G, G_0, G', \ldots$
atom	:	$A, A_0, A', \ldots$	empty clause	:	$\Box$
literal	:	$L, L_0, L', \ldots$	body	:	$B, B_0, B', \ldots$
clause	:	$C, C_0, C', \ldots$	head of a (program) clause C	:	C^+
program	:	$P, P_0, P', \ldots$	body of a clause C	:	C^-

Whether a clause C means a program clause or a goal clause will be clear from the context.

6.2 Programs as Logical Formulae

We now roughly discuss the intended meanings of the syntactic categories described above. A complete account is given in §6.5. Here, in this section, we have omitted consideration of two factors, viz., that expressions may be undefined, and that the intended usage of *dscons* has to be made explicit.

The set of *terms* represent data structures as usual in logic programming. Constants represent atomic data objects, variables represent arbitrary data objects, and constructors (of arity > 0) build structured data objects. *Expressions* too represent data objects, obtained by applying the transformations denoted by the function symbols on input data objects. In terms and expressions, the constructor *dscons* is used whenever the intended usage is that its first argument should not belong to its second argument.

An *atom* specifies that a certain relation, named by its initial symbol, holds among its constituent data objects. A *negative literal* says the opposite — that the relation specified by its atom does *not* hold. A *clause* of the form $A \leftarrow L_1, \ldots, L_n$ is regarded as the logical formula $\forall(A \leftarrow L_1 \wedge \ldots \wedge L_n)$. A *program clause* $A \leftarrow L_1, \ldots, L_n$ thus states that A holds if L_1 and ...and L_n jointly hold, for all possible valuations of the variables in the clause. A *fact* $A \leftarrow$ is an unconditional assertion of A, for all valuations of its variables.

A *program* is regarded as a formula that is the conjunction of its program clauses. It can be seen as defining those relations in the heads of its clauses in terms of all the relations appearing in the program. A *query* $L_1, \ldots, L_n$ is interpreted as the logical formula $\exists(L_1 \wedge \cdots \wedge L_n)$, i.e., there is a valuation of its variables for which L_1 and ...and L_n holds. The query, in the context of a program, asks if there is a valuation of its variables for which the statement L_1 and ...and L_n follows from the program. A *goal clause* $\leftarrow L_1, \ldots, L_n$ says the opposite of its query, that is, for all valuations of its variables, it is not the case that L_1 and ...and L_n holds. An *empty clause* is to be understood as a contradiction.

6.3 The Class of Subset-Logic Languages

The definitions of a subset-logic language and its syntactic constituents are more general than we require. We will need only certain restricted forms in the languages we discuss, obtained by placing restrictions on the forms of atoms, clauses, and programs. Let Σ be some fixed but unspecified alphabet. Each subset-logic language will be based on Σ and will be determined by the restrictions placed on the syntactic components of section §6.1.

Let $\mathbf{L}$ be the first-order language (i.e., the set of *wffs*) based on Σ. Then, each subset-logic language of our study is a subset of $\mathbf{L}$. The language at the lowest level is the **subset-equational language** $\mathbf{L}_0$, and the language at the next level is the **subset-relational language** $\mathbf{L}_1$. Beyond these are **language with negation** $\mathbf{L}_2$, **language with choice operations** $\mathbf{L}_3$, and **language with infinite objects** $\mathbf{L}_4$; however, these do not form the subject of this dissertation. Without getting technical, we describe below the main features encapsulated in these languages.

In $\mathbf{L}_0$, a program clause is restricted to being either an equational assertion of the form $f(\vec{t}) \approx e \leftarrow$ or a subset assertion of the form $f(\vec{t}) \gtrsim e \leftarrow$. With $\mathbf{L}_1$ we begin our investigation of the interaction of relational atoms with equational and subset atoms. Its constructs have been chosen so as to bring out the basic issues involved in integrating relational forms. In it, a program clause has one of the following forms: an equational assertion $f(\vec{t}) \approx e \leftarrow$, a conditional subset assertion $f(\vec{t}) \gtrsim e \leftarrow p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})$, or a conditional relational assertion $q(\vec{t}) \leftarrow p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})$. In $\mathbf{L}_2$ we also allow negative subgoals. In $\mathbf{L}_3$ we additionally have choice operations such as element selection and lazy set generation, and in $\mathbf{L}_4$ we deal with infinite objects such as infinite sets and infinite lists. However, $\mathbf{L}_2$ to $\mathbf{L}_4$ are not covered in this dissertation.

The SuRE language is a specific subset-logic language that is an implemented version of the ideas embodied in subset-logic programming. Hence we may loosely call the programs in this study as SuRE programs. Below we describe the SuRE alphabet and certain syntactic conventions. In giving our program examples in the sequel, we will follow any of these systems of representation: the alphabet Σ with its typical symbols, or the SuRE alphabet and its conventions, or some mix of both — whichever of them proves convenient.

We choose a Prolog-like alphabet for SuRE. The alphabet is based on identifiers that are composed of finite strings of letters, digits, and the underscore character. We have:

Σ_V consists of all identifiers beginning with an uppercase letter.

Σ_F consists of all identifiers beginning with a lowercase letter.

Σ_C consists of all identifiers beginning with a lowercase letter, together with all integers, and the constant symbols $[\,], \emptyset$.

Σ_P consists of all identifiers beginning with a lowercase letter.

As usual, disjointness among the sets of symbols can be restored by taking the disjoint union of the sets of symbols in the alphabet.

Some other Prolog-like conventions we will follow for SuRE programs are:

We have the *anonymous variable*, represented by the underscore symbol _;

The familiar $[t_1 \mid t_2]$ replaces the *pair* or *list* constructor $cons(t_1, t_2)$;

scons is replaced by brace symbols as noted before;

Every clause is terminated with a period, and has its $\leftarrow$ replaced with `:-`;

Unit clauses omit their `:-` symbol.

6.4 The Semantic Framework

We intend to give logical semantics to our programs and so view them as well-formed formulae in a logical language. The proof theory of this language contains the computational aspect of the programs while the model theory provides their declarative aspect. An incomplete logical description of the language components was given in §6.2. Here, we attend to the necessary modifications brought about by the two factors omitted there, viz., that expressions may be undefined and that the intended usage of dscons has to be made explicit. The former is dealt with by treating functions as relations (explained below) and by the flattening transformation (described in the next section). The latter is treated by using the disjointness transformation in §3.3.

We shift attention to a first-order language $\mathbf{L}'$ different from the one utilised before, where programs P in the earlier language $\mathbf{L}$ are translated into new formulae in $\mathbf{L}'$. This is the approach taken in [JP89] and is similar to ideas present in the language K-LEAF ([GLMP91]). Such an approach reduces logical consequence to consideration of interpretations over a domain built from ground terms and not ground expressions. As a result, user-defined function symbols are treated as predicate symbols, to reflect their partial nature, and programs are translated to their *flattened* versions in $\mathbf{L}'$, to enable the use of a uniform computation strategy, such as SLD resolution, on both functions and relations.

The language $\mathbf{L}'$ has the same alphabet as $\mathbf{L}$ except that the function symbols Σ_F of $\mathbf{L}$ become predicate symbols $\Sigma_F \times \Sigma_{dis}$ in $\mathbf{L}'$. Also, *dscons* is dropped as a primitive symbol and is taken, instead, as a defined one, as laid out in Chapter 3. The alphabetic components of $\mathbf{L}'$ are then Σ_V, $\Sigma_C \backslash \{dscons\}$, $\Sigma_P \cup (\Sigma_F \times \Sigma_{dis})$, and auxiliary symbols. For instance, an atom $f(t) \approx t'$ in $\mathbf{L}$ is now $f_{\approx}(t, t')$ in $\mathbf{L}'$. Similarly, $f(t) \supsetsim t'$ in $\mathbf{L}'$ is to be read as $f_{\supsetsim}(t, t')$. However, we will retain the more suggestive $f(t) \approx t'$ and $f(t) \supsetsim t'$ in $\mathbf{L}'$ since the intended meanings are very close to the equality and superset relations. To restore the sense that f represents a partial function, we will consider all programs only in the presence of $FunAx$, which consists of the following axiom for each $f \in \Sigma_F$:

$$FunAx\colon \quad y = z \leftarrow f(\vec{x}) \approx y \wedge f(\vec{x}) \approx z \qquad \text{for all } f \in \Sigma_F$$

It is convenient to have the following definition.

Definition 6.4.1 A non-set predicate symbol p, i.e., when $p \in \Sigma_{\overline{P}} \cup (\Sigma_F \times \Sigma_{dis})$, is called a **program predicate**. An atom formed from a program predicate is called a **program atom**.

Once a program or a goal clause has been transformed, first by the flattening and then by the disjointness transformation, one can logically interpret its components such as term, atom, literal, clause, etc. as described in §6.2.

6.5 Flattening

Flattening of a clause involves replacing functional composition in expressions by conjunctions of atoms of the form $f(\vec{t}) \approx x$ in its antecedent. For example, the program clause

$$f(\{x \backslash t\}) \approx \{g(h(x))/h(t)\} \leftarrow$$

becomes the clause

$$f(\{x \backslash t\}) \approx \{y/s\} \leftarrow h(x) \approx z, g(z) \approx y, h(t) \approx s$$

which is to be read as: $f(\{x \backslash t\}) \approx \{y/s\}$ is asserted provided each of $h(x)$, $g(z)$, and $h(t)$ exist. The latter may fail to exist if they have nonterminating behaviour at those arguments, or if they have not been specified at those arguments within the program. Thus a function symbol occurs only at the outermost level of an atom.

A similar transformation is done for goal clauses. For example, the goal clause

$$\leftarrow g(h(\{1,2\}), h(\emptyset)) \approx x$$

becomes the clause

$$\leftarrow h(\{1,2\}) \approx y, h(\emptyset) \approx z, g(y,z) \approx x.$$

The flattened form of a query is just the negation of the flattened goal associated with the query.

If the usage of the $\approx$ symbol were identical to the $=$ symbol, then the flattening transform would be logically equivalent to the original formula. The transform then would simply result from repeated application of the equivalence $\varphi(t) \leftrightarrow (\varphi(x) \leftarrow t = x)$ with φ being the original formula.

As it stands, the transform is just the algorithmic process of converting an equational specification into a relational specification. For example, the equational specification

$$append([\,], z) = [\,]$$
$$append([x|y], z) = [x|append(y, z)]$$

would either intuitively or by flattening be transformed into the relational specification

$$append([\,], z, [\,])$$
$$append([x|y], z, [x|u]) \leftarrow append(y, z, u)$$

The difference with usual relational programming is that the relation symbols in $\Sigma_F \times \Sigma_{dis}$ are constrained by $FunAx$ to behave functionally.

We now make the above transformations precise. We follow the scheme used in K-LEAF.

Definition 6.5.1 The **data context** $\hat{e}$ of an expression e is:

(i) $x \mapsto x$

(ii) $c(e_1, \ldots, e_n) \mapsto c(\hat{e_1}, \ldots, \hat{e_n})$

(iii) $f(e_1, \ldots, e_n) \mapsto *$

where $*$ is a symbol not in the alphabet, and is called a **hole**.

For example, the data context of $c(f(x), d(g(h(x), c_0), y))$ is $c(*, d(*, y))$. A data context involves only constructors and holes, with the holes recording the slots where expressions, with functions for initial symbols, fit in. We have that $\hat{e} \equiv e$ exactly when e is a term and exactly when $\hat{e}$ has no holes.

Definition 6.5.2 The **application of a data context** $\hat{e}$ with n holes ($n \geq 0$) to expressions e_1, ..., e_n, denoted by $\hat{e}[e_1, \ldots, e_n]$, is the expression obtained by replacing the holes in $\hat{e}$ by e_1, ..., e_n from left to right.

For example, $c(*, d(*, y))[x, y] = c(x, d(y, y))$. When $\hat{e}$ has no holes then $\hat{e}[\,] \equiv \hat{e}$. Clearly, any expression e can be expressed as $\hat{e}[e_1, \ldots, e_n]$ in a unique way.

The flattening of a clause creates new conjuncts in the body of the clause, which in turn have to be flattened. The following definitions give this process.

Definition 6.5.3 Flattening of a body literal L, called $flatb(L)$, is given by:

(i) If L is $e \approx y$, where $e \equiv f(e_1, \ldots, e_n)$, y does not occur in e, and

$e_1 = \hat{e_1}[e_{11}, \ldots, e_{1m_1}]$,..., $e_n = \hat{e_n}[e_{n1}, \ldots, e_{nm_n}]$, then $flatb(L)$ is:

$$flatb(e_{11} \approx y_{11}), \ldots, flatb(e_{1m_1} \approx y_{1m_1}), \ldots, flatb(e_{n1} \approx y_{n1}), \ldots,$$

$$flatb(e_{nm_n} \approx y_{nm_n}), f(\hat{e_1}[y_{11}, \ldots, y_{1m_1}], \ldots, \hat{e_n}[y_{n1}, \ldots, y_{nm_n}]) \approx y$$

where $y_{11}, \ldots, y_{1m_1}, \ldots, y_{n1}, \ldots, y_{nm_n}$ are new variables.

(ii) a) If L is $q(e_1, \ldots, e_n)$ with $q \in \Sigma_P$ and

$e_1 = \hat{e_1}[e_{11}, \ldots, e_{1m_1}]$,..., $e_n = \hat{e_n}[e_{n1}, \ldots, e_{nm_n}]$, then $flatb(L)$ is:

$$flatb(e_{11} \approx y_{11}), \ldots, flatb(e_{1m_1} \approx y_{1m_1}), \ldots, flatb(e_{n1} \approx y_{n1}), \ldots,$$

$$flatb(e_{nm_n} \approx y_{nm_n}), q(\hat{e_1}[y_{11}, \ldots, y_{1m_1}], \ldots, \hat{e_n}[y_{n1}, \ldots, y_{nm_n}])$$

where $y_{11}, \ldots, y_{1m_1}, \ldots, y_{n1}, \ldots, y_{nm_n}$ are new variables.

(ii) b) If L is $\neg q(e_1, \ldots, e_n)$ with $q \in \Sigma_P$ then $flatb(L)$ is the same as above except that the last atom $q(\ldots)$ becomes a negative literal $\neg q(\ldots)$ with identical arguments.

We discuss only part (i) of the above definition since the other parts are very similar and have the same considerations.

In part (i) of the definition above, we require that the new variables in $flatb(e_i \approx y_i)$ be distinct from the new variables in $flatb(e_j \approx y_j)$, for $i \neq j$ and $i, j \in \{11, \ldots, 1m_1, \ldots, n1, \ldots, nm_n\}$. The indexing scheme used above for new variables gives a simple way to ensure this, provided the indices are suffixed to any indices present in the top-level atom $e \approx y$. Thus, if $flatb(e \approx y)$ leads to new variables like y_{11}, etc. then $flatb(e_{11} \approx y_{11})$ should lead to new variables y_{1111}, etc.

More precisely, let $\vec{x}$ be the variables in e, and $\vec{y}$ be the rest of the variables in $flatb(e \approx y)$.

Let $i \in \{11, \ldots, 1m_1, \ldots, n1, \ldots, nm_n\}$. The variables in e_i occur among the $\vec{x}$. Let $\vec{y_i}$ be the variables of $flatb(e_i \approx y_i)$ other than the variables in e_i. Then, $\vec{y}$ becomes partitioned as $\vec{y_{11}}$, ..., $\vec{y_{1m_1}}$, ..., $\vec{y_{n1}}$, ..., $\vec{y_{nm_n}}$ and y. This is simply a technical statement that is useful when giving proofs. (Since y itself is among the variables represented by $\vec{y}$, it would have been better to choose a different name for $\vec{y}$, such as $\vec{z}$. However, we use $\vec{y}$ because we run out of variable names later on.)

We note the following property about the form of $flatb(A)$, when $A \equiv f(\vec{e} \approx y)$. Each atom in $flatb(A)$ is of the form $g(\vec{t}) \approx z$, where z does not occur in $\vec{t}$. This follows directly by induction on the number of function symbols in A.

Definition 6.5.4 **Flattening of a clause** C, called $flatc(C)$, is given by:

(i) If C is $f(\vec{t}) \approx e$, where $e = \hat{e}[e_1, \ldots, e_n]$, then $flatc(C)$ is:

$$f(\vec{t}) \approx \hat{e}[y_1, \ldots, y_n] \leftarrow flatb(e_1 \approx y_1), \ldots, flatb(e_n \approx y_n)$$

where y_1, ..., y_n are new variables.

(ii) If C is $f(\vec{t}) \supsetsim e$, where $e = \hat{e}[e_1, \ldots, e_n]$, then $flatc(C)$ is:

$$f(\vec{t}) \supsetsim \hat{e}[y_1, \ldots, y_n] \leftarrow flatb(e_1 \approx y_1), \ldots, flatb(e_n \approx y_n)$$

where y_1, ..., y_n are new variables.

(iii) If C is $\leftarrow e \approx y$, where the initial symbol of e is a function, then $flatc(C)$ is:

$$\leftarrow flatb(e \approx y)$$

(iv) If C is $f(\vec{t}) \approx e \leftarrow L_1, \ldots, L_n$, then $flatc(C)$ is:

$$A \leftarrow flatb(L_1), \ldots, flatb(L_n), B$$

where $A \leftarrow B$ is the flattened clause in (i) above.

Similarly, if C is $f(\vec{t}) \supsetsim e \leftarrow L_1, \ldots, L_n$.

(v) If C is $p(\vec{t}) \leftarrow L_1, \ldots, L_n$, then $flatc(C)$ is:

$$p(\vec{t}) \leftarrow flatb(L_1), \ldots, flatb(L_n)$$

(vi) If C is $\leftarrow L_1, \ldots, L_n$, then $flatc(C)$ is:

$$\leftarrow flatb(L_1), \ldots, flatb(L_n)$$

We note in the above definition that all the new variables introduced upon flattening the clause occur in the body of the flattened clause. These variables are partitioned in disjoint sets amongst the $flatb$s in the body of the flattened clause, such as amongst the $flatb(e_1 \approx y_1)$, $\ldots$, $flatb(e_n \approx y_n)$ for parts (i) and (ii). That is, the new variables chosen in say $flatb(e_1 \approx y_1)$ is different from those in $flatb(e_2 \approx y_2)$. Similarly, new variables used in say $flatb(L_1)$ are different from those in $flatb(L_2)$.

Next is a lemma that follows from $FunAx$.

Lemma 6.5.5 Let $e \equiv f(e_1, \ldots, e_n)$ and e not contain *dscons* symbols. Let $\vec{x}$ be $Var(e)$ and $\vec{y}$ be the rest of the variables in $flatb(e \approx y)$, where y does not occur in e. Then,

$$FunAx, EqAx \models \forall \vec{x}(\exists \vec{y} flatb(e \approx y) \rightarrow \exists! \vec{y} flatb(e \approx y))$$

Proof: By induction on ν, the number of function symbols in e.

Base: $\nu = 1$. We have $flatb(e \approx y) = f(e_1, \ldots, e_n)$ and e_1, $\ldots$, e_n are all terms. From $FunAx$, we get $\forall y, y'(f(e_1, \ldots, e_n) \approx y \wedge f(e_1, \ldots, e_n) \approx y' \rightarrow y = y')$. Hence $\exists! y f(e_1, \ldots, e_n) \approx y$.

Induction step: $\nu \rightarrow \nu + 1$, $\nu \geq 1$. Let $flatb(e \approx y)$ be as in Defn. 6.6.3. Let $I = \{11, \ldots, 1m_1, \ldots, n1, \ldots, nm_n\}$ be an index set. Assume $\exists \vec{y} flatb(e \approx y)$.

As noted above, the $\vec{y}$ can be partitioned as $\vec{y_i}$, $i \in I$, and y. To show uniqueness of $\vec{y}$, we need two copies of $flatb(e \approx y)$, which we represent as $flatb(e \approx y)$ and $flatb(e \approx y)'$, where the only difference between the two is that all variables $\vec{y}$ in the former are replaced by their primed forms in the latter.

Suppose $flatb(e \approx y)$ and $flatb(e \approx y)'$. To show $\vec{y} = \vec{y'}$. From $\exists \vec{y} flatb(e \approx y)$ we get $\bigwedge_{i \in I} \exists \vec{y_i} flatb(e_i \approx y_i)$. Since each e_i has ν or less function symbols, by induction hypothesis, we get $\exists! \vec{y_i} flatb(e_i \approx y_i)$, i.e., $\forall \vec{y_i}, \vec{y_i'}(flatb(e_i \approx y_i) \wedge flatb(e_i \approx y_i)' \rightarrow \vec{y_i} = \vec{y_i'})$. So $\vec{y_i} = \vec{y_i'}$, for all $i \in I$.

Lastly, $f(\hat{e_1}[y_{11}, \ldots, y_{1m_1}], \ldots, \hat{e_n}[y_{n1}, \ldots, y_{nm_n}]) \approx y$ and $f(\hat{e_1}[y'_{11}, \ldots, y'_{1m_1}], \ldots,$

$\hat{e_n}[y'_{n1}, \ldots, y'_{nm_n}]) \approx y'$ gives $y = y'$, from $EqAx$ and $FunAx$. Thus $\exists! \vec{y} flatb(e \approx y)$. ■

Here, $\exists \vec{y} flatb(e \approx y)$ expresses that for a given $\vec{x}$, each subexpression of e is defined, and the lemma expresses that the values of these subexpressions is unique.

Finally, flattening of clauses corresponds to *eager evaluation* or *call by value* evaluation of expressions. For example, a clause like $f(\vec{t}) \approx e$ with initial symbol of e being a function symbol leads to the flattened clause $f(\vec{t}) \approx y \leftarrow flatb(e \approx y)$. This is equivalent to $f(\vec{t}) \approx y \leftarrow flatb(e \approx y), \exists \vec{y} flatb(e \approx y)$ (since $flatb(e \approx y) \rightarrow \exists \vec{y} flatb(e \approx y)$) whose reading is that $f(\vec{t})$ has the value e provided each subexpression of e is defined. A left to right evaluation of the body of the flattened clause would lead to evaluating the arguments of a function call before evaluating the function itself.

6.6 Collect-All Assumption

The use of the subset assertion in programs has an attendant assumption that is embodied in the following principle.

Principle If an expression $e \equiv f(\vec{t})$ in a program is such that $e \supsetsim s_1, \ldots, e \supsetsim s_n$, and there are no other subset assertions for e according to the program, then infer $e \approx \bigcup_{i=1}^{n} s_i$ by the **collect-all assumption**.

The principle will be applied to each subset-logic language to obtain the intended meaning of their programs. The assumption is in the same spirit as the closed-world assumption in logic programming.

7 Subset-Equational Language

In this chapter and the next we treat the lowest level of subset-logic languages. It is called the **subset-equational language** [JP89], and programming in this paradigm is called **subset-equational programming**. To make the presentation less complicated, we have divided it into two parts. In the present chapter, we treat the basic issues of equational and subset assertions. The semantics, however, does not adequately handle database like programs. Their appropriate treatment through an enhanced semantics is given in the next chapter.

Below, we give the syntax of the subset-equational language and the declarative and operational semantics of its programs. We then show the equivalence of the two semantics through soundness and completeness results. The completeness is shown by using an intermediary fixpoint semantics.

7.1 Syntax

Here we give only those definitions of the syntactic components of the subset-equational language that need changes from earlier chapters. Let Σ be an alphabet with $\Sigma_{\overline{P}} = \emptyset$. The following definitions are based on Σ.

Let Σ_F be partitioned into Σ_U and Σ_{nU}. The symbols in Σ_U are used to define functions through subset assertions and possibly equations. A function f is called **collect-all definable** if $f \in \Sigma_U$. The function symbols in Σ_{nU} are used to define functions exclusively through equations, and hence are non-collect-all definable.

Definition 7.1.1 A **program clause** is either an equational assertion of the form $f(\vec{t}) \approx e \leftarrow$ or a subset assertion of the form $f(\vec{t}) \supsetsim e \leftarrow$, for which $Var(e) \subseteq Var(\vec{t})$ and *scons* does not appear in

$\vec{t}$ and *dscons* does not appear in e (though *scons* and *dscons* may appear in e and $\vec{t}$ respectively).

Here, we disallow *scons* on the lhs and *dscons* on the rhs of a program clause for the following reason. Set *arguments* of a function, in its function definition, such as on the lhs of an assertion, are naturally expressed in terms of *dscons*; while, expressions involving sets, represent *values*, such as on the rhs of assertions, and they are naturally expressed in terms of *scons*. A subterm like $\{s_1 \backslash s_2\}$ in the argument on a lhs allows the rhs to be expressed in terms of s_1, s_2 with s_1 not in s_2, thereby avoiding possible non-termination, say, in a recursive call with argument s_2. A subexpression like $\{e_1/e_2\}$ in the rhs allows for e_1 to be in e_2, a natural possibility.

Definition 7.1.2 A set of program clauses P is **non-overlapping** if for every $f(\vec{t}) \approx e$ in P, $\vec{t}$ does not unify with (a variant of) the arguments of any other equational or subset assertion of f in P.

The condition of non-overlapping helps to avoid conflicting definitions of a function at some argument within a set of program clauses. It does not completely eliminate this possibility, however, as we shall see later.

Definition 7.1.3 A **program** is a finite set of program clauses that is non-overlapping.

We will identify a program clause with its atom. Some examples of subset-equational programs are as below. In program $\mathtt{P}_1$, `intersect` defines the intersection of two sets. In program $\mathtt{P}_2$, `perms(x)` is the set of all permutations of the elements of a set `x`, while `distr(x,y)` assumes `y` is a set of lists, and includes `x` at the head of every list in `y`. In $\mathtt{P}_3$, `size` is a function denoting the cardinality of a set and `succ` is a constructor denoting the successor operation.

$\mathtt{P}_1$: $\mathtt{intersect(\{x\backslash\_\},\{x\backslash\_\})} \supsetsim \mathtt{\{x\}}$

$\mathtt{P}_2$: $\mathtt{perms(\emptyset)} \approx \mathtt{\{[\,]\}}$

$\mathtt{perms(\{x\backslash y\})} \supsetsim \mathtt{distr(x,perms(y))}$

$\texttt{distr(x,\{y\backslash\_\})} \supseteq\!\!\approx \texttt{\{[x|y]\}}$

$\texttt{P}_3$: $\texttt{size(}\emptyset\texttt{)} \approx \texttt{0}$

$\texttt{size(\{x\backslash y\})} \approx \texttt{succ(size(y))}$

Definition 7.1.4 A **query** is of the form $e \approx y$ and a **goal clause** is of the form $\leftarrow e \approx y$, for a ground expression e with initial symbol being a function symbol. Also, *dscons* does not appear in e.

Here, since the value of e is being evaluated, any set values involved are naturally expressed using *scons* and not *dscons*.

Let $\mathbf{L}_0$ be the subset-equational language based on Σ, i.e. the set of all subset-equational programs. Let $\mathbf{Q}_0$ and $\mathbf{G}_0$ be, respectively, the set of all subset-equational queries and goals. Clearly, all these sets are recursive.

7.2 Completion

Given a program P, we obtain its *completion*, called $comp(P)$, which is a logical description of P reflecting more accurately its declarative content. We arrive at $comp(P)$ by the steps below.

We first flatten P by flattening each of its clauses, and denote it by P^F. Thus, a clause like $f(\vec{t}) \approx e$ in P becomes $f(\vec{t}) \approx s \leftarrow F$, where s is the term and F is the body introduced by flattening. Let $\vec{x}$ be the variables in $f(\vec{t}) \approx e$, i.e., $\vec{x} = Var(\vec{t})$. Upon flattening certain new variables are introduced, and let $\vec{y}$ be these variables. As noted in §6.5, the $\vec{y}$ occur in F and a subset of them occur in s, and the formula $\exists \vec{y} F$ asserts that the expression e and all its subexpressions are well-defined for a given $\vec{x}$. When e is a term, $F \equiv true$, no new variables are introduced, and clearly e is well-defined. From Lemma 6.6.5 and Defn. 6.6.4(i) we get that $FunAx, EqAx \models \forall \vec{x}(\exists \vec{y} F \rightarrow \exists! \vec{y} F)$. We will refer to this as the uniqueness property of F. Similarly, for a subset assertion in P.

Then we apply the disjointness transform of Defn. 3.2.2 to the clauses of P^F to obtain P^{FD}. This step does not introduce any new variables. Here, a typical clause like $f(\vec{t}) \approx s \leftarrow F$ in P^F is transformed to $f(\tilde{\vec{t}}) \approx s \leftarrow F, D$ where D is $nonmem(\vec{t})$, since $dscons$ does not appear in s and F. Similarly, for a subset clause in P^F.

Next, we apply the collect-all assumption to P^{FD} as follows. Let function f be collect-all definable and suppose that f is defined through m conditional equational assertions and n conditional subset assertions, $m \geq 0$, $n > 1$, where the j-th equational clause $(1 \leq j \leq m)$ and the i-th subset clause $(1 \leq i \leq n)$ have the forms:

$$f(\tilde{\vec{t'_j}}) \approx s'_j \leftarrow F'_j, D'_j$$

$$f(\tilde{\vec{t_i}}) \underset{\sim}{\supset} s_i \leftarrow F_i, D_i$$

In applying the collect-all assumption we are primarily interested in the subset clauses, while the interest in equational clauses is rather a technical one. Let $\vec{x'_j}$ be $Var(\tilde{\vec{t'_j}})$ and $\vec{x_i}$ be $Var(\tilde{\vec{t_i}})$. Let $\vec{y_i}$ be the new variables introduced upon flattening the i-th subset clause. Then, by the logic of equality, we have the i-th clause to be logically equivalent to:

$$f(\vec{v}) \underset{\sim}{\supset} w \leftarrow \exists \vec{x_i}, \vec{y_i}(\vec{v} = \tilde{\vec{t_i}} \wedge w = s_i \wedge F_i \wedge D_i)$$

where $\vec{v}$ and w are new variables to all the n subset assertions about f in P^{FD}. The collect-all assumption gives rise to the following clause for f, called $CAA(f)$, that expresses $f(\vec{v})$ to be the union of the various s_i, $1 \leq i \leq n$.

$$f(\vec{v}) \approx s \leftarrow \mathit{cond\_finset} \wedge \mathit{cond\_args} \wedge \mathit{cond\_welldef}$$

where s, $\mathit{cond\_finset}$, $\mathit{cond\_args}$, and $\mathit{cond\_welldef}$ are as follows.

$$s \equiv \cup_{i=1}^{n} \bigcup \{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}$$

$$= \cup_{i=1}^{n} \bigcup \{s_i \mid \exists \vec{z_i}(\vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}, \quad \text{where } \vec{z_i} = (\vec{x_i}, \vec{y_i}) \backslash Var(s_i).$$

$$\mathit{cond\_finset} \equiv \bigwedge_{i=1}^{n} \exists u(set(u) \wedge \forall w(w \in u \leftrightarrow \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)))$$

$$\mathit{cond\_args} \equiv \neg \exists \vec{x'_1}(D'_1 \wedge \vec{v} = \tilde{\vec{t'_1}}) \wedge \cdots \wedge \neg \exists \vec{x'_m}(D'_m \wedge \vec{v} = \tilde{\vec{t'_m}})$$

$$\mathit{cond\_welldef} \equiv \bigwedge_{i=1}^{n} \forall \vec{x_i}(D_i \wedge \vec{v} = \tilde{\vec{t_i}} \rightarrow \exists \vec{y_i} F_i)$$

Here, the condition $\mathit{cond\_finset}$ expresses that the collections described by the set abstractions exist as finite sets. Hence this condition allows to assert $f(\vec{v}) \approx s$ provided s is a finite set.

The condition *cond_welldef* specifies that all the functions that $f(\vec{v})$ depends upon (these appear in the F_i) are well-defined at the required arguments. If the $F_i \equiv true$, for $1 \leq i \leq n$, then *cond_welldef* is equivalent to $true$.

Next, *cond_args* is a condition that allows $f(\vec{v}) \approx s$ to hold provided $\vec{v}$ does not equal (unify with) any argument of f in any of its equational assertions. This is motivated by two factors. One is that without *cond_args* the collect-all assumption about f and the equational assertions about f might give conflicting definitions at some arguments. The other is to allow $f(\vec{v})$ to be $\emptyset$ when $\vec{v}$ does not equal any of the arguments of f in any of its equational and subset clauses. (From above it is easy to see that for such $\vec{v}$, s evaluates to $\emptyset$, and *cond_args* and *cond_welldef* are equivalent to $true$.)

We note that the free variables of s, *cond_finset*, *cond_args*, *cond_welldef*, and $CAA(f)$ are just $\vec{v}$. More about *cond_finset*, *cond_args*, and *cond_welldef* are explained later. These three conditions are best understood intuitively rather than through their somewhat formidable first-order formulae.

The collect-all assumption applied to P gives $CAA(P)$, the collection of $CAA(f)$ for all collect-all definable f in P.

Definition 7.2.1 The **completion** of a program P, called $comp(P)$, is given by

$$comp(P) = P^{FD} \cup CAA(P) \cup SetAx \cup FunAx$$

Strictly, all the information in the subset clauses of P^{FD} is captured in $CAA(P)$ and hence the subset clauses could be excluded from the definition of $comp(P)$. Nevertheless, we have included them for convenience.

Similar to programs, queries and goals too undergo appropriate transformations. Since *dscons* does not appear in them, only a flattening transformation is needed. A goal clause G of the form $\leftarrow e \approx y$ becomes its flattened version G^F, where the transformation is as given in §6.6.4(iii).

The flattened form of a query is simply the negation of the flattened form of its corresponding goal.

We now give examples $comp(P_1)$ to $comp(P_3)$ of completions of programs. In specifying $comp(P)$ we will only give $P^{FD} \cup CAA(P)$. Within the $CAA(P)$, it is best to drop the antecedents in the $CAA(f)$ to make it more readable. Nevertheless, just for the examples below, we give *cond_args* and *cond_welldef*, omitting *cond_finset*. In light of Lemma 7.2.2 below, *cond_finset* can be dropped in the context of ground terms, the interesting case.

P_1^F: $\quad P_1^F = P_1$

P_1^{FD}: $\quad intersect(\{x/y_1\}, \{x/y_2\}) \supsetsim \{x\} \leftarrow x \notin y_1, x \notin y_2$

$CAA(P_1)$:

$$intersect(v_1, v_2) \approx \bigcup\{\{x\} \mid \exists y_1, y_2(v_1 = \{x/y_1\} \wedge v_2 = \{x/y_2\} \wedge x \notin y_1 \wedge x \notin y_2)\}$$

P_2^F: $\quad perms(\emptyset) \approx \{[\,]\}$

$\quad perms(\{x\backslash y\}) \supsetsim y_2 \leftarrow perms(y) \approx y_1, distr(x, y_1) \approx y_2$

$\quad distr(x, \{y\backslash z\}) \supsetsim \{[x|y]\}$

P_2^{FD}: $\quad perms(\emptyset) \approx \{[\,]\}$

$\quad perms(\{x/y\}) \supsetsim y_2 \leftarrow perms(y) \approx y_1, distr(x, y_1) \approx y_2, x \notin y$

$\quad distr(x, \{y/z\}) \supsetsim \{[x|y]\} \leftarrow y \notin z$

$CAA(P_2)$:

$$perms(v) \approx \bigcup\{y_2 \mid \exists x, y, y_1(v = \{x/y\} \wedge x \notin y \wedge perms(y) \approx y_1 \wedge distr(x, y_1) \approx y_2)\}$$
$$\leftarrow v \neq \emptyset \wedge \forall x, y(x \notin y \wedge v = \{x/y\} \rightarrow \exists y_1, y_2(perms(y) \approx y_1 \wedge distr(x, y_1) \approx y_2))$$

$$distr(v_1, v_2) \approx \bigcup\{\{[x|y]\} \mid \exists z(v_1 = x \wedge v_2 = \{y/z\} \wedge y \notin z)\}$$

P_3^F: $\quad size(\emptyset) \approx 0$

$\quad size(\{x\backslash y\}) \approx succ(z) \leftarrow size(y) \approx z$

P_3^{FD}: $\quad size(\emptyset) \approx 0$

$$size(\{x/y\}) \approx succ(z) \leftarrow size(y) \approx z, x \notin y$$

$CAA(P_3)$: empty

To form the $CAA(P)$ for specific programs P, it is usually easier to use the following steps rather than the general formula, since the latter tends to introduce extra new variables. Consider the clause transformation below.

$$f(x, y, \{1\backslash x\}, [z], x) \gtrsim [x, g(z)] \tag{1}$$

$\Downarrow$ flattening & disjointness: new variables $= \{w\}$

$$f(x, y, \{1/x\}, [z], x) \gtrsim [x, w] \leftarrow g(z) \approx w, 1 \notin x \tag{2}$$

$\Downarrow$ make the lhs a list of distinct variables: new variables $= \{v_1, v_2, v_3\}$

$$f(x, y, v_1, v_2, v_3) \gtrsim [x, w] \leftarrow v_1 = \{1/x\}, v_2 = [z], v_3 = x, g(z) \approx w, 1 \notin x \tag{3}$$

$\Downarrow$ existentially quantify the body with

(vars in lhs of (1) + new vars in (2)) $-$ vars in head of (3) $= \{z\}$

$$f(x, y, v_1, v_2, v_3) \gtrsim [x, w] \leftarrow \exists z(\underbrace{v_1 = \{1/x\}, v_2 = [z], v_3 = x}_{\star}, \underbrace{g(z) \approx w}_{\circ}, \underbrace{1 \notin x}_{\diamond}) \tag{4}$$

$\Downarrow$ place in a set abstraction in toto

$$f(x, y, v_1, v_2, v_3) \approx \bigcup\{[x, w] \mid \exists z(v_1 = \{1/x\}, v_2 = [z], v_3 = x, g(z) \approx w, 1 \notin x)\} \tag{5}$$

Finally,

$$cond\_welldef \equiv \underbrace{\forall z}_{\dagger}(\underbrace{v_1 = \{1/x\}, v_2 = [z], v_3 = x}_{\star}, \underbrace{1 \notin x}_{\diamond} \rightarrow \underbrace{\exists w}_{\ddagger}(\underbrace{g(z) \approx w}_{\circ})) \tag{6}$$

$\dagger$ universal variables = vars in lhs (1) $-$ vars in lhs (5)

$\ddagger$ existential variables = new vars in (2) (from flattening)

We now further discuss the conditions *cond_args*, *cond_welldef*, and *cond_finset* appearing in $CAA(f)$.

It might appear that *cond_args* should be such that $CAA(f)$ asserts a value for $f(\vec{v})$ for exactly those $\vec{v}$ which unify with the lhs of some subset clause of f (rather than for all $\vec{v}$ which do not unify with lhs of any equational clause of f). That the $CAA(f)$ should apply also to those $\vec{v}$

extending beyond the arguments specified in the lhs of subset clauses of f is shown by the following example.

Consider the definition of set intersection $v_1 \cap v_2 = \{x \mid x \in v_1 \wedge x \in v_2\}$. Here, when v_1 or v_2 is the empty set, then the condition $x \in v_1 \wedge x \in v_2$ fails and the set abstraction forms $\emptyset$. The corresponding subset assertion $intersect(\{x \backslash y_1\}, \{x \backslash y_2\}) \supseteq \{x\}$ seeks to make a statement for those v_1, v_2 for which the set abstraction condition will hold, here the arguments $\{x \backslash y_1\}$, $\{x \backslash y_2\}$. Hence, it is appropriate that the collect-all arising from the subset assertion assert a value for *intersect* even for arguments that do not match $\{x \backslash y_1\}$ and $\{x \backslash y_2\}$, and that the value be $\emptyset$. The *cond_args* and set abstraction in $CAA(P_1)$ above indeed achieve this. In this respect, the *cond_args* partially meets the semantics provided by the non-standard *emptiness as failure* assumption made in [JP89].

The *cond_welldef* condition appears unusual, but is really in keeping with the idea of flattening. Just as an equational or subset clause is asserted provided that the subexpressions in its rhs are known to exist, similarly the collected set in $CAA(f)$ is asserted provided all the subexpressions that it depends on is known to exist. In a database context, however, we would want to collect a set even when some subexpression is undefined, excluding the elements arising from this undefined subexpression. Such a semantics is more complicated and is handled in the next chapter.

It is desirable to know if the intuitively appropriate collection defined by the set abstraction in $CAA(f)$ always exists as a finite set. In that case, *cond_finset* can simply be dropped from the $CAA(f)$. We can show this is true when $\vec{v}$ is a tuple of ground terms. It does not seem easy to prove otherwise.

Lemma 7.2.2 Let $CAA(f)$ be as above for a program P and let $\vec{t}$ be a tuple of ground terms with the length of the tuple being the arity of f. Then, for each $1 \leq i \leq n$,

$$SetAx, FunAx \models \exists u(set(u) \wedge \forall w(w \in u \leftrightarrow \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)))$$

Proof: The idea of the proof is that there are only finitely many matches of $\vec{t} = \tilde{\vec{t_i}}$, and for each match for which F_i holds, it must give rise to a unique value for $\vec{y_i}$ on account of $FunAx$. Hence there are only a finite number of valuations of $\vec{x_i}$, $\vec{y_i}$ that satisfy the set abstraction condition, so that it leads to a finite set.

We have $\forall \vec{x_i}(\tilde{\vec{t_i}} = \vec{t} \wedge D_i \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_k)$, $k \geq 0$, from Prop. 4.5.2(ii). Hence

$$\exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i) \leftrightarrow \bigvee_{j=1}^{k} \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge F_i \wedge \zeta_j),$$

$$\text{i.e., } \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i) \leftrightarrow \bigvee_{j=1}^{k} \exists \vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i).$$

We claim that: For all j, $1 \leq j \leq k$, $\exists u_j(set(u_j) \wedge \forall w(w \in u_j \leftrightarrow \exists \vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i)))$. The proof of the claim is as follows. Case (1): $\neg \exists \vec{y_i}(\zeta_j F_i)$. It is easy to see that $\emptyset$ is a witness for u_j. Case (2): $\exists \vec{y_i}(\zeta_j F_i)$. Therefore $\exists ! \vec{y_i}(\zeta_j F_i)$. So we have $\zeta_j F_i$ by existential elimination. We show that $\{\zeta_j s_i\}$ is a witness for u_j in the statement of the claim above. Clearly $set(\{\zeta_j s_i\})$. $(\rightarrow)$: Assume $w \in \{\zeta_j s_i\}$, i.e., $w = \zeta_j s_i$. So, $\exists \vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i)$. $(\leftarrow)$: Assume $\exists \vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i)$. So $w = \zeta_j s_i' \wedge \zeta_j F_i'$, by existential elimination, where s_i' and F_i' are a second copy of s_i and F_i, i.e., have their $\vec{y_i}$ replaced with $\vec{y_i'}$. Now, by $\exists ! \vec{y_i}(\zeta_j F_i)$, we have $\vec{y_i} = \vec{y_i'}$. Therefore $w = \zeta_j s_i$, i.e., $w \in \{\zeta_j s_i\}$.

Hence, upon existential elimination in the claim above, we have for all j, $1 \leq j \leq k$, $set(u_j) \wedge \forall w(w \in u_j \leftrightarrow \exists \vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i))$. The rest follows from the fact that the finite union of sets exists in ZF^-, i.e., $\cup_{j=1}^{k} u_j$ is a witness for u in the statement of the lemma. ■

Finally, we discuss a characteristic of equational assertions. It is possible that $comp(P)$ is inconsistent by virtue of conflicting definitions inherent within some equational clause of P. Take, for example, the clause $f(\{x \backslash y\}) \approx x$. Here, $f(\{1/\{2\}\}) \approx 1$ and $f(\{2/\{1\}\}) \approx 2$ holds, and so $1 = 2$ is obtained from $\{1, 2\} = \{2, 1\}$, $EqAx$, and $FunAx$. However, $1 \neq 2$ holds by $FreeAx$.

Naturally, we want to be concerned only with consistent programs P (by which we mean that their completions $comp(P)$ are consistent). However, the property of a program being consistent is very likely undecidable, and is an issue that has not been explored.

7.3 Declarative Semantics

The properties in this section imitate those of the declarative semantics of definite clause programs in logic programming ([Llo87]). However, the proofs are complicated by the collect-all assumption and the disjointness transformation.

Definition 7.3.1 The **declarative semantics** $\mathcal{D}_P$ of P is given by:

$$\mathcal{D}_P = \{A \mid comp(P) \models A, A \text{ a ground program atom (of the form } f(\vec{t}) \approx t' \text{ or } f(\vec{t}) \gtrsim t')\}$$

We are particularly interested in those atoms in $\mathcal{D}_P$ of the form $f(\vec{t}) \approx t'$, from which one can obtain the behaviour of more complicated ground queries formed from compositions of functions. The interest in atoms of the form $f(\vec{t}) \gtrsim t'$ is technical.

Proposition 7.3.2 Let $A_1, \ldots, A_n$ be ground program atoms, for some $n \geq 0$, and let $Q = \{\leftarrow A_1, \ldots, \leftarrow A_n\}$. Let P be a subset-equational program. If $comp(P) \cup Q$ has a model, then $comp(P) \cup Q$ has a Herbrand $\cong$-model.

Proof: The proof of this proposition is not the same as for definite clause programs, because the $CAA(f)$ is not in and does not seem capable of being put in definite clause form. What rescues it is various factors such as completeness of $SetAx$ over ground set prdicates, the finitary nature of matching in $SetAx$, and the *cond_welldef* condition.

Suppose M is a model of $comp(P) \cup Q$. Without loss of generality, we can take M to be a normal model, i.e., it interprets $=$ as identity. Let $M_{\cong} = \{[A] \mid A \text{ a ground program atom and } A^M \text{ holds}\}$. We claim that $M_{\cong}$ is a Herbrand $\cong$-model of $comp(P) \cup Q$.

Let φ be a formula in $comp(P) \cup Q$. We have the following cases.

Case (1): $\varphi \in Q$. So $\varphi \equiv \leftarrow A_i$, for some i, $1 \leq i \leq n$. Now, not A_i^M, i.e., $[A_i] \notin M_{\cong}$. So $\varphi^{M_{\cong}}$.

Case (2): $\varphi \in SetAx$. We have $M_{\cong} \models SetAx$, by Thm. 5.3.3.

Case (3): $\varphi \in FunAx$. Let $\varphi \equiv y = z \leftarrow f(\vec{x}) \approx y \wedge f(\vec{x}) \approx z$. Let $\sigma\varphi$ be any ground instance of φ.

Assume $[\sigma(f(\vec{x}) \approx y)], [\sigma(f(\vec{x}) \approx z)] \in M_{\cong}$. (We have used Lemma 5.4.2 here, and from hereon, usually, will not mention its use.) So, $(\sigma(f(\vec{x}) \approx y))^M$ and $(\sigma(f(\vec{x}) \approx z))^M$. Hence, $(\sigma(y = z))^M$. Therefore, $(\sigma(y = z))^{M_{\cong}}$ by completeness property of $SetAx$ (Thm. 4.5.1(iv)).

Case (4): $\varphi \in P^{FD}$. So, $\varphi \equiv f(\vec{t}) \approx s \leftarrow F, D$, or $\varphi \equiv f(\vec{t}) \gtrsim s \leftarrow F, D$, where F, D, are as described in §7.2. Let φ be an equational clause. The proof is the same for a subset clause.

Let $\sigma\varphi$ be any ground instance of φ. Assume $(\sigma D)^{M_{\cong}}$ and $(\sigma F)^{M_{\cong}}$. Then $(\sigma D)^M$, by the completeness property of $SetAx$. Also, by construction of $M_{\cong}$, $(\sigma F)^M$. (Each atom in σF is of the form $\sigma(g(\vec{t'}) \approx y)$, and $(\sigma(g(\vec{t'}) \approx y))^M$ holds since $[\sigma(g(\vec{t'}) \approx y)] \in M_{\cong}$.) So, $(\sigma(f(\vec{t}) \approx s))^M$, i.e., $[\sigma(f(\vec{t}) \approx s)] \in M_{\cong}$.

Case (5): $\varphi \in CAA(P)$. Let $\varphi \equiv CAA(f)$, for some f. Let $\sigma\varphi$ be any ground instance of φ with $Dom(\sigma) = \vec{v}$. Assume $(\sigma(\mathit{cond\_finset}))^{M_{\cong}}$, $(\sigma(\mathit{cond\_args}))^{M_{\cong}}$ and $(\sigma(\mathit{cond\_welldef}))^{M_{\cong}}$. We will show $(\sigma(\mathit{cond\_finset}))^M$, $(\sigma(\mathit{cond\_args}))^M$ and $(\sigma(\mathit{cond\_welldef}))^M$.

Clearly, $(\sigma(\mathit{cond\_finset}))^M$, by Lemma 7.2.2, since M models $SetAx$ and $FunAx$.

We have $(\neg\exists\vec{x'_j}(D'_j \wedge \sigma\vec{v} = \tilde{\vec{t'_j}}))^{M_{\cong}}$, for $1 \le j \le m$. Therefore, $(\neg\exists\vec{x'_j}(D'_j \wedge \sigma\vec{v} = \tilde{\vec{t'_j}}))^M$, for $1 \le j \le m$, by the completeness property of $SetAx$, Prop. 4.5.2(iii), i.e., $(\sigma(\mathit{cond\_args}))^M$.

Now, $\forall\vec{x_i}(D_i \wedge \sigma\vec{v} = \tilde{\vec{t_i}} \leftrightarrow \zeta_1 \vee \cdots \vee \zeta_{k_i})$, for some $k_i \ge 0$, $1 \le i \le n$, by Prop. 4.5.2(ii). So, $\sigma(\mathit{cond\_welldef}) \leftrightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k_i} \exists\vec{y_i}\zeta_j F_i$. The following hold for each i, $1 \le i \le n$, and each j, $1 \le j \le k_i$.

We have $(\exists\vec{y_i}\zeta_j F_i)^{M_{\cong}}$. So there is a ground substitution θ_j with domain $\vec{y_i}$ such that $(\theta_j(\zeta_j F_i))^{M_{\cong}}$. Therefore $(\theta_j(\zeta_j F_i))^M$, by construction of $M_{\cong}$. (Each atom in $\theta_j(\zeta_j F_i)$ is of the form $\theta_j(\zeta_j(g(\vec{t'}) \approx y))$, and we get $(\theta_j(\zeta_j(g(\vec{t'}) \approx y)))^M$ from $[\theta_j(\zeta_j(g(\vec{t'}) \approx y))] \in M_{\cong}$.) Thus $(\exists\vec{y_i}\zeta_j F_i)^M$. Therefore $(\sigma(\mathit{cond\_welldef}))^M$.

Next, we evaluate the set abstraction under M and $M_{\cong}$. The following hold for each i, $1 \leq i \leq n$.

Let $\psi_i(\sigma\vec{v}) \equiv \exists\vec{x_i}, \vec{y_i}(w = s_i \wedge \sigma\vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)$. We have,

$$\psi_i(\sigma\vec{v}) \leftrightarrow \bigvee_{j=1}^{k_i} \exists\vec{x_i}, \vec{y_i}(w = s_i \wedge F_i \wedge \zeta_j) \qquad \text{using Prop. 4.5.2(ii)}$$

$$\leftrightarrow \bigvee_{j=1}^{k_i} \exists\vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i) \qquad \text{by logic of equality}$$

From *cond_welldef*, we have $(\exists\vec{y_i}\zeta_j F_i)^M$, $(\exists\vec{y_i}\zeta_j F_i)^{M_{\cong}}$, $(\theta_j(\zeta_j F_i))^M$, and $(\theta_j(\zeta_j F_i))^{M_{\cong}}$, for $1 \leq j \leq k_i$.

Let $t_i = \bigcup_{j=1}^{k_i}\{\theta_j(\zeta_j s_i)\} = \bigcup_{l=1}^{k_i}\{\theta_l(\zeta_l s_i)\} = \{\theta_1(\zeta_1 s_i), \ldots, \theta_{k_i}(\zeta_{k_i} s_i)/\emptyset\}$. Here t_i is a ground term. (Note that t_i and $\vec{t_i}$ are different. Also, we have used two index variables j and l for the same expression as preparation for below.)

We will show that $(\{w \mid \psi_i(\sigma\vec{v})\} = t_i)^{M_{\cong}}$ and $(\{w \mid \psi_i(\sigma\vec{v})\} = t_i)^M$ hold. It is easy to see, from the definition of t_i, that $t_i \subseteq \{w \mid \psi_i(\sigma\vec{v})\}$ is true under M and $M_{\cong}$. That $\{w \mid \psi_i(\sigma\vec{v})\} \subseteq t_i$ is true under M and $M_{\cong}$ follows from *cond_welldef* and the uniqueness property of F_i. We give details of these below.

First, it is convenient to obtain the following equivalences under $SetAx$.

$$\{w \mid \psi_i(\sigma\vec{v})\} \subseteq t_i \leftrightarrow \forall w(\psi_i(\sigma\vec{v}) \rightarrow w \in t_i) \qquad \text{by Lemma 7.2.2}$$

$$\leftrightarrow \forall w(\bigvee_{j=1}^{k_i} \exists\vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i) \rightarrow \bigvee_{l=1}^{k_i}(w = \theta_l(\zeta_l s_i)))$$

$$\leftrightarrow \bigwedge_{j=1}^{k_i} \forall w(\exists\vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i) \rightarrow \bigvee_{l=1}^{k_i}(w = \theta_l(\zeta_l s_i))) \qquad \text{by clausal logic}$$

$$\leftrightarrow \bigwedge_{j=1}^{k_i} \forall w\forall\vec{y_i}(w = \zeta_j s_i \wedge \zeta_j F_i \rightarrow \bigvee_{l=1}^{k_i}(w = \theta_l(\zeta_l s_i))) \qquad \text{by quantifier logic}$$

$$\leftrightarrow \bigwedge_{j=1}^{k_i} \forall\vec{y_i}(\zeta_j F_i \rightarrow \bigvee_{l=1}^{k_i}(\zeta_j s_i = \theta_l(\zeta_l s_i))) \qquad \text{by logic of equality}$$

It is enough to show $\forall\vec{y_i}(\zeta_j F_i \rightarrow \zeta_j s_i = \theta_j(\zeta_j s_i))$ is true under M and $M_{\cong}$, for $1 \leq j \leq k_i$.

The following hold for each j, $1 \leq j \leq k_i$. Let $\mathbf{B}$ be any assignment of $\vec{y_i}$ in M that makes $(\zeta_j F_i)^{\mathbf{B},M}$ hold. (If not $(\zeta_j F_i)^{\mathbf{B},M}$, then we are done.) By the uniqueness property of F_i, it is easy to see that $\mathbf{B}$ is an assignment based on θ_j in the sense of Defn. 5.4.1. (We have $(\exists\vec{y_i}\zeta_j F_i)^M$ and $(\theta_j(\zeta_j F_i))^M$. So, $(\exists!\vec{y_i}\zeta_j F_i)^M$, and therefore $\mathbf{B}$ is based on θ_j.) Now, $(\zeta_j s_i = \theta_j(\zeta_j s_i))^{\mathbf{B},M}$ holds since $(\theta_j(\zeta_j s_i) = \theta_j(\zeta_j s_i))^M$ always holds, and by using Lemma 5.4.2(ii).

Therefore $\forall \vec{y_i}(\zeta_j F_i \rightarrow \zeta_j s_i = \theta_j(\zeta_j s_i))$ is true under M, and the same argument shows it is true under $M_{\cong}$.

Finally, we have $(f(\sigma\vec{v}) \approx \cup_{i=1}^n \bigcup\{w \mid \psi_i(\sigma\vec{v})\})^M$ is $(f(\sigma\vec{v}) \approx \cup_{i=1}^n \bigcup t_i)^M$ which holds, since M models $CAA(f)$. Now, by Lemma 5.4.3, we know there is a ground term s such that $SetAx \models \cup_{i=1}^n \bigcup t_i = s$. So, $(f(\sigma\vec{v}) \approx s)^M$ holds, and therefore, $[f(\sigma\vec{v}) \approx s] \in M_{\cong}$. Hence, $\varphi^{M_{\cong}}$, since $(f(\sigma\vec{v}) \approx \cup_{i=1}^n \bigcup\{w \mid \psi_i(\sigma\vec{v})\})^{M_{\cong}}$ is $[f(\sigma\vec{v}) \approx s]$. ■

Theorem 7.3.3 $comp(P) \models A \Leftrightarrow comp(P) \models_{\mathcal{H}} A$, where A is a ground program atom.

Proof: The theorem statement is equivalent to

$$comp(P) \cup \{\leftarrow A\} \text{ has a model} \Leftrightarrow comp(P) \cup \{\leftarrow A\} \text{ has a Herbrand } \cong\text{-model},$$

and the rest follows by the above proposition. ■

Proposition 7.3.4 Let I be an index set, and $\{M_\alpha \mid \alpha \in I\}$ be a non-empty collection of Herbrand $\cong$-models of $comp(P)$. Then $\bigcap\{M_\alpha \mid \alpha \in I\}$ is a Herbrand $\cong$-model of $comp(P)$.

Proof: The proof closely parallels that of Prop. 7.3.2. Let $M_P = \bigcap\{M_\alpha \mid \alpha \in I\}$ and let φ be a formula in $comp(P)$. Then M_P has the role of $M_{\cong}$ and M_α the role of M in that proof, and instead of the construction of $M_{\cong}$, we use the definition of M_P to deduce analogous properties. We show just the case of $\varphi \in P^{FD}$ below.

Case: $\varphi \in P^{FD}$. So $\varphi \equiv f(\vec{t}) \approx s \leftarrow F, D$ or $\varphi \equiv f(\vec{t}) \gtrsim s \leftarrow F, D$. Here F, D are as described in §7.2. Let φ be an equational clause. The proof is the same for a subset clause.

Let $\sigma\varphi$ be any ground instance of φ. It also serves as an asignment for each M_α, $\alpha \in I$, since the universe is fixed.

Assume $(\sigma D)^{M_P}$ and $(\sigma F)^{M_P}$. Then $(\sigma D)^{M_\alpha}$, $\alpha \in I$, by the completeness property of $SetAx$. Also, by definition of M_P, $(\sigma F)^{M_\alpha}$, $\alpha \in I$. (Each atom in σF is of the form $\sigma(g(\vec{t'}) \approx y)$, and $[\sigma(g(\vec{t'}) \approx y)] \in M_\alpha$, $\alpha \in I$, since $[\sigma(g(\vec{t'}) \approx y)] \in M_P$.) So, $[\sigma(f(\vec{t}) \approx s)] \in M_\alpha$, $\alpha \in I$, i.e., $[\sigma(f(\vec{t}) \approx s)] \in M_P$. ■

The above model-intersection property justifies the existence of a least Herbrand $\cong$-model M_P of $comp(P)$, provided $comp(P)$ is consistent. Here M_P is as defined in the above proof. (By Prop. 7.3.2, $comp(P)$ is consistent imples it has a non-empty collection of Herbrand $\cong$-models.)

Theorem 7.3.5 $M_P = [\mathcal{D}_P]$, for any consistent subset-equational program P.

Proof: Let A be a ground program atom. We have

$[A] \in [\mathcal{D}_P] \Leftrightarrow A \in \mathcal{D}_P \Leftrightarrow comp(P) \models A$

$\Leftrightarrow comp(P) \models_{\mathcal{H}} A$ by Thm. 7.3.3

$\Leftrightarrow [A]$ is in every Herbrand $\cong$-model of $comp(P)$

$\Leftrightarrow [A] \in M_P$ ■

We note that in the collect-all clause, the condition *cond_welldef* plays an essential part in the development of the declarative semantics above. For example, it is crucial to the proof of Prop. 7.3.2 case (5). Also, in its absence Prop. 7.3.4 fails to hold, as evidenced by the counterexample below. The Prop. 7.3.4 is used later in the fixed-point semantics in Thm. 7.6.12. Hence, *cond_welldef* is a condition that cannot be dropped from the definition of the collect-all clause.

Consider P and $comp(P)$ below, where *cond_welldef* is absent from $comp(P)$. Assume f is collect-all definable and g is not.

P: $f(x) \supsetsim g(1)$

P^{FD}: $f(x) \supsetsim y \leftarrow g(1) \approx y$

$CAA(P)$:

$f(x) \supsetsim y \leftarrow g(1) \approx y$

$f(x) \approx \bigcup\{y \mid g(1) \approx y\}$

Consider the collection of Herbrand models of this alternate version of $comp(P)$, amongst which are the models M_1 and M_2 below.

$$M_1 = \{[g(1) \approx \{1\}], [f(1) \gtrsim \{1\}], [f(2) \gtrsim \{1\}], \ldots,$$
$$[f(1) \approx \{1\}], [f(2) \approx \{1\}], \ldots\}$$
$$M_2 = \{[g(1) \approx \{2\}], [f(1) \gtrsim \{2\}], [f(2) \gtrsim \{2\}], \ldots,$$
$$[f(1) \approx \{2\}], [f(2) \approx \{2\}], \ldots\}$$

It follows that M_P the intersection of all Herbrand models is just $\emptyset$. (If it were not empty an atom in it would be in both M_1 and M_2; but $M_1 \cap M_2 = \emptyset$.) Now $\bigcup\{y \mid g(1) \approx y\}$ evaluates to $\emptyset$ in M_P. However, M_P is not a model of P since it does not contain $[f(1) \approx \emptyset]$, $[f(2) \approx \emptyset]$, etc.

Lastly, for arbitrary queries (that are not necessarily ground atoms), we have the following declarative notion of a correct answer.

Definition 7.3.6 Let $P \in \mathbf{L}_0$ and $G \in \mathbf{G}_0$ with Q its corresponding query. Let θ be a substitution with $Dom(\theta) = Var(G^F)$. Then θ is a **correct answer** for $P \cup \{G\}$ if $comp(P) \models \forall \theta Q^F$.

It is not hard to see that if θ is a correct answer, then it is a ground substitution. Hence, $\forall \theta Q^F \equiv \theta Q^F$.

7.4 Matching

In computing with clauses and queries, we need to match the ground terms of queries with the lhs terms of clauses. The latter terms do not contain *scons* but may contain *dscons*. For this reason, it is useful to discard duplicates within the ground terms of queries before matching. Accordingly, we give a procedure $NoDups$ that removes duplicate elements from ground terms. After that, we give a rewriting procedure for the above-mentioned matching, and it is the basic computational component in the operational semantics of subset-equational programs. This procedure is different from the rewriting in §4.3 in that matching with dscons terms is involved and not unification with scons terms. We prove that the algorithm terminates and produces a correct, complete, and minimal set of matches. Recall our convention from §3.3 that, unless otherwise stated, terms will be understood to contain only primitive constructor symbols but no defined constructor symbols such as *dscons*.

The following procedure $NoDups(s)$ acts on a ground term s and returns a ground term that is s with all duplicates elements removed from each of its subterms. When duplicate elements are removed from a ground term, it may be specified using *dscons* instead of *scons*. For example, $NoDups(\{1, c(\{2, 2/\emptyset\}), 1/\emptyset\}) = \{1, c(\{2\backslash\emptyset\})\backslash\emptyset\}$. Recall our convention from §3.1 that $\{t_1, t_2, \ldots, t_n\backslash t_{n+1}\}$ denotes $\{t_1\backslash\{t_2\backslash\cdots\backslash\{t_n\backslash t_{n+1}\}\cdots\}$.

$$NoDups(c(t_1, \ldots, t_n)) = c(NoDups(t_1), \ldots, NoDups(t_n)) \quad \text{for } c \not\equiv scons,\ n \geq 0 \tag{1}$$

$$NoDups(\{s_1, \ldots, s_m/c(\vec{s'})\}) = \{NoDups(s_1)\backslash NoDups(\{s_2, \ldots, s_m/c(\vec{s'})\})\} \quad \text{for } c \in \Sigma_{\overline{C}},\ m \geq 1 \tag{2}$$

$$NoDups(\{s_1, \ldots, s_m/\emptyset\}) = NoDups(\{s_2, \ldots, s_m/\emptyset\}) \quad \text{for } s_1 \in \{s_2, \ldots, s_m/\emptyset\},\ m \geq 1 \tag{3}$$

$$NoDups(\{s_1, \ldots, s_m/\emptyset\}) = \{NoDups(s_1)\backslash NoDups(\{s_2, \ldots, s_m/\emptyset\})\} \quad \text{for } s_1 \notin \{s_2, \ldots, s_m/\emptyset\},\ m \geq 1 \tag{4}$$

In the above definition of $NoDups$, by $s_1 \in \{s_2, \ldots, s_m/\emptyset\}$ and $s_1 \notin \{s_2, \ldots, s_m/\emptyset\}$, we mean that these hold in $SetAx$. We can use the rewriting procedure of §4.3 to decide them. Not all the rules there are necessary now, as only ground terms are involved. Below we give the few rules that apply. The matching problem is $K = \{s_i = t_i\}_{i=1}^m \cup \{s_j \in t_j\}_{j=1}^n$ with all terms ground, and the solved form is T (denoting $true$).

(G1) $\quad K \cup \{c(\vec{s}) = d(\vec{t})\} \Rightarrow F$

$$\text{if } c \not\equiv d$$

(G2) $K \cup \{c(s_1, \ldots, s_m) = c(t_1, \ldots, t_m)\} \Rightarrow K \cup \{s_1 = t_1, \ldots, s_m = t_m\}$

if $m \geq 0$, $c \not\equiv scons$

(G3) $K \cup \{\{s_1, \ldots, s_m/\emptyset\} = \{t_1, \ldots, t_n/\emptyset\}\} \Rightarrow K \cup \{s_1 \in t, \ldots, s_m \in t\} \cup \{t_1 \in s, \ldots, t_n \in s\}$

where $s \equiv \{s_1, \ldots, s_m/\emptyset\}$, $t \equiv \{t_1, \ldots, t_n/\emptyset\}$, $m, n \geq 1$.

(G4) $K \cup \{\{s_1/s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{s_1 = t_1, s_2 = t_2\}$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv \emptyset$.

(G5) $K \cup \{s \in t\} \Rightarrow F$

if $init(last(t)) \in \Sigma_{\overline{C}}$ or $t \equiv \emptyset$

(G6) $K \cup \{s \in \{t_1, \ldots, t_n/\emptyset\}\} \Rightarrow K \cup \{s = t_i\}$

for some $1 \leq i \leq n$

Here, rule G6 is nondeterministic. It is easy to see that a nondeterministic application of these rules to K always terminates in either T or F. In particular, $\{s \in t\}$ always results in a tree with some leaf being T iff $SetAx \models s \in t$ and all leaves being F iff $SetAx \models s \notin t$.

Proposition 7.4.1 Let s be a ground term and $t \equiv NoDups(s)$. Then (i) $NoDups(s)$ terminates, (ii) t is a ground term that does not contain *scons* (but may contain *dscons*), (iii) $SetAx \models nonmem(t)$, (i.e., $SetAx \models nonmem(NoDups(s))$), and (iv) $SetAx \models t = s$, (i.e., $SetAx \models NoDups(s) = s$).

Proof: By induction on $size(s)$. The proof is straightforward, by cases (1) to (4), and we show a representative case.

Case (3): We have that $t = NoDups(\{s_2, \ldots, s_m/\emptyset\})$, where $s = \{s_1, \ldots, s_m/\emptyset\}$ and $SetAx \models s_1 \in \{s_2, \ldots, s_m/\emptyset\}$ holds. Using the induction hypothesis, we clearly have (i), (ii), and (iii); and we also have $SetAx \models NoDups(\{s_2, \ldots, s_m/\emptyset\}) = \{s_2, \ldots, s_m/\emptyset\}$. Hence $t = \{s_2, \ldots, s_m/\emptyset\}$. By Prop. 3.3.3(iii), we have $\{s_2, \ldots, s_m/\emptyset\} = \{s_1, \ldots, s_m/\emptyset\}$. Thus we have (iv). ∎

The parts (ii) and (iii) in the above proposition confirms that $NoDups$ removes duplicates, since it is easy to see that $SetAx \models nonmem(t)$ iff t has no duplicates, for ground terms t without $scons$ but possibly $dscons$.

It is useful to define $\tilde{\theta}$ as θ with all its $dscons$ converted to $scons$, i.e., $\tilde{\theta} = \{x \mapsto \tilde{t} \mid x \mapsto t \in \theta\}$. Also, we define $NoDups$ on tuples of terms, i.e., if $\vec{t} = (t_1, \ldots, t_n)$ then $NoDups(\vec{t}) = (NoDups(t_1), \ldots, NoDups(t_n))$.

It is straightforward to extend the above proposition to a tuple of terms, viz., if $\vec{s}$ is ground and $\vec{t} = NoDups(\vec{s})$, then $\vec{t}$ is ground and does not contain $scons$, $SetAx \models nonmem(\vec{t})$, and $SetAx \models \vec{t} = \vec{s}$.

We next give a preparatory lemma about substitutions on the $nonmem$ relation.

Lemma 7.4.2 Let s be a term possibly containing $dscons$. Let θ be a ground substitution such that for each $x \in Dom(\theta)$, θx possibly contains $dscons$ and $SetAx \models nonmem(\theta x)$. Then

$$SetAx \models \theta(nonmem(s)) \leftrightarrow nonmem(\theta s)$$

Proof: Straightforward, by induction on s. We take cases on s as in the definition of $nonmem$. We show just the following case.

Case: $s \equiv \{t_1 \backslash t_2\}$. By using induction hypothesis on t_1, t_2, we have $\theta nonmem(s) = \theta(nonmem(t_1) \wedge nonmem(t_2) \wedge \tilde{t_1} \notin \tilde{t_2}) \leftrightarrow \theta nonmem(t_1) \wedge \theta nonmem(t_2) \wedge \theta t_1 \notin \theta t_2 \leftrightarrow nonmem(\theta t_1) \wedge nonmem(\theta t_2) \wedge \theta t_1 \notin \theta t_2 \leftrightarrow nonmem(\theta s)$. We used Prop. 3.2.1(i) in these equivalences. ■

The above lemma extends easily to a tuple of terms, viz., $SetAx \models \theta(nonmem(\vec{s})) \leftrightarrow nonmem(\theta \vec{s})$.

We now give the rewriting rules for matching dscons terms. We would like the matching problem to be $K = \{s_i = t_i\}_{i=1}^{m}$ with t_i being ground and s_i, t_i not containing $scons$ but possibly containing $dscons$. However, the rule M5 below introduces equalities that do not obey this form of terms and additionally have existential variables. Hence we broaden the matching problem to

$K = \exists \vec{z}\{s_i = t_i\}_{i=1}^{m}$ such that the conditions (i) to (v) below hold.

This is a somewhat different style of formalising the matching problem from that in §4.4. It appears somewhat complicated but the intuition is clear from the rules M1 to M5. Its appeal is that it places no restriction on the selection strategy of choosing the rewrite atom and the proof follows the usual style for such proofs.

(i) s_i, t_i do not contain *scons* but possibly contain *dscons*.

(ii) A variable z is an existential variable iff it occurs once each in exactly two equations in the following manner. In one it occurs on the rhs in the form $\{s\backslash z\}$, s ground. In the other it occurs on the lhs in either of the forms $\{s_1\backslash z\} = t$, t ground, or $z = t$, t ground, or $z = \{s_2\backslash z_1\}$, s_2 ground.

(iii) A rhs term of an equation is either ground or of the form $\{s\backslash z\}$, s ground and z an existential variable.

(iv) The digraph based on K, with the equations of K being the nodes and arrows between each pair of equations containing the same existential variable, is acyclic. The arrow is outward from the equation containing the existential variable on the rhs. (This implies that the digraph forms a set of 0 or more disconnected acyclic chains.)

(v) If each chain, starting with the equation at its head, has the rhs of the equations in its nodes as $\{t_1\backslash z_1\}$, ..., $\{t_n\backslash z_n\}$, t, $n \geq 1$, then $SetAx \models nonmem(\{t_1, \ldots, t_n, \backslash t\})$. If an equation has no existential variables (i.e., it is not part of a chain), and has its rhs as t, then $SetAx \models nonmem(t)$.

An example of a matching problem is

$$\exists z_1, z_2\{c(x_0) = c(1),\ \underbrace{y_1 = \{1\backslash z_1\}},\ \underbrace{z_1 = \{2\backslash z_2\}},\ \underbrace{\{x_1\backslash z_2\} = \emptyset}\}$$

We will never have to use the complicated parts of this definition on actual examples, since we mean to apply the matching only to problems not containing any existential variables. The definition is used only for proving the correctness of the matching.

The solved forms for this matching problem and the definitions of matcher and minimal, complete set of matchers are the same as for the matching problem in §4.4. We recall the definition of solved form and matcher here.

Solved forms are matching problems of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ with all the x_i's distinct. Hence, they may be written as $\mathcal{E}(\theta)$ for ground substitutions θ or simply identified with θ. A matcher ζ for a matching problem χ is a solved form with $Dom(\zeta) = Var(\chi)$ and $SetAx \models \forall(\zeta \rightarrow \chi)$.

The following are the rewriting rules, all of which apply only when the rhs term of the selected equation is ground.

(M1) $K \cup \{x = t\} \Rightarrow \{x \mapsto t\}K \cup \{x = t\}$

if x is a free variable and x occurs in K.

(M2) $K \cup \{c(\vec{s}) = d(\vec{t})\} \Rightarrow F$

if $c \not\equiv d$.

(M3) $K \cup \{c(s_1, \ldots, s_m) = c(t_1, \ldots, t_m)\} \Rightarrow K \cup \{s_1 = t_1, \ldots, s_m = t_m\}$

if $m \geq 0$, $c \not\equiv scons$ but possibly $c \equiv dscons$.

(M4) $K \cup \{\{s_1 \backslash s_2\} = \{t_1 \backslash t_2\}\} \Rightarrow \exists z K \cup \{s_2 = \{t_1 \backslash z\}, \{s_1 \backslash z\} = t_2\}$

if $last(t_2) \equiv \emptyset$. Here, z is a new variable.

(M5) $\exists z K \cup \{z = t\} \Rightarrow \{z \mapsto t\}K$

The above rules are nondeterministic due to M3 and M4, both of which apply for dscons terms.

Theorem 7.4.3 Starting with a matching problem $K = \exists \vec{z}\{s_i = t_i\}_{i=1}^{m}$ and using the rules M1 to M5 repeatedly, under any selection strategy, until none are applicable results in:

(i) A finite tree whose leaves are F or matchers of K.

(ii) The leaves define a complete set of matchers of K. Specifically, if $\vec{s} \equiv (s_1, \ldots, s_m)$, $t \equiv$

$(t_1, \ldots, t_m)$, $\chi \equiv \tilde{\vec{s}} = \tilde{\vec{t}}$, $\delta \equiv nonmem(\vec{s}) \wedge nonmem(\vec{t})$, and $\{\mathcal{E}(\theta_1), \ldots, \mathcal{E}(\theta_k)\}$, $k \geq 0$ is the set of solved forms, then

$$SetAx \models \forall(\exists \vec{z}(\chi \wedge \delta) \leftrightarrow \mathcal{E}(\theta_1) \vee \cdots \vee \mathcal{E}(\theta_k))$$

(iii) The set of matchers of K is also minimal.

Proof: We first prove the various steps involved in a rewriting procedure. Then, based on it, we prove the parts (i) to (iii) above.

Soundness: For each rule, we show that the free variables are preserved, the form of the matching problem is preserved, and that the two sides of the rewriting are logically equivalent thereby making the rules sound and preserving in the solutions to the matching problem. We show only the cases of rules M1, M4, and M5.

Case M1: Clearly the free variables are preserved. Let $\theta = \{x \mapsto t\}$. Since x is a free variable, θK does not affect the ground rhs of equations nor the existential variables in K. Hence the chains remain the same and relevant *nonmem* relations still hold. If x occurs in a term (on the lhs of an equation) containing an existential variable, then the term is of the form $\{s \backslash z\}$ and $\{\theta s \backslash z\}$ is still of the same form. Thus, the form of the matching problem is preserved.

Let $K = \exists \vec{z}\{\vec{s'} = \vec{t'}\}$. The equivalence to be shown is

$$\exists \vec{z}(\tilde{\vec{s'}} = \tilde{\vec{t'}} \wedge nonmem(\vec{s'}) \wedge nonmem(\vec{t'}) \wedge \tilde{x} = \tilde{t} \wedge nonmem(x) \wedge nonmem(t))$$
$$\leftrightarrow \exists \vec{z}(\widetilde{\theta \vec{s'}} = \tilde{\vec{t'}} \wedge nonmem(\theta \vec{s'}) \wedge nonmem(\vec{t'}) \wedge \tilde{x} = \tilde{t} \wedge nonmem(x) \wedge nonmem(t))$$

This is easily shown, using the meaning of *nonmem*, Prop. 3.2.1(i), Lemma 7.4.2, and substitutivity of $EqAx$. For example, in the $(\leftarrow)$ direction, we have $\tilde{x} = \tilde{t}$, $x = t$, $nonmem(\theta \vec{s'})$, $\theta nonmem(\vec{s'})$ and $nonmem(\vec{s'})$, using the mentioned properties.

Case M4: In this case, both rules M3 and M4 apply nondeterministically. In both cases, clearly the free variables are preserved. In M3, if s_2 does not contain any existential varaiable, then the previous chains are unaffected. If s_2 does contain an existential variable, then $\{s_1 \backslash s_2\} \equiv \{s_1 \backslash z_1\}$

and hence $s_2 \equiv z_1$. The previous chain that ended in $\{s_1 \backslash s_2\}$ now ends in the new equation $s_2 = t_2$. So no cycles are created. The difference is that t_1 is now omitted in the chain. Nevertheless, if $nonmem(\{\ldots \backslash \{t_1 \backslash t_2\}\})$ held before then $nonmem(\{\ldots \backslash t_2\})$ holds now, since $set(t_2)$ and omitting t_1 from a set containing no duplicates still leaves no duplicates. In M4, if s_2 does not contain any existential varaiable, then a new chain is created. If s_2 does contain an existential variable, then $s_2 \equiv z_1$ and the previous chain is extended by one. The previous chain that ended in $\{s_1 \backslash s_2\}$ now connects with the new equation $s_2 = \{t_1 \backslash z\}$. So no cycles are created. Clearly, the *nonmem* relations holds, since they held before. It is easy to see, for M3 and M4, that the other requirements of a matching problem are satisfied.

Letting $K = \{\vec{s'} = \vec{t'}\}$, $\chi \equiv \tilde{\vec{s'}} = \tilde{\vec{t'}}$, and $\delta \equiv nonmem(\vec{s'}) \wedge nonmem(\vec{t'})$, we show that

$$
\begin{aligned}
&\exists \vec{z}(\chi \wedge \delta \wedge \widetilde{\{s_1 \backslash s_2\}} = \widetilde{\{t_1 \backslash t_2\}} \wedge nonmem(s_1) \wedge nonmem(s_2) \wedge \tilde{s_1} \notin \tilde{s_2} \\
&\qquad\qquad\qquad\qquad \wedge nonmem(t_1) \wedge nonmem(t_2) \wedge \tilde{t_1} \notin \tilde{t_2}) \\
&\quad \leftrightarrow \exists \vec{z}(\chi \wedge \delta \wedge \tilde{s_1} = \tilde{t_1} \wedge \tilde{s_2} = \tilde{t_2} \wedge nonmem(s_1) \wedge nonmem(t_1) \\
&\qquad\qquad\qquad\qquad \wedge nonmem(s_2) \wedge nonmem(t_2)) \\
&\quad \vee \quad \exists \vec{z}(\chi \wedge \delta \wedge \tilde{s_2} = \widetilde{\{t_1 \backslash z\}} \wedge \widetilde{\{s_1 \backslash z\}} = \tilde{t_2} \wedge nonmem(s_2) \wedge nonmem(t_1) \wedge \tilde{t_1} \notin \tilde{z} \\
&\qquad\qquad\qquad\qquad \wedge nonmem(s_1) \wedge \tilde{s_1} \notin \tilde{z} \wedge nonmem(t_2))
\end{aligned}
$$

($\rightarrow$): Since $set(t_2)$, all scons terms are sets. We take cases of $\tilde{s_1} = \tilde{t_1}$ and $\tilde{s_1} \neq \tilde{t_1}$ to obtain the two disjunctions. In the latter case, $\tilde{s_1} \notin \tilde{s_2} = \widetilde{\{t_1 \backslash z\}}$ gives $\tilde{s_1} \notin \tilde{z}$, and $\tilde{t_1} \notin \tilde{t_2} = \widetilde{\{s_1 \backslash z\}}$ gives $\tilde{t_1} \notin \tilde{z}$.

($\leftarrow$): In the case of the first disjunct, we get $\tilde{s_1} \notin \tilde{s_2}$ from $\tilde{t_1} \notin \tilde{t_2}$. In the case of the second disjunct, we have $\tilde{s_1} \neq \tilde{t_1}$ and $\tilde{s_1} \notin \tilde{s_2}$ as follows. If $\tilde{s_1} = \tilde{t_1}$ then $\widetilde{\{s_1 \backslash z\}} = \widetilde{\{t_1 \backslash z\}} = \tilde{t_2}$ which gives $\tilde{t_1} \in \tilde{t_2}$. Contradiction. If $\tilde{s_1} \in \tilde{s_2} = \widetilde{\{t_1 \backslash z\}}$ then $\tilde{s_1} = \tilde{t_1}$ or $\tilde{s_1} \in \tilde{z}\}$, neither of which holds.

The remaining aspects of the equivalence are easily shown.

Case M5: Clearly the free variables are preserved. Let $\theta = \{z \mapsto t\}$ and let $s = \{t_1 \backslash z\}$

and $z = t$ be the two equations containing z. They form a chain by themselves or are the last link of one. Now θK only affects these two equations. Upon substitution, we get $s = \{t_1 \backslash t\}$. Clearly, $nonmem(\{t_1 \backslash t\})$ holds, since the two equations were the end link of a chain before. It is easy to see that the other requirements of a matching problem are satisfied.

Letting $K = \exists \vec{z}\{\vec{s'} = \vec{t'}\} \cup \{s = \{t_1 \backslash z\}\}$, $\chi \equiv \tilde{\vec{s'}} = \tilde{\vec{t'}}$, and $\delta \equiv nonmem(\vec{s'}) \wedge nonmem(\vec{t'})$, it is easy to show, similar to case M1 that

$$\exists z \exists \vec{z}(\chi \wedge \delta \wedge \tilde{s} = \widetilde{\{t_1 \backslash z\}} \wedge nonmem(s) \wedge nonmem(\{t_1 \backslash z\}) \wedge \tilde{z} = \tilde{t} \wedge nonmem(z) \wedge nonmem(t))$$
$$\leftrightarrow \exists \vec{z}(\chi \wedge \delta \wedge \tilde{s} = \widetilde{\{t_1 \backslash t\}} \wedge nonmem(s) \wedge nonmem(\{t_1 \backslash t\}))$$

Completeness: A normal form K_n is reached when it is either F or when no rule applies or when a rule applies without change to the normal form. When no rule applies, if K_n is nonempty, then it consists of equations having existential variables in its rhs terms. As such, it forms an infinite chain (which has no terminal link since no equation has a ground rhs). Contradiction. Hence $K_n = \emptyset$ and the normal form is T.

When some rule applies without change, it can only be rule M1 with the variable x in $x = t$ not appearing in the rest of K_n. If K_n has any existential variables, then again we get an infinite chain. Hence K_n has no existential variables and is in solved form.

Termination: We form the multiset Φ from the multiset K as

$$\Phi = \{(\nu_s, \eta_t) \mid s = t \in K\}$$

where ν_s is the number of free variables in s and η_t is $\underline{size}(t)$ as defined below on the form of equations ε. The pairs (ν, η) are lexicographically ordered and multisets Φ are ordered by the induced ordering ([DM79]), which is as follows. A multiset Φ_1 is greater than a multiset Φ_2, if for some multisets X, Y, where $\emptyset \neq X \subseteq \Phi_1$, we have $\Phi_2 = (\Phi_1 - X) \cup Y$ and $(\forall y \in Y)(\exists x \in X) x \succ y$. Here, the relation $x \succ y$ is the order relation among the elements of the multisets. In words, a multiset is reduced by the removal of at least one element (those in X) and their replacement with any finite number —

possibly zero — of elements (those in Y), each of which is smaller than one of the elements that have been removed.

$$\underline{size}(t) = \begin{cases} size(t) & \text{if } \varepsilon \equiv s = t,\ t \text{ ground} \\ 1 + size(t_1) + \underline{size}(z) & \text{if } \varepsilon \equiv s = \{t_1 \backslash z\} \end{cases}$$

To obtain $\underline{size}(z)$ we use the following, defined recursively on the length of the chain that ε occurs in. Let ε' be the pair equation for ε, i.e., containing the existential variable z.

$$\underline{size}(z) = \begin{cases} \underline{size}(t_2) & \text{if } \varepsilon' \equiv z = t_2 \\ size(t_2) - 1 & \text{if } \varepsilon' \equiv \{s_2 \backslash z\} = t_2 \end{cases}$$

The second part of $\underline{size}(z)$ reflects that the match for z is bound to be less than $size(t_2)$. In general, given a chain labelled by $\varepsilon_1, \ldots, \varepsilon_{n+1}$ with ε_1 at the head of the chain, we can say that $\varepsilon_1 \equiv s = \{t_1 \backslash z_1\}$, nodes ε_2 to ε_n have the form $\varepsilon_2 \equiv z_1 = \{t_2 \backslash z_2\}, \ldots, \varepsilon_n \equiv z_{n-1} = \{t_n \backslash z_n\}$, and $\varepsilon_{n+1} \equiv z_{n+1} = t$ or $\varepsilon_{n+1} \equiv \{t_{n+1} \backslash z_{n+1}\} = t$, where t is ground in both cases. Then, $\underline{size}(\{t_1 \backslash z_1\})$ is just $size(\{t_1, \ldots, t_n \backslash t\})$ or 1 less than this depending on the form of ε_{n+1}.

Now, for each rule, we show that the rewriting terminates or that Φ reduces in rewriting. We show just the cases M1, M4 and M5.

Case M1: If $x \notin K$, then we are in a normal form since no change occurs, and hence we terminate. When $x \in K$, Φ becomes $(\Phi - X) \cup Y$ where

$X = \{(\nu_s, \eta_t) \mid s = t \in K \text{ and } x \text{ occurs in } s\}$

$Y = \{(\nu_s - 1, \text{any}) \mid (\nu_s, \eta_t) \in X\}$

To see that Φ reduces, take the maximum element of X, say x and show $x > y$ for each $y \in Y$. This follows easily from $x \geq x' > y$, where $x' \in X$ 'corresponds' to $y \in Y$.

Case M4: For the equation $s_2 = \{t_1 \backslash z\}$, the (ν, η) becomes $(\leq, <)$ since $\underline{size}(\{t_1 \backslash z\}) = size(\{t_1 \backslash t_2\}) - 1$. For the equation $\{s_1 \backslash z\} = t_2$, again the (ν, η) becomes $(\leq, <)$ since $size(t_2) <$

$size(\{t_1 \backslash t_2\})$. Hence the Φ decreases since (ν, η) of one equation is replaced by lesser values corresponding to the two new equations.

Case M5: Here, only two equations are affected, say $s = \{t_1 \backslash z\}$ and $z = t$. The (ν, η) for the former is $(\nu_s, size(\{t_1 \backslash t\}))$ and upon rewriting remains the same. Hence the Φ reduces, since the (ν, η) for $z = t$ is excluded.

Fairness: By definition of the control strategy, rules are applied until a normal form is reached.

Finally, we give the proofs of parts (i) to (iii) of the theorem.

(i): This follows from the *completeness* part above. Also the free variables are preserved in rewriting, as shown in the *soundness* part. Hence the free variables in the solved forms are $Dom(\theta) = Var(K)$.

(ii): This equivalence follows from the equivalences shown for each rule in the *soundness* part, by doing an induction on the number β of branching points in the tree of rewritings. If $\beta = 0$, then clearly the equivalence holds, even when $k = 0$, for then the normal form is F.

If $\beta > 0$, consider the first branching point from the root, and apply the induction hypothesis to the two branches. Then we apply the 'disjunctive' equivalence obtained in case M4 in the *soundness* part above.

(iii): To show minimality of the set of solved forms, we again do an induction on the number β of branching points in the tree of rewritings. When the cardinality of the set of solved forms $k \leq 1$, then we are done. When $k > 1$, then $\beta \geq 1$ and we can consider the first branching point from the root. The solved forms obtained from each subtree are minimal by induction hypothesis. Hence it is enough to show that if θ and γ come from the different subtrees, then $SetAx \not\models \mathcal{E}(\theta) \rightarrow \mathcal{E}(\gamma)$.

The branching point arose out of applying rules M3 and M4, and they correspond to the two disjuncts in the rhs of the equivalence shown in case M4 in the *soundness* section above. Without loss of generality, $\mathcal{E}(\theta)$ implies one disjunct and $\mathcal{E}(\gamma)$ implies the other, by part (ii) above. In the proof of case M4 above, we see that in the ($\leftarrow$) direction, one disjunct implies $\tilde{s_1} = \tilde{t_1}$ and the other implies $\tilde{s_1} \neq \tilde{t_1}$. Hence if $SetAx \models \mathcal{E}(\theta) \rightarrow \mathcal{E}(\gamma)$, then we can deduce both $\tilde{s_1} = \tilde{t_1}$ and $\tilde{s_1} \neq \tilde{t_1}$, a contradiction. ■

Corollary 7.4.4 Let $MatchTuple(\vec{s}, \vec{t})$ be the set of substitutions corresponding to the set of solved forms of the matching problem $K = \{\vec{s} = \vec{t}\}$. Let $\chi \equiv \tilde{\vec{s}} = \tilde{\vec{t}}$, $\delta \equiv nonmem(\vec{s})$, $\vec{x} = Var(\vec{s})$, and $MatchTuple(\vec{s}, \vec{t}) = \{\theta_1, \ldots, \theta_k\}$, $k \geq 0$. Then

$$SetAx \models \forall \vec{x}(\chi \wedge \delta \leftrightarrow \mathcal{E}(\theta_1) \vee \cdots \vee \mathcal{E}(\theta_k))$$

In the corollary, $nonmem(\vec{t})$ does not appear since it is known to hold in $SetAx$ by virtue of being part of the matching problem.

7.5 Operational Semantics

The operational semantics is a modification of SLD-resolution ([Llo87]) that allows to compute sets by collecting (unioning) the answers obtained in different paths of a set-collecting computation tree. We call it **SLDU-resolution** for SLD-resolution with Unioning. It computes with the flattened forms of programs and queries and not with their completions. The disjointness transformation to programs is accounted for in the matching while the collect-all assumption to programs is accounted for in the computation of set-collecting trees. In SLDU -resolution, the leftmost atom of a goal clause is always selected for resolution. Thus the order of evaluation in the flattened forms corresponds to innermost-first evaluation order of functions in expressions in their pre-flattened forms. The following example illustrates these ideas.

P_4:	`father(Bob)` $\approx$ `Mark`	`mother(Bob)` $\approx$ `Mary`
	`father(Ann)` $\approx$ `Mark`	`mother(Ann)` $\approx$ `Mary`
	`father(Mark)` $\approx$ `Joe`	`mother(Mark)` $\approx$ `Jane`

$$\texttt{parents(x)} \supseteq\!\!\!\!\!\sim \{\texttt{father(x)}\}$$

$$\texttt{parents(x)} \supseteq\!\!\!\!\!\sim \{\texttt{mother(x)}\}$$

$$\texttt{Q: parents(father(Bob))} \approx \texttt{x}$$

Here, the program $\texttt{P}_4$ describes a small database of family relationships among `Bob`, `Mark`, `Ann`, `Joe`, `Mary` and `Jane`, which are taken as constructors. The query `Q` seeks to compute the paternal grandparents of `Bob`.

$$
\begin{array}{ll}
P_4^F: & father(Bob) \approx Mark \leftarrow \\
& father(Ann) \approx Mark \leftarrow \\
& father(Mark) \approx Joe \leftarrow \\
& mother(Bob) \approx Mary \leftarrow \\
& mother(Ann) \approx Mary \leftarrow \\
& mother(Mark) \approx Jane \leftarrow \\
& parents(x) \supseteq\!\!\!\!\!\sim \{y\} \leftarrow father(x) \approx y \\
& parents(x) \supseteq\!\!\!\!\!\sim \{y\} \leftarrow mother(x) \approx y \\
Q^F: & father(Bob) \approx y, parents(y) \approx x
\end{array}
$$

The computation is done using the above flattened forms of the program and query. The derivation of the computed answer is shown below which depends on the computation of a set-collecting tree.

$$
\begin{array}{l}
\langle \theta_0, G_0 \rangle \equiv \langle \epsilon, \leftarrow father(Bob) \approx y, parents(y) \approx x \rangle \\
\quad \Big| \; \theta = \{y \mapsto Mark\} \\
\langle \theta_1, G_1 \rangle \equiv \langle \theta, \leftarrow parents(Mark) \approx x \rangle \\
\quad \Big| \; \sigma = \{x \mapsto s\} = \{x \mapsto \{Joe, Jane\}\} \quad \text{(from set-collecting tree below)}
\end{array}
$$

$\langle \theta_2, G_2 \rangle \equiv \langle \sigma \circ \theta, \Box \rangle$

So the computed answer is θ_2 which is $\{y \mapsto Mark, x \mapsto \{Joe, Jane\}\}$. The set collecting tree for P_4^F and $\{\leftarrow parents(Mark) \approx x\}$ is given next.

$$\leftarrow parents(Mark) \stackrel{\supset}{\sim} x_0$$

$$\sigma_1 = \{x_1 \mapsto Mark, x_0 \mapsto \{y_1\}\} \qquad \sigma_2 = \{x_2 \mapsto Mark, x_0 \mapsto \{y_2\}\}$$

$$\langle \sigma_1, \leftarrow father(Mark) \approx y_1 \rangle \qquad \langle \sigma_2, \leftarrow mother(Mark) \approx y_2 \rangle$$

$$\sigma_3 = \{y_1 \mapsto Joe\} \qquad \sigma_4 = \{y_2 \mapsto Jane\}$$

$$\langle \sigma_3 \circ \sigma_1, \Box \rangle \qquad \langle \sigma_4 \circ \sigma_2, \Box \rangle$$

answer set $s = \sigma_3 \circ \sigma_1(x_0) \cup \sigma_4 \circ \sigma_2(x_0) = \{Joe, Jane\}$

We now make the above resolution procedure precise by means of the following definitions. We note that, from hereon, objects such as clause, goal, etc. will be with respect to $\mathbf{L}_0'$, where $\mathbf{L}_0'$ is the first-order language derived from $\mathbf{L}_0$, that views functions as predicates as described in §6.5.

Definition 7.5.1 Let θ be an idempotent substitution and G be a goal $\leftarrow A_1, \ldots, A_m$, $m \geq 0$ such that for each $1 \leq i \leq m$ $A_i \equiv f_i(\vec{t_i}) \approx x_i$, and for each variable $y \in Var(\vec{t_i})$, $y \in \{x_1, \ldots, x_{i-1}\}$; also $Dom(\theta) \cap \{x_1, \ldots, x_m\} = \emptyset$. Then $\langle \theta, G \rangle$ is called a **subst-goal pair**.

The goal G in a subst-goal pair $\langle \theta, G \rangle$ is the usual goal that occurs in derivations in logic programming. The θ is the composition of the accumulated susbtitutions from the starting goal upto the goal so far. We tag such substitutions along so as to allow for more succinct statements in subsequent definitions such as in Defn. 7.5.5(f) and 7.5.7(f). Thus, a subst-goal pair $\langle \theta, G \rangle$ is the kind of goal that appears in the steps of SLDU derivations. Note that if $\langle \theta, G \rangle$ is a subst-goal pair with $G \equiv \leftarrow A_1, \ldots, A_m$, $m \geq 0$, then A_1 is an equational atom with its lhs ground.

Definition 7.5.2 Let $\langle \theta_1, G_1 \rangle$ be a subst-goal pair with $G_1 \equiv \leftarrow A_1, \ldots, A_m$, $m > 0$, $A_1 \equiv f(\vec{t_1}) \approx x_1$. Then a subst-goal pair $\langle \theta_2, G_2 \rangle$ is **equationally derived** from $\langle \theta_1, G_1 \rangle$ using (a variant of) an equational clause $C \equiv A' \leftarrow A'_1, \ldots, A'_l$, $l \geq 0$, $A' \equiv f(\vec{t'}) \approx s'$, if there is a substitution $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t_1}))$ such that $\theta_2 = \sigma' \circ \theta_1$, where $\sigma' = \tilde{\sigma} \cup \{x_1 \mapsto \tilde{\sigma} s'\}$, and $G_2 \equiv\leftarrow \sigma'(A'_1, \ldots, A'_l, A_2, \ldots, A_m)$.

In the above definition, it is easy to verify, using the definition of flattening, Defn. 6.6.4, that $\langle\theta_2, G_2\rangle$ is indeed a subst-goal pair. Also, note that σ is *any* one match of the tuple $\vec{t'}$ with $NoDups(\vec{t_1}))$.

Definition 7.5.3 Let $G_1 \equiv\leftarrow f(\vec{t_1}) \supsetsim x_1$ with $\vec{t_1}$ ground. Then a subst-goal pair $\langle\theta_2, G_2\rangle$ is **subset derived** from G_1 using (a variant of) a subset clause $C \equiv A' \leftarrow B'$, $A' \equiv f(\vec{t'}) \supsetsim s'$ and a substitution $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t_1}))$ if $\theta_2 = \sigma'$, where $\sigma' = \tilde{\sigma} \cup \{x_1 \mapsto \tilde{\sigma} s'\}$, and $G_2 \equiv\leftarrow \sigma' B'$.

In the above definition, it is easy to verify, using the definition of flattening, Defn. 6.6.4, that $\langle\theta_2, G_2\rangle$ is indeed a subst-goal pair.

We next define, mutually recursively, *set-collecting SLDU tree of rank* k and *derived of rank* k, by simultaneous induction on k. Here, intuitively, k is the maximum depth to which auxiliary set-collecting SLDU trees need to be constructed.

Definition 7.5.4 Let $P \in \mathbf{L}_0$ and let $\langle\theta_1, G_1\rangle$, $\langle\theta_2, G_2\rangle$ be subst-goal pairs with $G_1 \equiv\leftarrow A_1, \ldots, A_m$, $m > 0$, $A_1 \equiv f(\vec{t_1}) \approx x_1$. Then $\langle\theta_2, G_2\rangle$ is **derived of rank** 0 from $\langle\theta_1, G_1\rangle$ using P^F if it is equationally derived from $\langle\theta_1, G_1\rangle$ using some equational clause in P^F.

Definition 7.5.5 Let $P \in \mathbf{L}_0$ and $G \equiv\leftarrow f(\vec{t}) \approx x$, with $\vec{t}$ ground and f collect-all definable. A **set-collecting SLDU tree of rank** 0 **for** $P^F \cup \{G\}$ **returning answer set** s is a tree satisfying the following:

(a) The tree is finite and each non-root node of the tree is a subst-goal pair.

(b) The root node is $\leftarrow f(\vec{t}) \supsetsim x_0$, where x_0 is a new variable.

(c) $\vec{t}$ does not match (using $MatchTuple$) with the lhs of any equational clause

of f in P^F,

(d) For every subset clause $C^F \equiv f(\vec{t'}) \supsetsim s' \leftarrow B'$ in P^F and

every $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t}))$, the root node has a child that is subset derived

from it using C^F and σ.

(e) Each non-root node $\langle \theta, G' \rangle$ is a leaf node when G' is $\Box$; and when G' is not $\Box$ the node

has a child that is derived of rank 0 from it using P^F.

(f) $s = \bigcup \{\sigma x \mid \langle \sigma, \Box \rangle$ a leaf node$\}$.

We note that, in the above definition, if the tree has only a root node then $s = \emptyset$.

Definition 7.5.6 Let $P \in \mathbf{L}_0$ and let $\langle \theta_1, G_1 \rangle$, $\langle \theta_2, G_2 \rangle$ be subst-goal pairs with $G_1 \equiv \leftarrow A_1, \ldots, A_m$, $m > 0$, $A_1 \equiv f(\vec{t_1}) \approx x_1$. Then $\langle \theta_2, G_2 \rangle$ is **derived of rank** $k+1$ from $\langle \theta_1, G_1 \rangle$ using P^F if

(i) f is collect-all definable,

(ii) there is a set-collecting SLDU tree of rank k returning answer set s

for $P^F \cup \{\leftarrow f(\vec{t_1}) \approx x_1\}$, and

(iii) $\sigma = \{x_1 \mapsto NoDups(s)\}$, $\theta_2 = \tilde{\sigma} \circ \theta_1$ and $G_2 \equiv \leftarrow \tilde{\sigma}(A_2, \ldots, A_m)$.

Again, it is easy to verify that $\langle \theta_2, G_2 \rangle$ is indeed a subst-goal pair.

Definition 7.5.7 Let $P \in \mathbf{L}_0$ and $G \equiv \leftarrow f(\vec{t}) \approx x$, with $\vec{t}$ ground and f collect-all definable. A **set-collecting SLDU tree of rank** $k+1$ **for** $P^F \cup \{G\}$ **returning answer set** s is a tree satisfying the following:

(a) The tree is finite and each non-root node of the tree is a subst-goal pair.

(b) The root node is $\leftarrow f(\vec{t}) \supsetsim x_0$, where x_0 is a new variable.

(c) $\vec{t}$ does not match (using $MatchTuple$) with the lhs of any equational clause

of f in P^F,

(d) For every subset clause $C^F \equiv f(\vec{t'}) \supsetsim s' \leftarrow B'$ in P^F (we assume C^F is a variant) and every $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t}))$, the root node has a child that is subset derived from it using C^F and σ.

(e) Each non-root node $\langle\theta, G'\rangle$ is a leaf node when G' is $\Box$; and when G' is not $\Box$ the node has a child that is derived of rank $\leq k+1$ from it using P^F. Also, there is a child of at least one non-root node that is derived of rank $k+1$ from it using P^F.

(f) $s = \bigcup\{\sigma x \mid \langle\sigma, \Box\rangle \text{ a leaf node}\}$.

Definition 7.5.8 Let $P \in \mathbf{L}_0$ and let $\langle\theta_1, G_1\rangle$, $\langle\theta_2, G_2\rangle$ be subst-goal pairs. Then $\langle\theta_2, G_2\rangle$ is **derived** from $\langle\theta_1, G_1\rangle$ using P^F if $\langle\theta_2, G_2\rangle$ is derived of rank k from $\langle\theta_1, G_1\rangle$ using P^F, for some k. If $k \geq 1$, then $\langle\theta_2, G_2\rangle$ is also said to be **collect-all derived** from $\langle\theta_1, G_1\rangle$ using P^F.

Definition 7.5.9 Let $P \in \mathbf{L}_0$ and $G \equiv\leftarrow f(\vec{t}) \approx x$, with $\vec{t}$ ground and f collect-all definable. A **set-collecting SLDU tree for** $P^F \cup \{G\}$ **returning answer set** s is a set-collecting SLDU tree of rank k for $P^F \cup \{G\}$ returning answer set s, for some k.

We note that every child of a non-root node in a set-collecting SLDU tree is either equationally or collect-all derived from its parent. Only the children of the root node are subset derived.

Definition 7.5.10 Let $P \in \mathbf{L}_0$ and $\langle\theta_0, G_0\rangle$ be a subst-goal pair. A **SLDU derivation from** $\langle\theta_0, G_0\rangle$ **using** P^F consists of a (finite or infinite) sequence of subst-goal pairs $\langle\theta_0, G_0\rangle$, $\langle\theta_1, G_1\rangle$, $\langle\theta_2, G_2\rangle$, ..., such that $\langle\theta_{i+1}, G_{i+1}\rangle$ is derived from $\langle\theta_i, G_i\rangle$. A SLDU derivation from $\langle\theta_0, G_0\rangle$ using P^F is **successful** if it is finite and the last goal is $\Box$. A **computed answer** in a successful derivation from $\langle\theta_0, G_0\rangle$ using P^F is the last substitution.

Note that every child of the root node of a set-collecting SLDU tree heads a successful derivation from it using P^F.

Definition 7.5.11 Let $P \in \mathbf{L}_0$ and let the sequence of subst-goal pairs $\langle\theta_0, G_0\rangle$, $\langle\theta_1, G_1\rangle$, ..., $\langle\theta_n, G_n\rangle$, $n \geq 0$, be a successful derivation from $\langle\theta_0, G_0\rangle$ using P^F, with each $\langle\theta_i, G_i\rangle$ being derived of rank r_i from its predecessor, for $1 \leq i \leq n$. Then, the **length** of the derivation is n, and the **rank** of the derivation is $\max\{r_i \mid 1 \leq i \leq n\}$.

In the above definition, if $n = 0$ then the rank of the derivation is 0 (from $\max\{\} = 0$ vacuously).

Definition 7.5.12 Let $P \in \mathbf{L}_0$ and $G \in \mathbf{G}_0$. A **SLDU derivation of** $P \cup \{G\}$ is a SLDU derivation from $\langle \epsilon, G^F \rangle$ using P^F. A **successful SLDU derivation of** $P \cup \{G\}$ is a successful SLDU derivation from $\langle \epsilon, G^F \rangle$ using P^F. A **computed answer** θ for $P \cup \{G\}$ is a computed answer in a successful SLDU derivation from $\langle \epsilon, G^F \rangle$ using P^F restricted to $Var(G^F)$ and having $NoDups$ applied to all ground terms in its range.

By a **computation** of $P \cup \{G\}$ we mean an attempt to construct a SLDU derivation of $P \cup \{G\}$.

We illustrate the use of the above definitions on the following example computations. Consider P_3 in §7.2 and the query $Q \equiv size(\{1,2\}) \approx x$. Here $Q^F \equiv Q$. A SLDU derivation for $P_3 \cup \{\leftarrow Q\}$ is as below, and it involves only equationally derived subst-goal pairs.

$$
\begin{array}{l}
\langle \epsilon, \leftarrow size(\{1,2\}) \approx x \rangle \\
\quad \Big| \ \theta_1 = \{x_0 \mapsto 2, y_0 \mapsto \{1\}, x \mapsto succ(z_0)\} \\
\langle \theta_1, \leftarrow size(\{1\}) \approx z_0 \rangle \\
\quad \Big| \ \theta_2 = \{x_1 \mapsto 1, y_1 \mapsto \emptyset, z_0 \mapsto succ(z_1)\} \\
\langle \theta_2 \circ \theta_1, \leftarrow size(\emptyset) \approx z_1 \rangle \\
\quad \Big| \ \theta_3 = \{z_1 \mapsto 0\} \\
\langle \theta_3 \circ \theta_2 \circ \theta_1, \square \rangle
\end{array}
$$

From this we get the computed answer to be $\theta_3 \circ \theta_2 \circ \theta_1 \lceil \{x\} = \{x \mapsto succ(succ(0))\}$. Notice that only one (any) match has been used at the first equationally derived step where multiple matches of x_0, y_0 exist in matching $\{x_0 \backslash y_0\}$ with $\{1, 2\}$.

Now consider P_2 in §7.2 and the query $Q \equiv perms(\{1,2\}) \approx x$. Here $Q^F \equiv Q$. A SLDU derivation for $P_2 \cup \{\leftarrow Q\}$ is as below. From it, we get the computed answer to be $\theta_1 \lceil \{x\} = \theta_1 = \{x \mapsto \{[1,2],[2,1]\}\}$.

$$\begin{array}{l}
\langle \epsilon, \leftarrow perms(\{1,2\}) \approx x \rangle \\
\quad \Big| \; \theta_1 = \{x \mapsto NoDups(s_3)\} = \{x \mapsto \{[1,2],[2,1]\}\} \quad \text{(from tree } T_1 \text{ below)} \\
\langle \theta_1, \Box \rangle
\end{array}$$

Below is the set-collecting tree T_1 for $P_2^F \cup \{\leftarrow perms(\{1,2\}) \approx x\}$ returning answer set $s_3 = \{[1,2],[2,1]\}$. Its right branch is symmetrical to the left branch and so we show only the details of the latter. The tree is of rank 2 and the variant of the subset clause used at the root node is $perms(\{x_0 \backslash y_0\}) \supsetsim y_2 \leftarrow perms(y_0) \approx y_1, distr(x_0, y_1) \approx y_2$. Hence, we have that the above successful derivation is of rank 3 since, in it, $\langle \theta_1, \Box \rangle$ is derived of rank 3 from its predecessor.

$$\leftarrow perms(\{1,2\}) \supsetsim v_0$$

$$\begin{array}{ll}
\sigma_1 = \{x_0 \mapsto 1, y_0 \mapsto \{2\}, v_0 \mapsto y_2\} \; / & \backslash \; \sigma_9 = \{x_0 \mapsto 2, y_0 \mapsto \{1\}, v_0 \mapsto y_2\} \\
\langle \sigma_1, \leftarrow perms(\{2\}) \approx y_1, distr(1, y_1) \approx y_2 \rangle & \langle \sigma_9, \leftarrow perms(\{1\}) \approx y_1, distr(2, y_1) \approx y_2 \rangle \\
\sigma_6 = \{y_1 \mapsto NoDups(s_1)\} \; \Big| = \{y_1 \mapsto \{[2]\}\} \text{ (from tree } T_2 \text{ below)} & \Big| \; \sigma_{10} = \{y_1 \mapsto \{[1]\}\} \\
\langle \sigma_6 \circ \sigma_1, \leftarrow distr(1, \{[2]\}) \approx y_2 \rangle & \langle \sigma_{10} \circ \sigma_9, \leftarrow distr(2, \{[1]\}) \approx y_2 \rangle \\
\sigma_8 = \{y_2 \mapsto NoDups(s_2)\} \; \Big| = \{y_2 \mapsto \{[1,2]\}\} \text{ (from tree } T_4 \text{ below)} & \Big| \; \sigma_{11} = \{y_2 \mapsto \{[2,1]\}\} \\
\langle \sigma_8 \circ \sigma_6 \circ \sigma_1, \Box \rangle & \langle \sigma_{11} \circ \sigma_{10} \circ \sigma_9, \Box \rangle
\end{array}$$

$$\text{answer set } s_3 = \bigcup \{\sigma_8 \circ \sigma_6 \circ \sigma_1(v_0), \sigma_{11} \circ \sigma_{10} \circ \sigma_9(v_0)\} = \{[1,2]\} \cup \{[2,1]\} = \{[1,2],[2,1]\}$$

Below is the set-collecting tree T_2 for $P_2^F \cup \{\leftarrow perms(\{2\}) \approx y_1\}$ returning answer set $s_1 = \{[2]\}$. The tree is of rank 1 and the variant of the subset clause used at the root node is $perms(\{x_1 \backslash y_3\}) \supsetsim y_5 \leftarrow perms(y_3) \approx y_4, distr(x_1, y_4) \approx y_5$.

$$\leftarrow perms(\{2\}) \supsetapprox v_1$$

$$\sigma_2 = \{x_1 \mapsto 2, y_3 \mapsto \emptyset, v_1 \mapsto y_5\}$$

$$\langle \sigma_2, \leftarrow perms(\emptyset) \approx y_4, distr(2, y_4) \approx y_5 \rangle$$

$$\sigma_3 = \{y_4 \mapsto \{[\,]\}\}$$

$$\langle \sigma_3 \circ \sigma_2, \leftarrow distr(2, \{[\,]\}) \approx y_5 \rangle$$

$$\sigma_5 = \{y_5 \mapsto NoDups(s_0)\} = \{y_5 \mapsto \{[2]\}\} \quad \text{from tree } T_3 \text{ below.}$$

$$\langle \sigma_5 \circ \sigma_3 \circ \sigma_2, \Box \rangle$$

answer set $s_1 = \bigcup\{\sigma_5 \circ \sigma_3 \circ \sigma_2(v_1)\} = \sigma_5 \circ \sigma_3 \circ \sigma_2(v_1) = \{[2]\}$

Below is the set-collecting tree T_3 for $P_2^F \cup \{\leftarrow distr(2, \{[\,]\}) \approx y_5\}$ returning answer set $s_0 = \{[2]\}$. The tree is of rank 0 and the variant of the subset clause used at the root node is $distr(x_2, \{y_6 \backslash z_0\}) \supsetapprox \{[x_2|y_6]\}$.

$$\leftarrow distr(2, \{[\,]\}) \supsetapprox v_2$$

$$\sigma_4 = \{x_2 \mapsto 2, y_6 \mapsto [\,], z_0 \mapsto \emptyset, v_2 \mapsto \{[2|[\,]]\}\}$$

$$\langle \sigma_4, \Box \rangle$$

answer set $s_0 = \bigcup\{\sigma_4(v_2)\} = \sigma_4(v_2) = \{[2]\}$

Below is the set-collecting tree T_4 for $P_2^F \cup \{\leftarrow distr(1, \{[2]\}) \approx y_2\}$ returning answer set $s_2 = \{[1, 2]\}$. The tree is of rank 0 and the variant of the subset clause used at the root node is $distr(x_3, \{y_7 \backslash z_1\}) \supsetapprox \{[x_3|y_7]\}$.

$$\leftarrow distr(1, \{[2]\}) \supsetapprox v_3$$

$$\sigma_7 = \{x_3 \mapsto 1, y_7 \mapsto [2], z_1 \mapsto \emptyset, v_3 \mapsto \{[1|[2]]\}\}$$

$$\langle \sigma_7, \Box \rangle$$

answer set $s_2 = \bigcup\{\sigma_7(v_3)\} = \sigma_7(v_3) = \{[1, 2]\}$

Lemma 7.5.13 Let $P \in \mathbf{L}_0$, and let $\langle \theta_0, G_0 \rangle$, ..., $\langle \theta_n, G_n \rangle$ be a finite SLDU derivation from $\langle \theta_0, G_0 \rangle$ using P^F. Then $\theta_n \circ \theta_0 = \theta_n$.

Proof: Trivial. By induction on n, the number of steps in the derivation. Base: $n = 0$. Clearly $\theta_0 \circ \theta_0 = \theta_0$. Induction step: $n > 0$. We have $\theta_{n-1} \circ \theta_0 = \theta_{n-1}$ by the induction hypothesis. Consider the last derivation step of $\langle \theta_n, G_n \rangle$ from $\langle \theta_{n-1}, G_{n-1} \rangle$. Then $\langle \theta_n, G_n \rangle$ is derived of rank k for some $k \geq 0$. According to either of Defns. 7.5.2 or 7.5.6 there is a σ such that $\theta_n = \sigma \circ \theta_{n-1}$. Hence $\theta_n \circ \theta_0 = \sigma \circ \theta_{n-1} \circ \theta_0 = \sigma \circ \theta_{n-1} = \theta_n$. ∎

7.6 Fixpoint Semantics

We now develop a fixpoint semantics for subset-equational programs P. It is given in terms of the fixpoint of what is usually called an *immediate consequence operator* T_P (also called a *modus ponens* operator). Such a semantics provides a link between the operational and declarative semantics, which helps to prove the completeness direction of their equivalence.

The fixpoint semantics is also significant in its own right in that it provides an inductive construction of $\mathcal{D}_P$, the set of ground logical consequences of *comp*(P). Thus it provides a basis for bottom-up computation strategies for program evaluation. That consequences of a modus ponens operator are logical consequences are but natural. That *all* logical consequences can be obtained through the operator is significant. Thus the operator helps to deduce also what are not logical consequences.

We begin with the definition of T_P. For a subset-equational program P, let $T_P: D_{\cong} \rightarrow \mathcal{P}(B_{\cong})$ be as follows. The domain $D_{\cong} = \{I \in \mathcal{P}(B_{\cong}) \mid I \text{ models } FunAx\} \subseteq \mathcal{P}(B_{\cong})$. Thus $\emptyset \in D_{\cong}$ but $B_{\cong} \notin D_{\cong}$. Also, if $I \in D_{\cong}$ then any subset of I is also in $D_{\cong}$. (Usually, T_P is defined as an operator from $\mathcal{P}(B_{\cong}) \rightarrow \mathcal{P}(B_{\cong})$, but such a map presents difficulties, as discussed later on.)

Now let I be an interpretation in $D_{\cong}$. We define $T_P: D_{\cong} \rightarrow \mathcal{P}(B_{\cong})$ as:

$T_P(I) = \{[A] \mid$ there is a ground instance $A' \leftarrow B$ of a clause in $P^{FD} \cup CAA(P)$ such that B^I holds and $[A] = A'^I\}$

By a ground instance of a collect-all clause $CAA(f) \equiv f(\vec{v}) \approx s \leftarrow cond\_finset, cond\_args,$ $cond\_welldef$ is meant the clause with a ground instance for its universally quantified variables $\vec{v}$. We see that $T_P(I)$ is made up of atoms that are derived from the heads of two kinds of clauses — those in P^{FD} and those in $CAA(P)$. That the head of a ground instance of a clause in $CAA(P)$ is well formed, i.e., its set abstraction exists as a set, follows from Lemma 7.2.2 since I models $SetAx$ and $FunAx$. Indeed, this motivates the choice of $D_{\cong}$ as the domain of T_P. Thus, we have that $T_P(I) \in \mathcal{P}(B_{\cong})$.

Note that there are $I \in D_{\cong}$ such that $T_P(I) \notin D_{\cong}$, so that we cannot define $T_P\colon D_{\cong} \rightarrow D_{\cong}$, as would be desirable. For example, consider $comp(P_3)$ of §7.2, which we repeat here.

$$comp(P_3): \quad size(\emptyset) \approx 0$$

$$size(\{x/y\}) \approx succ(z) \leftarrow size(y) \approx z, x \notin y$$

Let $I = \{[size(\{1\}) \approx succ(0)], [size(\{2\}) \approx succ^2(0)]\} \in D_{\cong}$. Then $T_P(I) \supseteq \{[size(\{1,2\}) \approx succ^2(0)], [size(\{1,2\}) \approx succ^3(0)]\}$ and hence does not model $FunAx$.

Two additional operators on $D_{\cong} \rightarrow \mathcal{P}(B_{\cong})$ that help to describe T_P are as follows.

$$\begin{aligned} R_P(I) = \{[A] \mid {} & \text{there is a ground instance } A' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t'}) \\ & \text{of a clause in } P^{FD} \text{ such that } [A] = [A'],\ A' \equiv f(\vec{t'}) \approx s' \text{ or } f(\vec{t'}) \gtrsim s', \\ & nonmem(\vec{t'}) \text{ holds, and } \{[A_1], \ldots, [A_l]\} \subseteq I\} \end{aligned}$$

$$\begin{aligned} S_P(I) = \{[A] \mid {} & [A] = [f(\vec{t}) \approx s],\ f \text{ is collect-all definable,} && (1) \\ & s = \bigcup\{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \gtrsim s'] \in R_P(I)\}, && (2) \\ & \text{for any ground instance } \vec{t'} \text{ (with } nonmem(\vec{t'}) \text{ holding) of the} \\ & \text{lhs of any equational clause of } f \text{ in } P^{FD},\ [\vec{t}] \neq [\vec{t'}], && (3) \\ & \text{for every ground instance } \vec{t'} \text{ (with } nonmem(\vec{t'}) \text{ holding) of the} \end{aligned}$$

lhs of every subset clause of f in P^{FD} such that $[\vec{t}] = [\vec{t'}]$, there is a ground instance of the remaining variables of the same clause such that if $f(\vec{t'}) \supsetsim s' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t'})$ is the ground instance of the clause then $\{[A_1], \ldots, [A_l]\} \subseteq I\}$ (4)

The definition of $R_P(I)$ is motivated by those atoms in $T_P(I)$ arising out of the heads of clauses in P^{FD}, while the definition of $S_P(I)$ is motivated by those atoms in $T_P(I)$ arising out of the heads of clauses in $CAA(P)$. In $S_P(I)$, the conditions (3) and (4) are just descriptive statements of *cond_args* and *cond_welldef* in $CAA(f)$. The advantage of $R_P(I)$ and $S_P(I)$ over $T_P(I)$ is that the former do not refer to $CAA(P)$ and are somewhat easier to use. While it is clear that $R_P(I)$ is well-defined, we need to make sure that the set abstraction in definition of $S_P(I)$, viz., condition (2), exists as a finite set. The following lemma ensures this.

Lemma 7.6.1 Let I be an interpretation, f be collect-all definable, and let conditions (3) and (4) in the definition of $S_P(I)$ hold for a ground tuple $\vec{t}$. Also, let $Co = \{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \supsetsim s'] \in R_P(I)\}$. Then, there is a finite subset Su of Co such that $Su = Co$.

Proof: Here, form Su out of the finitely many matches of $\vec{t}$ with the head of every subset assertion about f (using Prop. 4.5.2(ii) or Cor. 7.4.4) in the following way. For each of these matches with a subset clause of f, by condition (4) of $S_P(I)$, we know there is an instance of the variables of flattening in the subset clause for which the body of the clause is true under I. Let $f(\vec{t'}) \supsetsim s' \leftarrow B$ be the ground instance of the subset clause arising out of this match and this instance of the variables of flattening. Then let $s' \in Su$. Thus Su is formed out of a finite number of terms like s'.

Clearly $Su \subseteq Co$. That $Su = Co$ follows by Prop. 4.5.2(ii) or Cor. 7.4.4, and by Lemma 6.6.5. That is, any other ground instance $f(\vec{t''}) \supsetsim s'' \leftarrow B'$ of a subset clause such that $[\vec{t}] = [\vec{t''}]$ and B' holds in I, must have $\vec{t''}$ equal (under $SetAx$) to one of the matches of $\vec{t}$ (by Prop. 4.5.2(ii)), and must have s'' equal (under $SetAx$) to some element of Su by uniqueness of solutions to variables of flattening (Lemma 6.6.5). ■

Proposition 7.6.2 $T_P(I) = R_P(I) \cup S_P(I)$.

Proof: That $[A] \in T_P(I)$ with $[A]$ arising from the head of a clause in P^{FD} iff $[A] \in R_P(I)$ is obvious. So we only show that $[A] \in T_P(I)$ with $[A]$ arising from the head of a clause in $CAA(P)$ iff $[A] \in S_P(I)$.

($\Rightarrow$): Let $[A] \in T_P(I)$ arise from the ground instance $A' \leftarrow B$ of a clause in $CAA(P)$ with B^I holding and $[A] = A'^I$. Let $A' \leftarrow B$ be:

$$f(\vec{t}) \approx \cup_{i=1}^{n} \bigcup\{w \mid \cdots\} \leftarrow \mathit{cond\_finset} \wedge \mathit{cond\_args} \wedge \mathit{cond\_welldef}$$

with

$$\mathit{cond\_finset} \equiv \bigwedge_{i=1}^{n} \exists u(set(u) \wedge \forall w(w \in u \leftrightarrow \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)))$$

$$\mathit{cond\_args} \equiv \neg\exists \vec{x'_1}(D'_1 \wedge \vec{t} = \tilde{\vec{t'_1}}) \wedge \cdots \wedge \neg\exists \vec{x'_m}(D'_m \wedge \vec{t} = \tilde{\vec{t'_m}})$$

$$\mathit{cond\_welldef} \equiv \bigwedge_{i=1}^{n} \forall \vec{x_i}(D_i \wedge \vec{t} = \tilde{\vec{t_i}} \rightarrow \exists \vec{y_i} F_i)$$

It is easy to see that condition (3) of $S_P(I)$ holds since $\mathit{cond\_args}^I$ holds and that condition (4) of $S_P(I)$ holds since $\mathit{cond\_welldef}^I$ holds. Also it is not hard to see that $[\cup_{i=1}^{n} \bigcup\{w \mid \cdots\}^I] = [\bigcup\{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \gtrsim s'] \in R_P(I)\}]$ by using Cor. 7.4.4, Lemma 6.6.5 and the meaning of $R_P(I)$. So $[A] = A'^I \equiv [f(\vec{t}) \approx \cup_{i=1}^{n} \bigcup\{w \mid \cdots\}] = [f(\vec{t}) \approx s]$. Hence $[A] \in S_P(I)$.

($\Leftarrow$): Let $[A] \in S_P(I)$ with $[A] = [f(\vec{t}) \approx s]$, f collect-all definable, and conditions (2), (3) and (4) in the definition of $S_P(I)$ holding. Consider the ground instance of $CAA(f)$:

$$f(\vec{t}) \approx \cup_{i=1}^{n} \bigcup\{w \mid \cdots\} \leftarrow \mathit{cond\_finset} \wedge \mathit{cond\_args} \wedge \mathit{cond\_welldef}$$

with *cond_finset*, *cond_args* and *cond_welldef* as above. By Lemma 7.2.2, $\mathit{cond\_finset}^I$ holds. It is easy to see that $\mathit{cond\_args}^I$ holds using condition (3), and that $\mathit{cond\_welldef}^I$ holds using condition (4). So $[f(\vec{t}) \approx \cup_{i=1}^{n} \bigcup\{w \mid \cdots\}] \in T_P(I)$. Also, we have that $[\cup_{i=1}^{n} \bigcup\{w \mid \cdots\}^I] = [\bigcup\{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \gtrsim s'] \in R_P(I)\}]$ by the same reasoning as above. Hence $[A] = [f(\vec{t}) \approx s] = [f(\vec{t}) \approx \cup_{i=1}^{n} \bigcup\{w \mid \cdots\}] \in T_P(I)$. ■

We assume acquaintanceship with the theory of structured sets such as in [Sch86] Chapter 6. We therefore take as understood terms such as *partially ordered set* (*poset*), *complete partial ordering* (*cpo*), *pointed cpo*, *complete lattice*, *least upper bound* (*lub* or *sup*) of a set X (denoted by $\bigsqcup X$), *greatest lower bound* (*glb* or *inf*) of a set X (denoted by $\bigsqcap X$), *chain*, *monotonic* function, *continuous* function, *fixed point* (*fp*) and *least fixed point* (*lfp*) of a function.

We have, as usual, that $\mathcal{P}(B_{\cong})$ is a complete lattice ordered by set inclusion $\subseteq$, with $\bigsqcup X = \bigcup X$ and $\bigsqcap X = \bigcap X$ for any $X \subseteq \mathcal{P}(B_{\cong})$. However, $D_{\cong}$ is not a complete lattice when ordered by $\subseteq$, for it has no greatest or top element and hence $\bigsqcap \emptyset$ does not exist. To see this, let $I_1 = \{[f(1) \approx 1]\}$ and $I_2 = \{[f(1) \approx 2]\}$. Clearly $I_1, I_2 \in D_{\cong}$ but any top element of $D_{\cong}$ must contain $\{[f(1) \approx 1], [f(1) \approx 2]\}$ so that $[1] = [2]$ by $FunAx$ — contradiction.

Since $D_{\cong}$ is not a complete lattice and T_P is not an operator in $D_{\cong} \rightarrow D_{\cong}$, the usual application of the theory of structured sets does not go through smoothly. Hence, we first develop below some additional definitions and modifications of some well-known properties of structured sets.

Definition 7.6.3 Let E be a poset ordered by $\sqsubseteq$. A set D is **downward closed** in E if $D \subseteq E$ and for all $x \in D$, for all $y \in E$ such that $y \sqsubseteq x$, we have $y \in D$.

This definition gives that any poset E is downward closed in itself.

Definition 7.6.4 A poset D ordered by $\sqsubseteq$ is a **near complete lattice** if for all nonempty subsets X of D, $\bigsqcap X$ exists.

It follows that a nonempty near complete lattice D posseses a least or bottom element, viz., $\bigsqcap D$. A near complete lattice D for which $\bigsqcap \emptyset$ exists, i.e., a top element exists, is a complete lattice; and hence the name near complete lattice.

Lemma 7.6.5 Let E be a near complete lattice ordered by $\sqsubseteq$. If D is downward closed in E then

D is a near complete lattice ordered by $\sqsubseteq$.

Proof: Since E is a poset ordered by $\sqsubseteq$, it is easy to see that $D \subseteq E$ is also a poset ordered by $\sqsubseteq$. Let $X \subseteq D$ and $X \neq \emptyset$. So let $y \in X$. We claim that $\bigsqcap X$ in D is just $\bigsqcap X$ in E. To avoid confusion, we denote the former simply as $\bigsqcap X$ and the latter by $\bigsqcap_E X$. Since E is a near complete lattice $\bigsqcap_E X$ exists. We have $\bigsqcap_E X$ is a lower bound for y and hence $\bigsqcap_E X \in D$. Therefore $\bigsqcap X = \bigsqcap_E X$. ∎

Definition 7.6.6 Let D, D' be subsets of a poset E ordered by $\sqsubseteq$, and let $T\colon D \to D'$. A point $x \in D$ is a **pre-fixpoint** of T if $T(x) \sqsubseteq x$.

Let D, D' be subsets of a poset E such that D, D' are pointed cpos, and let $T\colon D \to D'$ be a monotonic function. We define $T \uparrow \alpha$ for an ordinal α as follows.

$T \uparrow 0 = \bigsqcap D$ (= least element of D)

If $T \uparrow \beta \in D$ for all $\beta < \alpha$, then:

$$T \uparrow \alpha = \begin{cases} T(T \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \bigsqcup\{T \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

It is easy to see that if $T \uparrow \beta \in D$ for all $\beta < \alpha$ then $T \uparrow \alpha$ is well-defined, for when α is a limit ordinal, the set $\{T \uparrow \beta \mid \beta < \alpha\}$ is a chain in D whose lub therefore exists in D. The only problematic case is for α a successor ordinal, when $T \uparrow \alpha$ may not be in D.

The next two lemmas are modifications of a theorem of Knaster and Tarski (see [Llo87] Prop. 5.1) and a theorem of Kleene (see [Sch86] Thm. 6.11) respectively.

Lemma 7.6.7 Let D, D' be subsets of a poset E ordered by $\sqsubseteq$ such that D is a near complete lattice. Let $T\colon D \to D'$ be a monotonic function having a pre-fixpoint. Then *lfp*(T) exists and

$$\mathit{lfp}(T) = \bigsqcap\{x \mid T(x) = x\} = \bigsqcap\{x \mid T(x) \sqsubseteq x\}.$$

Proof: The proof is as usual (see [Llo87] Prop. 5.1) except that we need to ensure that $\bigsqcap\{x \mid$

$T(x) \sqsubseteq x\}$ and $\bigsqcap\{x \mid T(x) = x\}$ exist. The former exists (since the set is nonempty) and, as usual, can be shown to be a fixed point of T. Hence, the latter too exists. ∎

Lemma 7.6.8 Let D, D' be subsets of a poset E such that D, D' are pointed cpos and D is downward closed in E. Let $T: D \to D'$ be a continuous function. If $\mathit{lfp}(T)$ exists then $\mathit{lfp}(T) = T \uparrow \omega$.

Proof: We first show that $T \uparrow \omega$ is well-defined, i.e., $T \uparrow n \in D$ for all $n < \omega$. We have T is monotonic since it is continuous. Let $e = \mathit{lfp}(T)$. Then, it is easy to see by induction on n that, for all $n < \omega$, $T \uparrow n \sqsubseteq T(e) = e \in D$, using monotonicity of T. Hence $T \uparrow n \in D$ by downward closedness of D.

Next, as usual, we have that $T \uparrow \omega$ is a fixed point of T (see [Sch86] Thm. 6.11). Thus $e \sqsubseteq T \uparrow \omega$. Also, $T \uparrow \omega = \bigsqcup\{T \uparrow n \mid n < \omega\} \sqsubseteq e$, from $T \uparrow n \sqsubseteq e$. Therefore $e = T \uparrow \omega$. ∎

We now apply the above theory to $T_P: D_{\cong} \to \mathcal{P}(B_{\cong})$, where $D_{\cong} \subseteq \mathcal{P}(B_{\cong})$. We have that $D_{\cong}$ is downward closed in $\mathcal{P}(B_{\cong})$. Other properties of $D_{\cong}$ are as expressed in the following lemma.

Lemma 7.6.9

(i) $D_{\cong}$ is a near complete lattice and a pointed cpo ordered by $\subseteq$.

(ii) If X is a nonempty subset of $D_{\cong}$ then $\bigsqcap X = \bigcap X$.

(iii) If X is a chain in $D_{\cong}$ then $\bigsqcup X = \bigcup X$.

Proof: (i): From Lemma 7.6.5 we get that $D_{\cong}$ is a near complete lattice. Hence, $D_{\cong}$ has a least element, viz., $\emptyset$. Next we show that $D_{\cong}$ is a cpo. Let X be a chain in $D_{\cong}$. Hence, $X \neq \emptyset$ and for all $I_1, I_2 \in X$, either $I_1 \subseteq I_2$ or $I_2 \subseteq I_1$. We claim that $\bigsqcup X = \bigcup X$.

Firstly, we show that $\bigcup X \in D_{\cong}$, i.e., $\bigcup X$ models $FunAx$. Let $[f(\vec{t}) \approx t_1], [f(\vec{t}) \approx t_2] \in \bigcup X$. Then $[f(\vec{t}) \approx t_1] \in I_1$, $[f(\vec{t}) \approx t_2] \in I_2$ for some $I_1, I_2 \in X$. Without loss of generality let $I_1 \subseteq I_2$. Hence $[f(\vec{t}) \approx t_1], [f(\vec{t}) \approx t_2] \in I_2$. Therefore $[t_1] = [t_2]$.

Next, $\bigcup X$ is clearly an upper bound for X. Let I be any upper bound of X. Then $\bigcup X \subseteq I$, for if $[f(\vec{t}) \approx t_1] \in I_1$ for some $I_1 \in X$ then $[f(\vec{t}) \approx t_1] \in I$, as $I_1 \subseteq I$. Hence $\bigsqcup X = \bigcup X$.

(ii): Let X be a nonempty subset of $D_{\cong}$. In the proof of Lemma 7.6.5 we showed that $\bigsqcap X$ in $D_{\cong}$ is just $\bigsqcap X$ in $\mathcal{P}(B_{\cong})$, viz., $\bigcap X$.

(iii): This is in the proof of part (i) above. ∎

Proposition 7.6.10 I models $comp(P)$ $\Leftrightarrow$ $T_P(I) \subseteq I$.

Proof: ($\Rightarrow$): Let I model $comp(P)$. Hence I models $FunAx$ and $T_P(I)$ is defined. Let $[A] \in T_P(I)$. So there is a ground instance $A' \leftarrow B$ of a clause in $P^{FD} \cup CAA(P)$ such that B^I holds and $A'^I = [A]$. Hence A'^I holds, i.e., $[A] \in I$, since I models $A' \leftarrow B$. Thus $T_P(I) \subseteq I$.

($\Leftarrow$): Let $T_P(I) \subseteq I$. So I models $FunAx$ as I is an argument of T_P. Now, to show that I models $P^{FD} \cup CAA(P)$. Let $A' \leftarrow B$ be a ground instance of a clause in $P^{FD} \cup CAA(P)$. Suppose B^I holds. Let $[A] = A'^I$. Then $[A] \in T_P(I)$. Hence $[A] \in I$, i.e., A'^I holds. ∎

Proposition 7.6.11 R_P, S_P, and T_P are continuous.

Proof: We first need to show that R_P, S_P, and T_P are monotonic. Let $I_1 \subseteq I_2$ be interpretations. For R_P, we clearly have $R_P(I_1) \subseteq R_P(I_2)$. For S_P, we have $S_P(I_1) \subseteq S_P(I_2)$ by the following. Let $[f(\vec{t}) \approx s_1] \in S_P(I_1)$. Then $[f(\vec{t}) \approx s_2] \in S_P(I_2)$ and $s_2 \supseteq s_1$. That $s_2 = s_1$ follows by the same reasoning showing that $Co = Su$ in Lemma 7.6.1. For T_P, we have $T_P(I_1) = R_P(I_1) \cup S_P(I_1) \subseteq R_P(I_2) \cup S_P(I_2) = T_P(I_2)$, by using monotonicity of R_P and S_P.

Now, let X be a chain in $D_{\cong}$. We show that $R_P(\bigcup X) = \bigcup\{R_P(I) \mid I \in X\}$, and similar statements for S_P and T_P. We first note (to be used later) that $\{[A_1], \ldots, [A_l]\} \subseteq \bigcup X$, $l \geq 0$ iff $\{[A_1], \ldots, [A_l]\} \subseteq I$ for some $I \in X$. This follows from the fact that any nonempty finite subset of X is also a chain having a greatest element.

For R_P, we have $[A] \in R_P(\bigcup X)$

iff there is a ground instance $A' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t'})$

of a clause in P^{FD} such that $[A] = [A']$, $A' \equiv f(\vec{t'}) \approx s'$ or $f(\vec{t'}) \gtrsim s'$,

$nonmem(\vec{t'})$ holds, and $\{[A_1], \ldots, [A_l]\} \subseteq \bigcup X\}$

iff there is a ground instance $A' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t'})$

of a clause in P^{FD} such that $[A] = [A']$, $A' \equiv f(\vec{t'}) \approx s'$ or $f(\vec{t'}) \gtrsim s'$,

$nonmem(\vec{t'})$ holds, and $\{[A_1], \ldots, [A_l]\} \subseteq I$ for some $I \in X$

iff $[A] \in R_P(I)$ for some $I \in X$

iff $[A] \in \bigcup\{R_P(I) \mid I \in X\}$.

For S_P, we have $[A] \in S_P(\bigcup X)$

iff $[A] = [f(\vec{t}) \approx s]$, f is collect-all definable,

$s = \bigcup\{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \gtrsim s'] \in R_P(\bigcup X)\}$,

condition (3) holds, and condition (4) holds with $\{[A_1], \ldots, [A_l]\} \subseteq \bigcup X$

iff $[A] = [f(\vec{t}) \approx s]$, f is collect-all definable,

$s = \bigcup\{s' \mid [\vec{t}] = [\vec{t'}], [f(\vec{t'}) \gtrsim s'] \in R_P(I)\}$,

condition (3) holds, and condition (4) holds with $\{[A_1], \ldots, [A_l]\} \subseteq I$, for some $I \in X$

(We get the above equivalence using the following. The set abstraction involved is a finite set, and for each of its elements s', for which $[f(\vec{t'}) \gtrsim s'] \in R_P(\bigcup X)$, we have that there is a ground instance $f(\vec{t''}) \gtrsim s'' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t''})$ such that $[f(\vec{t'}) \gtrsim s'] = [f(\vec{t''}) \gtrsim s'']$, $nonmem(\vec{t'})$ holds, and $\{[A_1], \ldots, [A_l]\} \subseteq \bigcup X$. Hence, upon considering all the finite number of elements of the set abstraction, we see that only a finite subset of $\bigcup X$ is involved. By the note above, this finite subset is also a subset of I for some $I \in X$. For the rest, we reason as in the proof of $Co = Su$ in Lemma 7.6.1.)

iff $[A] \in S_P(I)$ for some $I \in X$

iff $[A] \in \bigcup\{S_P(I) \mid I \in X\}$.

For T_P, we have $T_P(\bigcup X) = R_P(\bigcup X) \cup S_P(\bigcup X) = \bigcup\{R_P(I) \mid I \in X\} \cup \bigcup\{S_P(I) \mid I \in X\} = \bigcup\{R_P(I) \cup S_P(I) \mid I \in X\} = \bigcup\{T_P(I) \mid I \in X\}$. ∎

Theorem 7.6.12 Let $P \in \mathbf{L}_0$ be consistent and M_P be the least model of $comp(P)$. Then,

$$lfp(T_P) = T_P \uparrow \omega = M_P.$$

Proof: We have that M_P is a pre-fixpoint of T_P since $T_P(M_P) \subseteq M_P$ by Prop. 7.3.4 and Prop. 7.6.10. Hence, by Lemma 7.6.7, $lfp(T_P)$ exists and $lfp(T_P) = \bigcap\{I \mid T_P(I) \subseteq I\} = \bigcap\{I \mid I$ models $comp(P)\} = M_P$. Also, by Lemma 7.6.8 and Prop. 7.6.11, $lfp(T_P) = T_P \uparrow \omega$. ∎

Finally, we discuss why $T_P\colon \mathcal{P}(B_{\cong}) \to \mathcal{P}(B_{\cong})$ is inadequate. Firstly, $T_P(I)$ may not even be defined for I not modeling $FunAx$. For example, let P be $f(x) \gtrsim g(x)$, for which $comp(P)$ is:

$$\begin{aligned} comp(P): \quad & f(x) \gtrsim y \leftarrow g(x) \approx y \\ & f(x) \approx \bigcup\{y \mid g(x) \approx y\} \leftarrow \exists y(g(x) \approx y) \end{aligned}$$

Then, $T_P(B_{\cong})$ is undefined since the collect-all set abstraction is not a finite set for any ground instance of x. Furthermore, T_P is not monotonic even for those I at which it is defined. For let $I_1 = \{[g(1) \approx \{1\}]\}$ and $I_2 = \{[g(1) \approx \{1\}], [g(1) \approx \{2\}]\}$. Then $T_P(I_1) = \{[f(1) \gtrsim \{1\}], [f(1) \approx \{1\}]\}$ and $T_P(I_2) = \{[f(1) \gtrsim \{1\}], [f(1) \gtrsim \{2\}], [f(1) \approx \{1, 2\}]\}$, but $T_P(I_1) \not\subseteq T_P(I_2)$.

7.7 Soundness and Completeness

We now show the equivalence of the declarative and operational semantics for subset-equational programs. We first give the soundness direction of this equivalence.

From hereon, for the rest of the chapter, a derivation will refer to a SLDU derivation and a set-collecting tree will refer to a set-collecting SLDU tree.

The soundness property of subset-equational programs follows as a corollary of the following theorem.

Theorem 7.7.1 Let $P \in \mathbf{L}_0$, $\langle \theta, G \rangle$ be a subst-goal pair, and $\vec{t}$ be ground. Let query Q correspond to G. We have the following:

(i) If ρ is the computed answer in a successful derivation from $\langle \theta, G \rangle$ using P^F, then ρQ is ground and $comp(P) \models \rho Q$.

(ii) If s is the answer set returned by a set-collecting tree for $P \cup \{\leftarrow f(\vec{t}) \approx x\}$, then s is ground and $comp(P) \models f(\vec{t}) \approx s$.

Proof: The two parts, (i) and (ii), are proved by simultaneous induction on k, where in (i), k is the rank of the successful derivation, and in (ii), is the rank of the set-collecting tree.

Basis step: $k = 0$. We show the statements (i) and (ii) for rank 0. For (i), we need to do an auxiliary induction on the length of the successful derivation, and in (ii) we need the result from (i).

(i): Auxiliary basis step: $n = 0$. Here $\langle \theta, G \rangle \equiv \langle \theta, \Box \rangle$. Since $Q \equiv true$, ρQ is ground and $comp(P) \models \rho Q$.

Auxiliary induction step: $n > 0$. Let $\langle \theta_0, G_0 \rangle$, $\langle \theta_1, G_1 \rangle$, $\ldots$, $\langle \theta_n, \Box \rangle$ be the sequence of subst-goal pairs in the successful derivation. So $\langle \theta_0, G_0 \rangle \equiv \langle \theta, G \rangle$ and $\theta_n = \rho$. Clearly, every derived subst-goal pair is of rank 0, i.e., it is equationally derived.

Let $G_0 \equiv \leftarrow A_1, \ldots, A_m$, $m > 0$, $A_1 \equiv f(\vec{t_1}) \approx x_1$, and let $\langle \theta_1, G_1 \rangle$ be derived from $\langle \theta_0, G_0 \rangle$ using clause $C^F \equiv A' \leftarrow A'_1, \ldots, A'_l$, $A' \equiv f(\vec{t'}) \approx s'$, and some $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t_1}))$. Then $G_1 \equiv \leftarrow \sigma'(A'_1, \ldots, A'_l, A_2, \ldots, A_m)$ with $\theta_1 = \sigma' \circ \theta_0$ and $\sigma' = \tilde{\sigma} \cup \{x_1 \mapsto \tilde{\sigma} s'\}$. By the auxiliary induction hypothesis, $\theta_n Q_1$ is ground and $comp(P) \models \theta_n Q_1$, where Q_1 is the query corresponding to G_1, i.e., $Q_1 \equiv \sigma'(A'_1, \ldots, A'_l, A_2, \ldots, A_m)$.

We have $\theta_n \circ \theta_1 = \theta_n$ and $\rho \circ \theta = \theta_n \circ \theta_0 = \theta_n = \rho$ by Lemma 7.5.13. We also have $\rho Q \equiv \theta_n(A_1, \ldots, A_m) \equiv f(\theta_n \vec{t_1}) \approx \theta_n x_1, \theta_n(A_2, \ldots, A_m)$. Now $\theta_n \vec{t_1} = \vec{t_1}$ is ground, since $\vec{t_1}$ is ground and $\theta_n x_1 = \theta_n \theta_1 x_1 = \theta_n \tilde{\sigma} s'$. All the variables of $\tilde{\sigma} s'$ are amongst A'_1, $\ldots$, A'_l and since $\theta_n(A'_1, \ldots, A'_l)$ is ground, we have $\theta_n \tilde{\sigma} s'$ and hence $\theta_n x_1$ as ground. Thus ρQ is ground.

Next we have $C^{FD} \equiv f(\tilde{\vec{t'}}) \approx s' \leftarrow A'_1, \ldots, A'_l, nonmem(\vec{t'})$ is in $comp(P)$. Therefore

$$comp(P) \models f(\theta_n \tilde{\vec{t'}}) \approx \theta_n s' \leftarrow \theta_n(A'_1, \ldots, A'_l), \theta_n nonmem(\vec{t'}).$$

Now $\theta_n \tilde{\vec{t'}} = \theta_n \theta_1 \tilde{\vec{t'}} = \theta_n \sigma' \theta_0 \tilde{\vec{t'}} = \theta_n \sigma' \tilde{\vec{t'}}$ (since C^F is a variant) $= \theta_n \tilde{\sigma} \tilde{\vec{t'}} = \theta_n \vec{t_1}$ (by Cor. 7.4.4 and Prop. 7.4.1). Also, $\theta_n nonmem(\vec{t'}) = \theta_n \theta_1 nonmem(\vec{t'}) = \theta_n \sigma' \theta_0 nonmem(\vec{t'}) = \theta_n \sigma' nonmem(\vec{t'}) =$

$\theta_n \tilde{\sigma} nonmem(\vec{t'}) = \tilde{\sigma} nonmem(\vec{t'})$ since the latter is ground. By Cor. 7.4.4 $\sigma nonmem(\vec{t'})$ holds in $SetAx$, i.e., $\tilde{\sigma} nonmem(\vec{t'})$. Thus $comp(P) \models \theta_n nonmem(\vec{t'})$.

We have $\theta_n s' = \theta_n \sigma' s' = \theta_n \tilde{\sigma} s'$, and that $\theta_n x_1 = \theta_n \tilde{\sigma} s'$. So $\theta_n s' = \theta_n x_1$. Hence $comp(P) \models f(\theta_n \tilde{\vec{t'}}) \approx \theta_n s'$, i.e., $comp(P) \models f(\theta_n \vec{t_1}) \approx \theta_n x_1$, i.e., $comp(P) \models \theta_n A_1$. Therefore $comp(P) \models \rho Q$.

(ii): Let P^F have the $m \geq 0$ equational clauses

$$
\begin{array}{l}
f(\vec{t'_1}) \approx s'_1 \leftarrow F'_1 \\
\quad \vdots \\
f(\vec{t'_m}) \approx s'_m \leftarrow F'_m
\end{array}
$$

and the $n \geq 1$ subset clauses

$$
\begin{array}{l}
f(\vec{t_1}) \underset{\sim}{\supset} s_1 \leftarrow F_1 \\
\quad \vdots \\
f(\vec{t_n}) \underset{\sim}{\supset} s_n \leftarrow F_n.
\end{array}
$$

Here, let $\vec{x'_1} = Var(\vec{t'_1}), \ldots, \vec{x'_m} = Var(\vec{t'_m})$ and $\vec{x_1} = Var(\vec{t_1}), \ldots, \vec{x_n} = Var(\vec{t_n})$. Let $\vec{y_i}$ be the new variables introduced upon flattening the i-th subset clause, $1 \leq i \leq n$. Let the set-collecting tree arising out of these subset clauses be as below.

$$
\begin{array}{ccccc}
 & & \leftarrow f(\vec{t}) \underset{\sim}{\supset} x_0 & & \\
/ & \cdots & | & \cdots & \backslash \\
\langle \theta_{1,1}, \leftarrow \theta_{1,1} F_1 \rangle & & \langle \theta_{1,p_1}, \leftarrow \theta_{1,p_1} F_1 \rangle & & \langle \theta_{n,p_n}, \leftarrow \theta_{n,p_n} F_n \rangle \\
\vdots & & \vdots & & \vdots \\
| & & | & & | \\
\langle \rho_{1,1}, \Box \rangle & & \langle \rho_{1,p_1}, \Box \rangle & & \langle \rho_{n,p_n}, \Box \rangle
\end{array}
$$

In the following, let $1 \leq i \leq n$ and $0 \leq j \leq p_i$. Let $\theta'_{i,j} = \theta_{i,j} \lceil \vec{x_i}$. Each child of the root node of the tree is at the head of a successful derivation of rank 0. Hence, $\rho_{i,j} \circ \theta_{i,j} = \rho_{i,j}$ by Lemma

7.5.13, and by (i) above, $\rho_{i,j}F_i$ is ground, and $comp(P) \models \rho_{1,1}F_1 \wedge \cdots \wedge \rho_{n,p_n}F_n$.

Consider the $CAA(f)$ in $comp(P)$. We have

$$comp(P) \models f(\vec{t}) \approx s' \leftarrow cond\_args' \wedge cond\_welldef$$

where s', $cond\_finset'$, $cond\_args'$, and $cond\_welldef$ are as follows.

$s' \equiv \cup_{i=1}^{n} \bigcup\{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}$

$cond\_finset' \equiv \bigwedge_{i=1}^{n} \exists u(set(u) \wedge \forall w(w \in u \leftrightarrow \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)))$

$cond\_args' \equiv \neg\exists \vec{x'_1}(D'_1 \wedge \vec{t} = \tilde{\vec{t'_1}}) \wedge \cdots \wedge \neg\exists \vec{x'_m}(D'_m \wedge \vec{t} = \tilde{\vec{t'_m}})$

$cond\_welldef \equiv \bigwedge_{i=1}^{n} \forall \vec{x_i}(D_i \wedge \vec{t} = \tilde{\vec{t_i}} \rightarrow \exists \vec{y_i} F_i)$

Now $comp(P) \models cond\_finset'$, by Lemma 7.2.2. Also $SetAx \models cond\_args'$, by Defn. 7.5.5(c), Cor. 7.4.4, and Prop. 7.4.1. Next, we show that $comp(P) \models cond\_welldef$. By Cor. 7.4.4, we have

$$D_i \wedge \vec{t} = \tilde{\vec{t_i}} \leftrightarrow \mathcal{E}(\theta'_{i,1}) \vee \cdots \vee \mathcal{E}(\theta'_{i,p_i}).$$

Hence it is enough to show $comp(P) \models \forall \vec{x_i}(\mathcal{E}(\theta'_{i,j}) \rightarrow \exists \vec{y_i} F_i)$, i.e., $comp(P) \models \exists \vec{y_i} \theta_{i,j} F_i$. But this follows from $comp(P) \models \rho_{i,j}F_i$ above, since $\rho_{i,j}F_i = \rho_{i,j}\theta_{i,j}F_i$.

Thus we have $comp(P) \models f(\vec{t}) \approx s'$. Now we show that s' is ground, s is ground, and $comp(P) \models s' = s$, from whence it follows that $comp(P) \models f(\vec{t}) \approx s$. We have $\rho_{i,j}x_0 = \rho_{i,j}\theta_{i,j}x_0 = \rho_{i,j}(\theta'_{i,j}s_i) = \rho_{i,j}\theta_{i,j}s_i = \rho_{i,j}s_i$ using Defn. 7.5.3. Also,

$s' = \cup_{i=1}^{n} \bigcup\{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{t} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}$

$= \cup_{i=1}^{n} \bigcup\{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge F_i \wedge (\mathcal{E}(\theta'_{i,1}) \vee \cdots \vee \mathcal{E}(\theta'_{i,p_i})))\}$

$= \cup_{i=1}^{n} \bigcup(\{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge F_i \wedge \mathcal{E}(\theta'_{i,1}))\} \cup \cdots \cup \{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge F_i \wedge \mathcal{E}(\theta'_{i,p_i}))\})$

$= \cup_{i=1}^{n} \bigcup(\{w \mid \exists \vec{y_i}(w = \theta_{i,1}s_i \wedge \theta_{i,1}F_i)\} \cup \cdots \cup \{w \mid \exists \vec{y_i}(w = \theta_{i,p_i}s_i \wedge \theta_{i,p_i}F_i)\})$

$= \cup_{i=1}^{n} \bigcup(\{\rho_{i,1}s_i\} \cup \cdots \cup \{\rho_{i,p_i}s_i\})$ since the solution for w in

$\exists \vec{y_i}(w = \theta_{i,j}s_i \wedge \theta_{i,j}F_i)$ is unique by Lemma 6.6.5 and is $\rho_{i,j}s_i$.

$= \rho_{1,1}s_1 \cup \cdots \cup \rho_{n,p_n}s_n$ which is a union of ground terms

since $\rho_{i,j}F_i$ is ground and $\rho_{i,j}s_i = \rho_{i,j}\theta_{i,j}s_i$.

$= \rho_{1,1}x_0 \cup \cdots \cup \rho_{n,p_n}x_0$

$= \bigcup\{\sigma x_0 \mid \langle\sigma,\square\rangle \text{ a leaf node}\} = s.$

Induction step: $k+1 > 0$, $k \geq 0$. We show the statements (i) and (ii) for rank $k+1$. Again, for (i), we need to do an auxiliary induction on the length of the successful derivation, and in (ii) we need the result from (i).

(i): Auxiliary basis step: $n = 0$. This is the same as in the above (auxiliary) basis step. Auxiliary induction step: $n > 0$. Let $\langle\theta_0, G_0\rangle$, $\langle\theta_1, G_1\rangle$, $\ldots$, $\langle\theta_n,\square\rangle$ be the sequence of subst-goal pairs in the successful derivation. So $\langle\theta_0, G_0\rangle \equiv \langle\theta, G\rangle$ and $\theta_n = \rho$. Consider the first derived subst-goal pair $\langle\theta_1, G_1\rangle$. If it is derived of rank 0, then the remaining proof for this case is the same as in the above (main) basis step. (Note that the derivation from $\langle\theta_1, G_1\rangle$ is of rank $k+1$ but of length $n-1$.) If it is derived of rank > 0, then the remaining proof of this case is as follows.

Let $G_0 \equiv\leftarrow A_1, \ldots, A_m$, $m > 0$, $A_1 \equiv f(\vec{t_1}) \approx x_1$, and let $\langle\theta_1, G_1\rangle$ be derived from $\langle\theta_0, G_0\rangle$ using a set-collecting tree for $P^F \cup \{\leftarrow f(\vec{t_1}) \approx x_1\}$ of rank $\leq k$ returning answer set s. Then $G_1 \equiv\leftarrow \tilde{\sigma}(A_2, \ldots, A_m)$ with $\theta_1 = \tilde{\sigma} \circ \theta_0$ and $\sigma = \{x_1 \mapsto NoDups(s)\}$. Now the derivation from $\langle\theta_1, G_1\rangle$ is of rank $\leq k+1$ but of length $n-1$. So either by the main induction hypothesis or by the auxiliary induction hypothesis, $\theta_n Q_1$ is ground and $comp(P) \models \theta_n Q_1$, where Q_1 is the query corresponding to G_1, i.e., $Q_1 \equiv \tilde{\sigma}(A_2, \ldots, A_m)$.

As before we have $\theta_1 \circ \theta_0 = \theta_1$ and $\rho \circ \theta = \rho$. Also $\rho Q \equiv \theta_n(A_1, \ldots, A_m) \equiv f(\theta_n\vec{t_1}) \approx \theta_n x_1, \theta_n(A_2, \ldots, A_m)$. Now $\theta_n\vec{t_1} = \vec{t_1}$ is ground, since $\vec{t_1}$ is ground and $\theta_n x_1 = \theta_n\theta_1 x_1 = \theta_n\tilde{\sigma}x_1 = \widetilde{NoDups}(s)$ which is ground. Thus ρQ is ground.

Now to show $comp(P) \models \rho Q$. It is enough to show $comp(P) \models \theta_n A_1$. But $\theta_n A_1 \equiv f(\theta_n\vec{t_1}) \approx \theta_n x_1$, and $\theta_n\vec{t_1} = \vec{t_1}$ while $\theta_n x_1 = \widetilde{NoDups}(s) = NoDups(s) = s$ by Prop. 7.4.1. Thus we need to show that $comp(P) \models f(\vec{t_1}) \approx s$, which is true by the main induction hypothesis on the set-collecting tree.

(ii): This is almost the same as that in the above (main) basis step. The only difference is that in the given set-collecting tree, each child of the root node is at the head of a successful derivation of rank $\leq k+1$. ■

Corollary 7.7.2 Let $P \in \mathbf{L}_0$ and $G \in \mathbf{G}_0$. If θ is a computed answer for $P \cup \{G\}$ then θ is a correct answer for $P \cup \{G\}$.

We now examine the completeness direction of the equivalence between the declarative and operational semantics. We begin with a preparatory property.

Theorem 7.7.3 Let P be a consistent program in $\mathbf{L}_0$, $\langle \theta, \leftarrow A \rangle$ be a subst-goal pair with $A \equiv f(\vec{t}) \approx y$, and ρ be a ground substitution with $Dom(\rho) = \{y\}$. If $[\rho A] \in T_P \uparrow k$ for some $k \geq 0$, then $\langle \theta, \leftarrow A \rangle$ has a successful derivation with computed answer ρ' such that $\rho' \lceil \{y\} = \rho$.

Proof: By induction on k. We have that $\vec{t}$ is ground and $y \notin Dom(\theta)$ since $\langle \theta, \leftarrow A \rangle$ is a subst-goal pair. Base: $k = 0$. The theorem is vacuously true as $T_P \uparrow 0 = \emptyset$.

Induction step: $k > 0$. We have $[\rho A] \in T_P \uparrow k = T_P(T_P \uparrow (k-1)) = R_P(T_P \uparrow (k-1)) \cup S_P(T_P \uparrow (k-1))$. Case (a): $[\rho A] \in R_P(T_P \uparrow (k-1))$. So there is a ground instance of some equational clause C^{FD} in P^{FD} from which $[\rho A]$ arises. Let $C^{FD} \equiv f(\vec{t'}) \approx s' \leftarrow A'_1, \ldots, A'_l, nonmem(\vec{t'})$, $l \geq 0$, with $A'_1 \equiv f_1(\vec{t'_1}) \approx y'_1$, $\ldots$, $A'_l \equiv f_l(\vec{t'_l}) \approx y'_l$. Let $\vec{x'} = Var(\vec{t'})$ and $\vec{y'} = (y'_1, \ldots, y'_l)$. Here, $\vec{y'}$ are the variables introduced upon flattening clause C. Thus the variables of clause C^{FD} are $\vec{x'}$ and $\vec{y'}$. Let θ' be a ground substitution for $\vec{x'}$ and $\vec{y'}$ leading to the ground instance $f(\theta'\vec{t'}) \approx \theta's' \leftarrow \theta'A'_1, \ldots, \theta'A'_l, nonmem(\theta'\vec{t'})$ of C^{FD} such that $[f(\theta'\vec{t'}) \approx \theta's'] = [\rho A]$. So $\theta'\vec{t'} = \vec{t}$ under $SetAx$. Also $nonmem(\theta'\vec{t'})$ holds and $\{[\theta'A'_1], \ldots, [\theta'A'_l]\} \subseteq T_P \uparrow (k-1)$.

We have that $\theta' \lceil \{\vec{x'}\}$ is a matcher of $\vec{t'} = \vec{t}$. Hence it is the same as some $\sigma \in MatchTuple(\vec{t'}, NoDups(\vec{t}))$ by Theorem 4.5.3. We have for this σ that $\theta' \lceil \{\vec{x'}\} = \tilde{\sigma}$ and $\sigma' = \tilde{\sigma} \cup \{y \mapsto \tilde{\sigma}s'\}$. Thus we have the derivation step

$$\begin{array}{l} \langle \theta, \leftarrow f(\vec{t}) \approx y \rangle \;\equiv\; \langle \theta, \leftarrow \theta(f(\vec{t}) \approx y) \rangle \qquad \text{(since } \theta \text{ has no effect on } f(\vec{t}) \approx y) \\ \quad | \\ \langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle \;\equiv\; \langle \sigma' \circ \theta, \leftarrow \sigma'\theta(A'_1, \ldots, A'_l) \rangle \end{array}$$

Now the idea is that computing with $\langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle$ is related to $[\theta' A'_1]$, ..., $[\theta' A'_l]$, which are in $T_P \uparrow (k-1)$. So, by using the induction hypothesis on $k-1$, there are successful derivations for each of $\langle \sigma' \circ \theta, \leftarrow \sigma' A'_i \rangle$, $1 \leq i \leq l$. All these derivations can be put together to form a successful derivation from $\langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle$. A formalisation of this idea is given below, which is to do an auxiliary induction on i, $1 \leq i \leq l$.

The auxiliary induction statement is that for each i, $0 \leq i \leq l$, there is a derivation

$$\begin{array}{l} \langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle \\ \quad \vdots \\ \langle \rho'_i, \leftarrow \rho'_i(A'_{i+1}, \ldots, A'_l) \rangle \end{array}$$

such that $\rho'_i \lceil \{y'_1, \ldots, y'_i\} = \theta' \lceil \{y'_1, \ldots, y'_i\}$.

Auxiliary base: $i = 0$. So $\rho'_0 = \sigma' \circ \theta$ and $\rho'_0 \lceil \emptyset = \epsilon = \theta' \lceil \emptyset$. Auxiliary induction step: $i > 0$. By the auxiliary induction hypothesis there is a derivation

$$\begin{array}{l} \langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle \\ \quad \vdots \\ \langle \rho'_{i-1}, \leftarrow \rho'_{i-1}(A'_i, \ldots, A'_l) \rangle \end{array}$$

with $\rho'_{i-1} \lceil \{y'_1, \ldots, y'_{i-1}\} = \theta' \lceil \{y'_1, \ldots, y'_{i-1}\}$. Now $\langle \rho'_{i-1}, \leftarrow \sigma' A_i \rangle$ is a subst-goal pair since $\langle \rho'_{i-1}, \leftarrow \sigma'(A'_i, \ldots, A'_l) \rangle$ is one. Using $\rho'_{i-1} \circ \sigma' \circ \theta = \rho'_{i-1}$ (by Lemma 7.5.13) we get that $\rho'_{i-1} \lceil \{\vec{x'}\} = \tilde{\sigma} = \theta' \lceil \{\vec{x'}\}$.

Let $\theta'' = \theta' \lceil \{y'_i\}$. We have $\rho'_{i-1} A'_i = f_i(\rho'_{i-1} \vec{t'_i}) \approx y'_i = f_i(\theta' \vec{t'_i}) \approx y'_i$ and that $\theta' A'_i = \theta'' \rho'_{i-1} A'_i$. Hence $[\theta'' \rho'_{i-1} A'_i] \in T_P \uparrow (k-1)$. Now, by the main induction hypothesis, there is a successful derivation

$$\begin{array}{l} \langle \rho'_{i-1}, \leftarrow \rho'_{i-1} A'_i \rangle \\ \quad \vdots \\ \langle \rho'_i, \Box \rangle \end{array}$$

with $\rho'_i \lceil \{y'_i\} = \theta'' = \theta' \lceil \{y'_i\}$ and $\rho'_i \circ \rho'_{i-1} = \rho'_i$. So $\rho'_i \lceil \{y'_1, \ldots, y'_{i-1}\} = \rho'_{i-1} \lceil \{y'_1, \ldots, y'_{i-1}\} = \theta' \lceil \{y'_1, \ldots, y'_{i-1}\}$ and $\rho'_i \lceil \{y'_1, \ldots, y'_i\} = \theta' \lceil \{y'_1, \ldots, y'_i\}$. Thus we can put the two derivations together to get the derivation

$$\begin{array}{l} \langle \rho'_{i-1}, \leftarrow \rho'_{i-1}(A'_i, \ldots, A'_l) \rangle \\ \quad \vdots \\ \langle \rho'_i, \leftarrow \rho'_i(A'_{i+1}, \ldots, A'_l) \rangle \end{array}$$

such that $\rho'_i \lceil \{y'_1, \ldots, y'_i\} = \theta' \lceil \{y'_1, \ldots, y'_i\}$.

Finally, using the auxiliary induction statement, we have the successful derivation

$$\begin{array}{l} \langle \theta, \leftarrow A \rangle \\ \quad \vdots \\ \langle \sigma' \circ \theta, \leftarrow \sigma'(A'_1, \ldots, A'_l) \rangle \\ \quad \vdots \\ \langle \rho', \Box \rangle \end{array}$$

with $\rho' = \rho'_l$ and $\rho' \lceil \{y'_1, \ldots, y'_l\} = \theta' \lceil \{y'_1, \ldots, y'_l\}$. Now to show that $\rho' \lceil \{y\} = \rho$.

We have $\rho' \circ \sigma' \circ \theta = \rho'$. From this, it is easy to see that $\rho' \lceil \{\vec{x'}\} = \tilde{\sigma}$ and $\rho' \circ \tilde{\sigma} = \rho'$. So $\rho' \lceil \{y\} = \{y \mapsto \rho' \tilde{\sigma} s'\} = \{y \mapsto \rho' s'\}$. Now $\rho = \{y \mapsto \theta' s'\}$, and since $\rho' s' = \theta' s'$ under $SetAx$ by the following, we have $\rho' \lceil \{y\} = \rho$. The variables of s' are in $\vec{x'}$ and $y'_1, \ldots, y'_l$, and from above, we get $\rho' \lceil \{y'_1, \ldots, y'_l\} = \theta' \lceil \{y'_1, \ldots, y'_l\}$ and $\rho' \lceil \{\vec{x'}\} = \tilde{\sigma} = \theta' \lceil \{\vec{x'}\}$.

Case (b): $[\rho A] \in S_P(T_P \uparrow (k-1))$. So $[\rho A] \equiv [f(\vec{t}) \approx \rho y] = [f(\vec{t''}) \approx s]$ for some f that is collect-all definable, $s = \bigcup\{s' \mid [\vec{t''}] = [\vec{t'}], [f(\vec{t'}) \supsetsim s'] \in R_P(T_P \uparrow (k-1))\}$, and conditions (3) and (4) of definition of S_P holding, i.e., for any ground instance $\vec{t'}$ (with $nonmem(\vec{t'})$ holding) of the lhs of any equational clause of f in P^{FD}, $[\vec{t''}] \neq [\vec{t'}]$, for every ground instance $\vec{t'}$ (with $nonmem(\vec{t'})$ holding) of the lhs of every subset clause of f in P^{FD} such that $[\vec{t''}] = [\vec{t'}]$, there is a ground instance of the remaining variables of the same clause such that if $f(\vec{t'}) \supsetsim s' \leftarrow A_1, \ldots, A_l, nonmem(\vec{t'})$ is the ground instance of the clause then $\{[A_1], \ldots, [A_l]\} \subseteq T_P \uparrow (k-1)$.

By condition (3) $\vec{t}$ does not match with the lhs of the head of any equational clause for f (for if it did then so would $\vec{t''}$ match, leading to a contradiction). So we can begin to construct a SLDU tree for $P \cup \{\leftarrow f(\vec{t}) \approx y\}$. Let the $n \geq 1$ subset clauses for f in P^F be

$$f(\vec{t_i}) \supsetsim s_i \leftarrow F_i, \qquad 1 \leq i \leq n$$

where $F_i \equiv A_{i,1}, \ldots, A_{i,l_i}$. Let $\vec{x_i}$ be the variables in $\vec{t_i}$ and $\vec{y_i}$ be the variables of flattening in the i-th clause. Hence the children of the root node of the tree are

$$\begin{array}{ccccc} & & \leftarrow f(\vec{t}) \supsetsim z_0 & & \\ / & \cdots & | & \cdots & \backslash \\ \langle \theta_{1,1}, \leftarrow \theta_{1,1} F_1 \rangle & & \langle \theta_{1,p_1}, \leftarrow \theta_{1,p_1} F_1 \rangle & & \langle \theta_{n,p_n}, \leftarrow \theta_{n,p_n} F_n \rangle \end{array}$$

where for each $1 \leq i \leq n$, $1 \leq j \leq p_i$, $\theta_{i,j} = \sigma'_{i,j}$ with $MatchTuple(\vec{t_i}, NoDups(\vec{t})) = \{\sigma_{i,j} \mid 1 \leq j \leq p_i\}$ and $\sigma'_{i,j} = \tilde{\sigma_{i,j}} \cup \{z_0 \mapsto \tilde{\sigma_{i,j}} s_i\}$. We also have $\theta_{i,j}\vec{t_i} = \vec{t}$ and $nonmem(\theta_{i,j}\vec{t_i})$ holds. So by condition (4) there is a ground instance $\theta'_{i,j}$ of the remaining variables $\vec{y_i}$ such that if $\theta''_{i,j} = \theta_{i,j} \cup \theta'_{i,j}$ then $\{[\theta''_{i,j} A_{i,1}], \ldots, [\theta''_{i,j} A_{i,l_i}]\} \subseteq T_P \uparrow (k-1)$.

As in case (a) above, computing with $\langle \theta_{i,j}, \leftarrow \theta_{i,j} F_i \rangle$ is related to $[\theta''_{i,j} A_{i,1}]$, $\ldots$, $[\theta''_{i,j} A_{i,l_i}]$ which are in $T_P \uparrow (k-1)$. As before we can compute successful derivations

$$\begin{array}{l} \langle \theta_{i,j}, \leftarrow \theta_{i,j} F_i \rangle \\ \quad \vdots \\ \langle \rho_{i,j}, \Box \rangle \end{array}$$

such that $\rho_{i,j}\lceil\{\vec{y_i}\} = \theta''_{i,j}\lceil\{\vec{y_i}\}$ and $\rho_{i,j}\lceil\{\vec{x_i}\} = \theta''_{i,j}\lceil\{\vec{x_i}\}$. Hence $\rho_{i,j}z_0 = \rho_{i,j}s_i = \theta''_{i,j}s_i$.

Based on this SLDU tree we can construct the SLDU derivation

$$\langle \theta, \leftarrow f(\vec{t}) \approx y\rangle$$
$$\Big|$$
$$\langle \tilde{\sigma} \circ \theta, \Box\rangle$$

where $\sigma = \{y \mapsto NoDups(s'')\}$ and $s'' = \bigcup\{\sigma'' z_0 \mid \langle\sigma'', \Box\rangle$ is a leaf node $\} = \bigcup\{\rho_{i,j}z_0 \mid 1 \leq i \leq n, 1 \leq j \leq p_i\}$.

Now to establish that $\tilde{\sigma}\circ\theta\lceil\{y\} = \rho$, i.e., to show that $\tilde{\sigma}\theta y = \rho y$. We have $\tilde{\sigma}\theta y = s''$ using Prop. 7.4.1, and $\rho y = s$ (from $[f(\vec{t}) \approx \rho y] = [f(\vec{t''}) \approx s]$). Let $Set1 = \{\rho_{i,j}z_0 \mid 1 \leq i \leq n, 1 \leq j \leq p_i\}$ and $Set2 = \{s' \mid [\vec{t''}] = [\vec{t'}], [f(\vec{t'}) \underset{\sim}{\supset} s'] \in R_P(T_P \uparrow (k-1))\}$. It is enough to show that $Set1 = Set2$.

We have that if $\rho_{i,j}z_0 \in Set1$ then $\theta''_{i,j}s_i \in Set1$. We also have $f(\theta''_{i,j}\vec{t_i}) \underset{\sim}{\supset} \theta''_{i,j}s_i \in R_P(T_P \uparrow (k-1))$ and that $[\theta''_{i,j}\vec{t_i}] = [\vec{t}] = [\vec{t''}]$. Hence $\theta''_{i,j}s_i \in Set2$, and therefore $Set1 \subseteq Set2$.

Now let $s' \in Set2$ with $[\vec{t''}] = [\vec{t'}]$ and $[f(\vec{t'}) \underset{\sim}{\supset} s'] \in R_P(T_P \uparrow (k-1))$. So $[f(\vec{t'}) \underset{\sim}{\supset} s']$ arises out of the head of the ground instance of some subset clause of f, say the i-th subset clause. Since the ground instance of this clause leads to a solution, i.e., a matcher of $\vec{t_i} = \vec{t}$, hence it is the same as $\theta''_{i,j}$ for some $1 \leq j \leq p_i$. This follows by Thm. 4.5.3 and by uniqueness of flattening Lemma 6.6.5. Thus $s' = \theta''_{i,j}s_i \in Set1$ and therefore $Set2 \subseteq Set1$. ■

Finally, we have the completeness property of subset-equational programs.

Theorem 7.7.4 Let P be a consistent program in $\mathbf{L}_0$, and $G \in \mathbf{G}_0$. If θ is a correct answer for $P \cup \{G\}$ then θ is a computed answer for $P \cup \{G\}$.

Proof: Let $G^F \equiv\leftarrow A_1, \ldots, A_n$, $n \geq 0$. The proof is by induction on n. Let $A_i \equiv f_i(\vec{t_i}) \approx y_i$, $1 \leq i \leq n$. So $Var(G^F) = \{y_1, \ldots, y_n\}$ and $Dom(\theta) = \{y_1, \ldots, y_n\}$ with θ being a ground substitution.
Base: $n = 0$. So $G^F \equiv \Box$ and $\theta \equiv \epsilon$ since $Dom(\theta) = Var(G^F) = \emptyset$. Clearly the subst-goal pair $\langle\epsilon, \Box\rangle$

succeeds and θ is a computed answer.

Induction step: $n > 0$. We have $comp(P) \models \theta(A_1, \ldots, A_n)$. Let $\theta' = \theta\lceil\{y_1, \ldots, y_{n-1}\}$. So θ' is a correct answer for $P \cup \{\leftarrow A_1, \ldots, A_{n-1}\}$. By the induction hypothesis there is a successful derivation for $P \cup \{\leftarrow A_1, \ldots, A_{n-1}\}$ such as

$$\begin{array}{l} \langle \epsilon, \leftarrow A_1, \ldots, A_{n-1} \rangle \\ \quad \vdots \\ \langle \sigma, \Box \rangle \end{array}$$

with $\sigma\lceil\{y_1, \ldots, y_{n-1}\} = \theta'$. Hence, we can build the following derivation.

$$\begin{array}{l} \langle \epsilon, \leftarrow A_1, \ldots, A_n \rangle \\ \quad \vdots \\ \langle \sigma, \leftarrow \sigma A_n \rangle \end{array}$$

Consider $A_n \equiv f_n(\vec{t_n}) \approx y_n$. We have $\sigma\vec{t_n}$ is ground and $y_n \notin Dom(\sigma)$ since $\langle \sigma, \leftarrow \sigma A_n \rangle$ is a subst-goal pair. Let $\theta'' = \theta\lceil\{y_n\}$. Now θ'' is a correct answer for σA_n, i.e., $comp(P) \models \theta''\sigma A_n$, since $\theta''\sigma A_n = f_n(\sigma\vec{t_n}) \approx \theta'' y_n = f_n(\theta'\vec{t_n}) \approx \theta'' y = \theta A_n$. Hence $[\theta''\sigma A_n] \in T_P \uparrow \omega$, i.e., $[\theta''\sigma A_n] \in T_P \uparrow k$, for some $k \geq 0$. By Thm. 7.7.3 above, $\langle \sigma, \leftarrow \sigma A_n \rangle$ has a successful derivation with computed answer ρ' such that $\rho'\lceil\{y_n\} = \theta''$. Hence, we have the successful derivation

$$\begin{array}{l} \langle \epsilon, \leftarrow A_1, \ldots, A_n \rangle \\ \quad \vdots \\ \langle \sigma, \leftarrow \sigma A_n \rangle \\ \quad \vdots \\ \langle \rho', \Box \rangle \end{array}$$

with computed answer being $\rho'\lceil Var(G^F)$. Now, from $\rho' \circ \sigma = \rho'$ we get $\rho'\lceil\{y_1, \ldots, y_{n-1}\} = \sigma\lceil\{y_1, \ldots, y_{n-1}\} = \theta'$. Using this fact and the fact that $\rho'\lceil\{y_n\} = \theta''$ we get $\rho'\lceil\{y_1, \ldots, y_n\} = \theta$, i.e., $\rho' \lceil Var(G^F) = \theta$. Thus θ is a computed answer. ■

In the above theorem, we have shown that if θ is a correct answer for $P \cup \{G\}$ then there is a derivation computing θ. However, this particular derivation may not be the one followed in the computation procedure for the following reason. In resolving with an equational clause any one match of the lhs of the clause is considered in the derivation step, and this choice of the match may not be the one constructed in the proof of the above theorem. Hence, we need to establish that computed answers in derivations are independent of the particular matches chosen at equationally derived steps. This does not hold for all consistent programs, e.g., the program P below.

$$\begin{aligned} P: \quad & f(\{x \backslash y\}) \approx g(x) \\ & g(1) \approx 1 \end{aligned}$$

Here, $f(\{1, 2\})$ depends on $g(1)$ or $g(2)$, only one of which is defined. Hence, the match $\{x \mapsto 2, y \mapsto \{1\}\}$ leads to incompleteness in the semantics.

An enhanced computation strategy that explores a tree of derivations based on *all* equational matches is bound to be a complete procedure. However, this is computationally expensive, and we do not pursue it. It is desirable then to characterize the programs for which our given strategy is a complete procedure. We have a fixpoint characterization, but do not readily see any relatively simple declarative characterization of such programs. Also, any such characterization is very likely to be undecidable. Hence, we have not explored this issue any further.

8 Subset-Equational Language: An Enhanced Semantics

This chapter gives an enhanced semantics that provides an appropriate treatment for database like programs. Such programs typically contain incomplete information. Whereas the previous semantics would give no answers for queries for which the database contained only partial information, the present semantics gives answers that provide as much information as is present in the database. For functions unaffected by such database considerations, the previous semantics carries over.

Below, we give the revised declarative and operational semantics. The declarative semantics is based on the *Clark completion* in logic programming. Here, the syntax of functional clauses permits a simplification in the syntactic statement of the completion that makes it readable and usable.

On the operational side, the additional factor is finite failure. As with finite failure for definite clause programs we need fair derivations. While the soundness property holds for them, unlike the definite clause case, the completeness property does not. Nevertheless, the operational semantics computes more than provided by the semantics in [JP89].

Our description below will be largely informal since the steps to formalisation are clear in light of the rigorous development in previous chapters.

8.1 Syntax

The syntax carries over from the previous chapter, except that we need to make a further distinction within the collect-all definable functions Σ_U. Let Σ_U be partitioned into Σ_A and Σ_{nA}. The function symbols in Σ_A are meant for defining functions that aggregate as much information as is present in programs. The function symbols in Σ_{nA} are meant for defining functions which have values only when complete information about the function is present in the program. A function f is called **aggregating** if $f \in \Sigma_A$ and the symbols in Σ_{nA} are called non-aggregating. The aggregating functions are expected to be the often-used variety by programmers as compared to the non-aggregating functions.

Consider program $\mathtt{P}_1$ below, which is an augmented version of program $\mathtt{P}_4$ of section §7.5. In it, `father`, `mother` and `bothparents` are non-collect-all definable (in Σ_{nU}), and among the rest of the collect-all definable functions, `parents` and `ancestors` are aggregating (in Σ_A) while `twoparents` is non-aggregating (in Σ_{nA}).

$\mathtt{P}_1$: `father(Bob)` $\approx$ `Mark`   `mother(Bob)` $\approx$ `Mary`

`father(Ann)` $\approx$ `Mark`   `mother(Ann)` $\approx$ `Mary`

`father(Mark)` $\approx$ `Joe`   `mother(Mark)` $\approx$ `Jane`

`father(Joe)` $\approx$ `John`   `mother(Jane)` $\approx$ `Meg`

`parents(x)` $\underset{\sim}{\supseteq}$ {`father(x)`}

`parents(x)` $\underset{\sim}{\supseteq}$ {`mother(x)`}

`ancestors(x)` $\underset{\sim}{\supseteq}$ `parents(x)`

`ancestors(x)` $\underset{\sim}{\supseteq}$ `ancestors(father(x))`

`ancestors(x)` $\underset{\sim}{\supseteq}$ `ancestors(mother(x))`

`bothparents(x)` $\approx$ {`father(x), mother(x)`}

`twoparents(x)` $\underset{\sim}{\supseteq}$ {`father(x)`}

`twoparents(x)` $\underset{\sim}{\supseteq}$ {`mother(x)`}

Here the database contains incomplete information regarding the parents of `Joe` and `Jane`, among others. Yet, for the query `parents(Joe)` $\approx$ `x`, we might be satisfied with the answer `x` $\mapsto$ {`John`}, or for the query `ancestors(Mark)` $\approx$ `x`, we might seek the answer `x` $\mapsto$ {`Joe, Jane,` `John, Meg`}. Both queries would have no answers in the previous semantics, since some of the subexpressions they depend on are undefined. The query `bothparents(Joe)` $\approx$ `x`, however, has no answer in the present semantics since `bothparents` is not aggregating. Similarly, the query `twoparents(Joe)` $\approx$ `x` has no answer since `twoparents` is not aggregating. The purpose of `bothparents` and `twoparents` is to show that their semantics does not differ whether one chooses to represent their functions through equations or through non-aggregating subset assertions.

8.2 Completion

The completion of a subset-equational program is now influenced by the *Clark completion* ([Llo87] §14) of logic programming. The basic idea of the Clark completion is to form if-and-only-if definitions for each predicate symbol from the given clauses for that predicate symbol. Using such definitions both positive and negative logical consequences can be derived.

Let us call such a completion in the subset-equational case as a IFF-completion. Due to the non-overlapping nature of the different clauses of a function symbol the IFF-completion takes a simpler syntactic form than it does in the usual relational case. This can be seen from the example below.

Let a logic program have the following two clauses for a predicate p with distinct data constructors a and b.

$$p(a) \leftarrow p(b), q(a)$$

$$p(b) \leftarrow q(b)$$

The completed definition of p is

$$\forall x(p(x) \leftrightarrow x = a \wedge p(b) \wedge q(a) \vee x = b \wedge q(b)).$$

From the fact that a is distinct from b, an equivalent completion is

$$p(a) \leftrightarrow p(b) \wedge q(a)$$

$$p(b) \leftrightarrow q(b)$$

$$\forall x(\neg p(x) \leftarrow x \neq a \wedge x \neq b)$$

Note that each clause gives rise to an if-and-only-if formula in the equivalent completion, followed by a single clause that takes care of all the cases not covered by the other if-and-only-if formulae.

In the functional context, an equational clause such as

$$f(\vec{t}) \approx s \leftarrow B$$

with body B is transformed to an equivalent clause

$$f(\vec{t}) \approx w' \leftarrow \exists \vec{y}(w' = s \wedge B)$$

where w' is a new variable and $\vec{y}$ are all the variables not in the head of the clause. (If s was already a variable there is no need to introduce w', since the idea of w' is to get the rhs of the head atom as a variable.) We next complete the clause as

$$f(\vec{t}) \approx w' \leftrightarrow \exists \vec{y}(w' = s \wedge B)$$

The purpose of the completion is to enable to deduce $f(\vec{t}) \not\approx w'$ for any w' when $f(\vec{t})$ is undefined. We show later why it is desirable to do so through an example in §8.3. Hence it is not enough to complete the clause *before* the introduction of w', as in

$$f(\vec{t}) \approx s \leftrightarrow \exists \vec{z} B$$

where $\vec{z}$ are the variables not in the head. For if $\neg\exists\vec{z}B$ holds then we can deduce $f(\vec{t}) \not\approx s$ but may not be able to deduce $f(\vec{t}) \not\approx w'$ for all w'. By introducing w' we are capable of doing the latter. We show this through the **bothparents** example later in this section.

We now give the IFF-completions for a function symbol f based on its kind in the alphabet. We assume that clauses have undergone the flattening and disjointness transforms of §7.2 represented by F and D respectively. Also, we assume that the collect-all set s, *cond_finset*, *cond_args*, and *cond_welldef* are as in §7.2. We have kept to the usage of variable names as in §7.2.

For f that is not collect-all definable, i.e., $f \in \Sigma_{nU}$, each j-th equational clause of f

$$f(\tilde{\vec{t_j'}}) \approx s_j' \leftarrow F_j', D_j'$$

translates to

$$f(\tilde{\vec{t'_j}}) \approx w' \leftrightarrow \exists \vec{y'_j}(w' = s'_j \wedge F'_j \wedge D'_j)$$

where $\vec{y'_j}$ are the new variables introduced upon flattening. Finally, the following additional clause is added for f.

$$f(\vec{v}) \not\approx w' \leftarrow cond\_args.$$

It covers the definition of f for all argument values $\vec{v}$ not covered by the lhs of the given equational clauses of f.

For f that is collect-all definable but not aggregating, i.e., $f \in \Sigma_{nA}$, each j-th equational clause of f

$$f(\tilde{\vec{t'_j}}) \approx s'_j \leftarrow F'_j, D'_j$$

translates as before to

$$f(\tilde{\vec{t'_j}}) \approx w' \leftrightarrow \exists \vec{y'_j}(w' = s'_j \wedge F'_j \wedge D'_j)$$

The subset clauses for f lead to

$$(f(\vec{v}) \approx w' \leftrightarrow w' = s \wedge cond\_finset \wedge cond\_welldef) \leftarrow cond\_args.$$

For f that is collect-all definable and aggregating, i.e., $f \in \Sigma_A$, each j-th equational clause of f

$$f(\tilde{\vec{t'_j}}) \approx s'_j \leftarrow F'_j, D'_j$$

translates as before to

$$f(\tilde{\vec{t'_j}}) \approx w' \leftrightarrow \exists \vec{y'_j}(w' = s'_j \wedge F'_j \wedge D'_j)$$

The subset clauses for f lead to

$$(f(\vec{v}) \approx w' \leftrightarrow w' = s \wedge cond\_finset) \leftarrow cond\_args.$$

For a subset-equational program P, let P^{FD}_{IFF} be P^{FD} after replacing all its equational clauses with their IFF-completions. Let $CAA_{IFF}(P)$ be the IFF-completions arising from the subset assertions.

Definition 8.2.1 The **IFF-completion** of a program P, called $comp_{IFF}(P)$, is given by

$$comp_{IFF}(P) = P_{IFF}^{FD} \cup CAA_{IFF}(P) \cup SetAx \cup FunAx$$

Queries and goals undergo the same transformation as before, namely, just the flattening transformation.

For the program $\mathtt{P}_1$, we have the IFF-completions of `father`, `parents`, `bothparents` and `twoparents` as follows. As usual, we omit the condition *cond_finset*.

$$father(Bob) \approx y \leftrightarrow y = Mark$$

$$father(Ann) \approx y \leftrightarrow y = Mark$$

$$father(Mark) \approx y \leftrightarrow y = Joe$$

$$father(Joe) \approx y \leftrightarrow y = John$$

$$father(v) \not\approx y \leftarrow v \neq Bob \wedge v \neq Ann \wedge v \neq Mark \wedge v \neq Joe$$

$$parents(x) \approx y \leftrightarrow y = \bigcup\{\{w\} \mid father(x) \approx w\} \cup \bigcup\{\{w\} \mid mother(x) \approx w\}$$

$$bothparents(x) \approx y \leftrightarrow \exists z_1, z_2(y = \{z_1, z_2\} \wedge father(x) \approx z_1 \wedge mother(x) \approx z_2)$$

$$\begin{aligned} twoparents(x) \approx y \leftrightarrow y = &\bigcup\{\{w\} \mid father(x) \approx w\} \cup \bigcup\{\{w\} \mid mother(x) \approx w\} \\ &\wedge \exists w(father(x) \approx w) \wedge \exists w(mother(x) \approx w) \end{aligned}$$

In the case of *bothparents*, we omit the clause

$$bothparents(v) \not\approx y \leftarrow cond\_args$$

since $cond\_args \leftrightarrow false$ here, making the clause equivalent to $true$. Intuitively, the lhs variable x of the single equational clause for *bothparents* covers all the argument values and hence the above clause is unnecessary. Lastly, we show why is not enough to complete *bothparents* as

$$bothparents(x) \approx \{z_1, z_2\} \leftrightarrow father(x) \approx z_1 \wedge mother(x) \approx z_2$$

Since $father(Jane)$ is undefined in the database, we have $\forall y(father(Jane) \not\approx y)$, which obtains from the last clause in the IFF-completion of $father$. Hence for any z_1, z_2, we obtain

$\forall z_1, z_2(bothparents(Jane) \not\approx \{z_1, z_2\})$. However, we are unable to deduce $\forall y(bothparents(Jane) \not\approx y)$.

8.3 Declarative Semantics

We define the declarative semantics as before in terms of logical consequence from the completion $comp_{IFF}(P)$.

Definition 8.3.1 The **declarative semantics** $\mathcal{D}_P$ of P is given by:

$$\mathcal{D}_P = \{A \mid comp_{IFF}(P) \models A, A \text{ a ground program atom (of the form } f(\vec{t}) \approx t' \text{ or } f(\vec{t}) \supsetsim t')\}$$

Again, we are interested in atoms of the form $f(\vec{t}) \approx t'$, with the interest in atoms of the form $f(\vec{t}) \supsetsim t'$ being technical.

Since the IFF-completions are not in definite clause form we do not expect a parallel development of the declarative semantics of the previous chapter, such as reducing logical consequence from $comp_{IFF}(P)$ to Herbrand logical consequence.

We show the desirability of deducing negative logical consequences using the following program. Consider P and $comp_{IFF}(P)$ below, and let f be aggregating and g be non-collect-all definable.

P: $\quad f(x) \supsetsim g(1)$

P^{FD}: $\quad f(x) \supsetsim y \leftarrow g(1) \approx y$

$comp_{IFF}(P)$:

$$g(x) \not\approx y$$

$$f(x) \supsetsim y \leftarrow g(1) \approx y$$

$$f(x) \approx z \leftrightarrow z = \bigcup\{y \mid g(1) \approx y\}$$

By being able to deduce $g(1) \not\approx y$ for any y, we obtain $\bigcup\{y \mid g(1) \approx y\} = \emptyset$ as a logical consequence of $comp_{IFF}(P)$. This gives $comp_{IFF}(P) \models f(1) \approx \emptyset$ whereas $f(1)$ does not have a value in $comp(P)$. (The latter follows easily from Thm. 7.6.12 of the fixpoint semantics in §7.6.)

The next property shows that the present semantics augments the previous.

Proposition 8.3.2 Let P be a subset-equational program and $f(\vec{t}) \approx t'$ be a ground atom. Then,

$$comp(P) \models f(\vec{t}) \approx t' \;\Rightarrow\; comp_{IFF}(P) \models f(\vec{t}) \approx t'$$

Proof: Using the T_P semantics of the previous chapter. Show by induction that for all $k \geq 0$, if $[f(\vec{t}) \approx t'] \in T_P \uparrow k$ then $comp_{IFF}(P) \models f(\vec{t}) \approx t'$. ∎

Lastly, as before, for arbitrary queries (that are not necessarily ground atoms), we have the following declarative notion of a correct answer.

Definition 8.3.3 Let $P \in \mathbf{L}_0$ and $G \in \mathbf{G}_0$ with Q its corresponding query. Let θ be a substitution with $Dom(\theta) = Var(G^F)$. Then θ is a **correct answer** for $P \cup \{G\}$ if $comp(P) \models \forall \theta Q^F$.

Again, if θ is a correct answer, then it is a ground substitution. Hence, $\forall \theta Q^F \equiv \theta Q^F$.

8.4 Operational Semantics

The operational semantics is a now influenced by *finite failure* ([Llo87] §13) and we describe below how the computation procedure incorporates it. We call it **SLDUFF-resolution** for SLDU-resolution with Finite Failure. As before, it computes with the flattened forms of programs and queries.

There are two kinds of ways by which finite failure can occur. One aspect of it is shown by the following example. Consider P_1^F, of which a fragment is given below, together with the query Q^F seeking the paternal grandparents of $Mark$.

P_1^F : $father(Bob) \approx Mark \leftarrow$

$father(Ann) \approx Mark \leftarrow$

$father(Mark) \approx Joe \leftarrow$

$$father(Joe) \approx John \leftarrow$$

$$mother(Bob) \approx Mary \leftarrow$$

$$mother(Ann) \approx Mary \leftarrow$$

$$mother(Mark) \approx Jane \leftarrow$$

$$mother(Jane) \approx Meg \leftarrow$$

$$parents(x) \mathrel{\underset{\sim}{\supset}} \{w\} \leftarrow father(x) \approx w$$

$$parents(x) \mathrel{\underset{\sim}{\supset}} \{w\} \leftarrow mother(x) \approx w$$

$$Q^F : \quad father(Mark) \approx y, parents(y) \approx x$$

The computation for the query has the following derivation and set-collecting tree, where we assume that the variants of clauses used for resolution will be apparent. The tree has one finitely failed branched which we denote by $\otimes$. Also, in each derivation step, we have underlined the atom being resolved, since in general we mean to depart from the previous strategy of always resolving the leftmost atom.

$$\langle \theta_0, G_0 \rangle \equiv \langle \epsilon, \leftarrow \underline{father(Mark) \approx y}, parents(y) \approx x \rangle$$

$$\theta = \{y \mapsto Joe\}$$

$$\langle \theta_1, G_1 \rangle \equiv \langle \theta, \leftarrow \underline{parents(Joe) \approx x} \rangle$$

$$\sigma = \{x \mapsto s\} = \{x \mapsto \{John\}\} \quad \text{(from set-collecting tree below)}$$

$$\langle \theta_2, G_2 \rangle \equiv \langle \sigma \circ \theta, \Box \rangle$$

So the computed answer is θ_2 which is $\{y \mapsto Joe, x \mapsto \{John\}\}$. The set collecting tree for P_1^F and $\{\leftarrow parents(Joe) \approx x\}$ is given next.

$$\leftarrow \underline{parents(Joe) \mathrel{\underset{\sim}{\supset}} x_0}$$

$$\sigma_1 = \{x_1 \mapsto Joe, x_0 \mapsto \{y_1\}\} \qquad \sigma_2 = \{x_2 \mapsto Joe, x_0 \mapsto \{y_2\}\}$$

$$
\begin{array}{ccc}
 & \langle \sigma_1, \leftarrow \underline{father(Joe) \approx y_1} \rangle & \qquad \underbrace{\langle \sigma_2, \leftarrow \underline{mother(Joe) \approx y_2} \rangle}_{\otimes} \\
\sigma_3 = \{y_1 \mapsto John\} & \Big| & \\
 & \langle \sigma_3 \circ \sigma_1, \Box \rangle &
\end{array}
$$

$$\text{answer set } s = \sigma_3 \circ \sigma_1(x_0) = \{John\}$$

Here, the finite failure occurred when the goal for $mother$, which is a non-collect-all definable function (i.e., in Σ_{nU}), failed to match with any of the clauses for $mother$. The other way that finite failure occurs is when a non-aggregating function (i.e., in Σ_{nA}) is resolved and it creates an auxiliary set-collecting tree at least one of whose branches is finitely failed. This is shown in the next example. Assume that $h, h' \in \Sigma_{nU}$, $g \in \Sigma_{nA}$, and $f \in \Sigma_A$.

$$
\begin{array}{llll}
P: & h'(x) \approx h'([x]) & P^F: & h'(x) \approx y \leftarrow h'([x]) \approx y \\
 & g(x) \supsetsim h'(x) & & g(x) \supsetsim y \leftarrow h'(x) \approx y \\
 & g(x) \supsetsim h(x) & & g(x) \supsetsim y \leftarrow h(x) \approx y \\
 & f(x) \supsetsim \{1\} & & f(x) \supsetsim \{1\} \\
 & f(x) \supsetsim g(x) & & f(x) \supsetsim y \leftarrow g(x) \approx y
\end{array}
$$

Now consider the query $Q \equiv f(1) \approx x$. Here $Q^F \equiv Q$. A SLDUFF derivation for $P \cup \{\leftarrow Q\}$ is as below. From it, we get the computed answer to be $\theta_1 \lceil \{x\} = \theta_1 = \{x \mapsto \{1\}\}$.

$$
\begin{array}{l}
\langle \epsilon, \underline{\leftarrow f(1) \approx x} \rangle \\
\Big| \; \theta_1 = \{x \mapsto NoDups(s)\} = \{x \mapsto \{1\}\} \quad \text{(from tree } T_1 \text{ below)} \\
\langle \theta_1, \Box \rangle
\end{array}
$$

Below is the set-collecting tree T_1 for $P^F \cup \{\leftarrow f(1) \approx x\}$ returning answer set $s = \{1\}$. Its right branch is the failed branch since the auxiliary tree T_2 below has at least one failed branch.

$$\leftarrow \underline{f(1) \supsetsim v_0}$$

$$\sigma_1 = \{x_1 \mapsto 1, v_0 \mapsto \{1\}\} \qquad \sigma_2 = \{x_2 \mapsto 1, v_0 \mapsto y_2\}$$

$$\langle \sigma_1, \Box \rangle \qquad \underbrace{\langle \sigma_2, \leftarrow \underline{g(1) \approx y_2} \rangle}_{\otimes}$$

answer set $s = \sigma_1(v_0) = \{1\}$

Below is the tree T_2 for $P^F \cup \{\leftarrow g(1) \approx y_2\}$ which fails to return an answer because it has a failed branch.

$$\leftarrow \underline{g(1) \gtrsim v_1}$$

$$\sigma_3 = \{x_3 \mapsto 1, v_1 \mapsto y_3\} \qquad \sigma_4 = \{x_4 \mapsto 1, v_1 \mapsto y_4\}$$

$$\langle \sigma_3, \leftarrow \underline{h'(1) \approx y_3} \rangle \qquad \underbrace{\langle \sigma_4, \leftarrow \underline{h(1) \approx y_4} \rangle}_{\otimes}$$

$$\sigma_5 = \{x_5 \mapsto 1, y_3 \mapsto y_5\}$$

$$\langle \sigma_5 \circ \sigma_3, \leftarrow \underline{h'([1]) \approx y_5} \rangle$$

$$\vdots$$

Notice that the auxiliary tree T_2 is not finite as the left branch is non-terminating. Here, the finitely failed node for g in T_1 contributed meaningfully to the set-collecting tree T_1. However, this kind of finite failure is unlikely to occur since it means that an aggregating function f depends on the tree of a non-aggregating function g, thereby contradicting the sense of an aggregating function.

A drawback with a tree of kind T_2 is that a 'breadth-first' computation of the tree is required to identify the failing branch. It is well-known that a breadth-first computation is computationally expensive and hence some less expensive traversal of the tree such as 'depth-first' execution is done even if it should lead to incompleteness in the semantics. The latter is acceptable in our case since, as was pointed out above, this kind of finite failure is unlikely to occur in practise.

We next give an informal sketch of the definitions required to describe the operational semantics. Previously, we had mutually recursive definitions of *set-collecting SLDU tree of rank k* and *derived of rank* k. Now, we also incorporate the definition of *finitely failed of rank* k. We will give these definitions in terms of a *selection rule* R which at each step of a derivation selects an atom to be resolved whose lhs is ground. This is so that we may differ from always selecting the leftmost atom.

The definition of *derived of rank* k is as before. The definition of *set-collecting tree of rank* k must now distinguish between aggregating and non-aggregating functions at its root. For the former case, the leaves can be subst-goal pairs that either contain the empty clause $\Box$ or are finitely failed of rank k. In the latter case, each leaf *must* be a subst-goal pair containing the empty clause.

A subst-goal pair $\langle \theta, G \rangle$ is *finitely failed of rank* 0 if the function symbol f of the selected atom is non-collect-all definable and the atom does not match with any equational clause of f. A subst-goal pair is *finitely failed of rank* $k+1$ if there is a finite tree for the selected atom all of whose non-root nodes are derived of rank k and the tree has a leaf that is finitely failed of rank k. (Note that the other leaves can be subst-goal pairs that are neither finitely failed of rank k nor contain the empty clause.)

The development of the rest of the definitions proceeds as in §7.5.

We require that the selection rule R lead to *fair* derivations. In a fair derivation, R seeks to select every atom whose lhs is ground within a finite number of steps. This would provide the most opportunity to identify finite failure.

Finally, we show that our semantics computes more than that in [JP89]. Consider the following program P in which $g, h, h' \in \Sigma_{nU}$ and $f \in \Sigma_A$.

$P:$	$h'(x) \approx h'([x])$	$P^F:$	$h'(x) \approx y \leftarrow h'([x]) \approx y$
	$g(x) \approx [h(x), h'(x)]$		$g(x) \approx [z_1, z_2] \leftarrow h(x) \approx z_1, h'(x) \approx z_2$
	$f(x) \supsetsim \{1\}$		$f(x) \supsetsim \{1\}$
	$f(x) \supsetsim \{g(x)\}$		$f(x) \supsetsim \{y\} \leftarrow g(x) \approx y$

Now consider the query $Q \equiv f(1) \approx x$. Here $Q^F \equiv Q$. A SLDUFF derivation for $P \cup \{\leftarrow Q\}$ is as below.

$$\langle \epsilon, \leftarrow \underline{f(1) \approx x} \rangle$$
$$\Big| \; \theta_1 = \{x \mapsto NoDups(s)\} = \{x \mapsto \{1\}\} \quad \text{(from tree } T_1 \text{ below)}$$
$$\langle \theta_1, \Box \rangle$$

From it, we get the computed answer to be $\theta_1 \lceil \{x\} = \theta_1 = \{x \mapsto \{1\}\}$. Below is the set-collecting tree T_1 for $P^F \cup \{\leftarrow f(1) \approx x\}$ returning answer set $s = \{1\}$.

$$\leftarrow \underline{f(1) \underset{\sim}{\supset} v_0}$$
$$\sigma_1 = \{x_1 \mapsto 1, v_0 \mapsto \{1\}\} \;/ \qquad \backslash\; \sigma_2 = \{x_2 \mapsto 1, v_0 \mapsto \{y_2\}\}$$
$$\langle \sigma_1, \Box \rangle \qquad \langle \sigma_2, \leftarrow \underline{g(1) \approx y_2} \rangle$$
$$\Big| \; \sigma_3 = \{x_3 \mapsto 1, y_2 \mapsto \{z_1, z_2\}\}$$
$$\underbrace{\langle \sigma_3 \circ \sigma_2, \leftarrow \underline{h(1) \approx z_1}, h'(1) \approx z_2 \rangle}_{\otimes}$$
$$\text{answer set } s = \sigma_1(v_0) = \{1\}$$

Note that the difference with [JP89] is not simply on account of having a fair selection rule as against a leftmost selection rule. The principal difference is on account of the different ways that finite failure is handled. In our case, failure leads to termination of the failing branch preventing further processing of the failed node. In [JP89], the goal $\leftarrow \underline{h(1) \approx z_1}, h'(1) \approx z_2$ would *succeed* with a value of $z_1 \approx ?$ and the atom $h'(1) \approx z_2$ would be taken up for further processing. This would lead to non-termination and hence to no computed answer.

8.5 Soundness and Completeness

We now show that soundness, as can be expected, holds but completeness does not. The soundness property follows as a corollary of the following theorem.

Theorem 8.5.1 Let $P \in \mathbf{L}_0$, $\langle \theta, G \rangle$ be a subst-goal pair, and $\vec{t}$ be ground. Let query Q correspond to G. We have the following:

(i) If ρ is the computed answer in a successful derivation from $\langle \theta, G \rangle$ using P^F, then ρQ is ground and $comp_{IFF}(P) \models \rho Q$.

(ii) If there is a finitely failed derivation from $\langle \theta, G \rangle$ using P^F, then $comp_{IFF}(P) \models \neg \exists Q$.

(iii) If s is the answer set returned by a set-collecting tree for $P \cup \{\leftarrow f(\vec{t}) \approx x\}$, then s is ground and $comp_{IFF}(P) \models f(\vec{t}) \approx s$.

Proof: The three parts, (i), (ii) and (iii), are proved by simultaneous induction on k, where in (i), k is the rank of the successful derivation, in (ii), is the rank of the finitely failed node, and in (iii), is the rank of the set-collecting tree. ■

Corollary 8.5.2 Let $P \in \mathbf{L}_0$ and $G \in \mathbf{G}_0$. If θ is a computed answer for $P \cup \{G\}$ then θ is a correct answer for $P \cup \{G\}$.

Now we turn to addressing the incompleteness of the semantics. As mentioned before, in §8.4, incompleteness could arise if a compromise like 'depth-first' rather than 'breadth-first' execution is made in the trees for non-aggregating functions, for such a scheme might fail to detect a failing branch. There is, however, a more important reason why incompleteness occurs which is that matching rather than unification is used as the heart of the computation procedure. This means that a fair selection rule can select only atoms in goals whose lhs are ground. It is unable to explore finite failure that might occur for those atoms whose lhs are not ground. This is illustrated in the following example.

Consider the following program P in which $h, h' \in \Sigma_{nU}$ and $f \in \Sigma_A$.

$$P: \quad h'(x) \approx h'([x]) \qquad\qquad P^F: \quad h'(x) \approx y \leftarrow h'([x]) \approx y$$

$$f(x) \supseteq \{1\} \qquad\qquad f(x) \supseteq \{1\}$$

$$f(x) \mathrel{\underset{\sim}{\supset}} h(h'(x)) \qquad\qquad f(x) \mathrel{\underset{\sim}{\supset}} y \leftarrow h'(x) \approx z, h(z) \approx y$$

$comp_{IFF}(P)$:

$$h(x) \not\approx y$$

$$h'(x) \approx y \leftrightarrow h'([x]) \approx y$$

$$f(x) \mathrel{\underset{\sim}{\supset}} \{1\}$$

$$f(x) \mathrel{\underset{\sim}{\supset}} y \leftarrow h'(x) \approx z, h(z) \approx y$$

$$f(x) \approx y \leftrightarrow y = \bigcup\{\{1\} \mid true\} \cup \bigcup\{y \mid h'(x) \approx z, h(z) \approx y\}$$

We have that $\bigcup\{\{1\} \mid true\} = \{1\}$ and $\bigcup\{y \mid h'(x) \approx z, h(z) \approx y\} = \bigcup\emptyset = \emptyset$ in the presence of $comp_{IFF}(P)$. The latter follows from the completed clause for h. Hence, declaratively, we have $comp_{IFF}(P) \models f(1) \approx \{1\}$. However, the query $f(1) \approx x$ is unable to obtain a computed answer since its set-collecting tree does not terminate. The problem arises because the atom for h is never resolved since its argument is never ground.

$$\leftarrow \underline{f(1) \mathrel{\underset{\sim}{\supset}} v_0}$$

$$\sigma_1 = \{x_1 \mapsto 1, v_0 \mapsto \{1\}\} \qquad\qquad \sigma_2 = \{x_2 \mapsto 1, v_0 \mapsto y_2\}$$

$$\langle \sigma_1, \Box \rangle \qquad\qquad \langle \sigma_2, \leftarrow \underline{h'(1) \approx z_2}, h(z_2) \approx y_2 \rangle$$

$$\sigma_3 = \{x_3 \mapsto 1, z_2 \mapsto y_3\}$$

$$\langle \sigma_3 \circ \sigma_2, \leftarrow \underline{h'([1]) \approx y_3}, h(y_3) \approx y_2 \rangle$$

$$\vdots$$

We could enhance the matching to a limited form of unification where we only test for unifiability — yes or no — without seeking the unifiers in the yes case. This would enable to select atoms whose lhs are not ground and fail them if they are not unifiable. If, however, they are unifiable, then the current atom is deselected and some other atom is selected to proceed with the derivation.

Such a scheme would, however, not completely eliminate incompleteness. Also, such forms of incompleteness are not expected to be common in practise. Hence, we find such incompleteness to be acceptable.

9 Subset-Relational Language

In this chapter we treat the next level of the subset-logic languages, involving equational, subset and relational assertions, but excluding their negations. It is called the **subset-relational language** and programming in this paradigm is called **subset-relational programming**.

We give the syntax of these languages and continue with the Clark completion style for the declarative semantics. On the declarative side, the new factor is the presence of infinite sets and we discuss its implications. However, it is not clear how important this is. On the operational side, the heart of the resolution procedure now becomes unification and disunification. As with the previous chapter, finite failure is a necessity for collecting sets over resolution trees. We also discuss the soundness and completeness issues. Our description below is only in the form of a sketch of the semantics since the ideas have not been formalised, doing which could be involved and extended.

9.1 Syntax

Here we give only those definitions of the syntactic components of the subset-relational language that need changes from earlier chapters. Let Σ be an alphabet with $\Sigma_{\overline{P}} \neq \emptyset$, i.e., relation symbols are now included. As before, we assume Σ_F is partitioned into Σ_{nU}, Σ_A and Σ_{nA}. The following definitions are based on Σ.

The definitions have sought to strike a balance between simplicity of statement and generality of purpose. They can be tuned to be more general or more restricted, but their essential treatment would not, however, differ from the development in this chapter.

Definition 9.1.1 An **equational clause** is a clause of the form

$$f(\vec{t}) \approx e \leftarrow L_1, \ldots, L_n$$

where $n \geq 0$, $Var(e) \subseteq Var(\vec{t})$, and for $1 \leq i \leq n$, $Var(L_i) \subseteq Var(\vec{t})$, and $L_i \equiv q_i(\vec{e_i})$ with $q_i \in \Sigma_P$ or $L_i \equiv \neg q_i(\vec{e_i})$ with q_i a set predicate. Also *scons* does not appear in $\vec{t}$ and *dscons* does not appear in e (though *scons* may appear in e and *dscons* may appear in $\vec{t}$).

Note that L_i is either a positive literal whose predicate $q_i \in \Sigma_{\overline{P}}$ or a (positive or negative) literal involving a set predicate.

Definition 9.1.2 A **subset clause** is a clause in one of two forms:

(i) If $f \in \Sigma_{nA}$, then it is as before, i.e., it is of the form $f(\vec{t}) \underset{\sim}{\supset} e \leftarrow$, for which $Var(e) \subseteq Var(\vec{t})$ and *scons* does not appear in $\vec{t}$ and *dscons* does not appear in e.

(ii) If $f \in \Sigma_A$, then it is of the form

$$f(\vec{t}) \underset{\sim}{\supset} e \leftarrow L_1, \ldots, L_n$$

where $n \geq 0$, $Var(e) \subseteq Var(\vec{t}, L_1, \ldots, L_n)$, and $L_i \equiv q_i(\vec{e_i})$ with $q_i \in \Sigma_P$ or $L_i \equiv \neg q_i(\vec{e_i})$ with q_i a set predicate. Also *scons* does not appear in $\vec{t}$ and *dscons* does not appear in e (though *scons* may appear in e and *dscons* may appear in $\vec{t}$).

We do not allow a non-empty clause body for subset clauses in part (i) because relations and non-aggregating functions do not seem to go together. For example, if $f \in \Sigma_{nA}$ and $f(x) \underset{\sim}{\supset} \{g(x)/y\} \leftarrow p(x, y)$ were an allowed clause, then the collect-all for f would require, through the *cond_welldef* condition, that $g(x)$ be defined. However, it does not make sense to insist that $g(x)$ be defined when the body $p(x, y)$ may possibly not hold for any argument values of x, y so that the head cannot be asserted, thereby making the definedness of $g(x)$ unnecessary. That is, while it is natural to require a function not to be undefined, it is entirely natural for a predicate to hold at 0 or more values of its arguments and these two seem to be at odds.

Definition 9.1.3 A **relational clause** is a clause of the form

$$p(\vec{t}) \leftarrow L_1, \ldots, L_n$$

where $n \geq 0$, $p \in \Sigma_{\overline{P}}$, and $L_i \equiv q_i(\vec{e_i})$ with $q_i \in \Sigma_P$ or $L_i \equiv \neg q_i(\vec{e_i})$ with q_i a set predicate.

Definition 9.1.4 A set of program clauses P is **non-overlapping** if for every equational clause in P with head atom $f(\vec{t}) \approx e$ and any other equational or subset clause of f in P with head atom $f(\vec{t'}) \approx e'$ or head atom $f(\vec{t'}) \gtrsim e'$ having a syntactically different argument $f(\vec{t'})$, we have that $\vec{t}$ does not unify with (a variant of) $\vec{t'}$.

Such a definition permits two or more clauses for f with syntactically identical arguments $f(\vec{t})$, such as the clauses $f(\vec{t'}) \approx e_1 \leftarrow B_1$ and $f(\vec{t'}) \approx e_2 \leftarrow B_2$, to be non-overlapping. This would allow definition by cases.

Definition 9.1.5 A **program** is a finite set of program clauses that is non-overlapping.

Some examples of subset-relational programs are as below. In program $\mathtt{P}_1$, `permute_set` intends to convert a set of n elements into a list of some permutation of the elements, with no duplications, while `perms(x)` is the set of all permutations of the elements of a set `x`. In program $\mathtt{P}_2$, `setdiff` denotes set difference and `disjoint` denotes disjointness of its argument sets. Assume that `perms` and `setdiff` are aggregating.

```
P1:  permute_set(∅, [])
     permute_set({x\y}, [x|z]) ← permute_set(y, z)
     perms(x) ⊇~ {y} ← permute_set(x, y)

P2:  setdiff(x, y) ⊇~ {z} ← z ∈ x, z ∉ y
     disjoint(x, y) ← setdiff(x, y) = x
```

Definition 9.1.6 A **query** is of the form $L_1, \ldots, L_n$ and a **goal clause** is of the form $\leftarrow L_1, \ldots, L_n$, where $L_i \equiv q_i(\vec{e_i})$ with $q_i \in \Sigma_P$ or $L_i \equiv \neg q_i(\vec{e_i})$ with q_i a set predicate or $L_i \equiv e_i \approx y_i$ with initial symbol of e_i being a function symbol and y_i not occurring elsewhere in the query or goal.

Let $\mathbf{L}_1$ be the subset-relational language based on Σ, i.e. the set of all subset-relational programs. Let $\mathbf{Q}_1$ and $\mathbf{G}_1$ be, respectively, the set of all subset-relational queries and goals. Clearly, all these sets are recursive.

There are two other criteria, that are very likely undecidable, that one would like to impose on the syntax of programs. One is, as before, that programs be equationally consistent, but now there is an additional way by which inconsistency might occur. This might arise from the multiple equational clauses for a function with the same lhs arguments, such as in a definition by cases. The other is that when functions are involved in the bodies of clauses, one would like some syntactic condition that ensures their arguments are ground when they are evaluated during execution.

9.2 Completion

For a program P, we assume the clauses of P have undergone the flattening and disjointness transforms of §6.5 represented by F and D respectively. Thus, an equational clause like $f(\vec{t}) \approx e \leftarrow L_1, \ldots, L_n$ becomes first $f(\vec{t}) \approx s \leftarrow F$ and then $f(\tilde{\vec{t}}) \approx s \leftarrow F, D$. Similarly, a subset clause like $f(\vec{t}) \gtrsim e \leftarrow L_1, \ldots, L_n$ becomes $f(\tilde{\vec{t}}) \gtrsim s \leftarrow F, D$. Also, a relational clause like $p(\vec{t}) \leftarrow L_1, \ldots, L_n$ becomes first $p(\vec{t}) \leftarrow F$ and then $p(\tilde{\vec{t}}) \leftarrow F, D$.

Let, as before, P^{FD} represent this flattened and disjointed transformation of P. The subsequent manipulations applied to clauses of P^{FD} to obtain the collect-all set s, *cond_finset*, *cond_args*, and *cond_welldef* are the same as in §7.2, with slight differences which we explicate below.

Let function f be collect-all definable and suppose that f is defined through m equational clauses and n subset clauses, $m \geq 0$, $n > 1$, where the j-th equational clause $(1 \leq j \leq m)$ and the i-th subset clause $(1 \leq i \leq n)$ have the forms:

$f(\tilde{\vec{t'_j}}) \approx s'_j \leftarrow F'_j, D'_j$

$f(\tilde{\vec{t_i}}) \gtrsim s_i \leftarrow F_i, D_i$

Let $\vec{x'_j}$ be $Var(\tilde{\vec{t'_j}})$ and $\vec{x_i}$ be $Var(\tilde{\vec{t_i}})$. Let $\vec{y_i}$ be the rest of the variables in the i-th subset clause.

Then we have the i-th clause to be logically equivalent to:

$$f(\vec{v}) \gtrsim w \leftarrow \exists \vec{x_i}, \vec{y_i}(\vec{v} = \tilde{\vec{t_i}} \wedge w = s_i \wedge F_i \wedge D_i)$$

where $\vec{v}$ and w are new variables to all the n subset clauses about f in P^{FD}. The collect-all assumption gives rise to the following clause for f when $f \in \Sigma_A$.

$$f(\vec{v}) \approx s \leftarrow \mathit{cond\_finset} \wedge \mathit{cond\_args}$$

where s, *cond_finset*, and *cond_args* are as follows.

$$s \equiv \cup_{i=1}^{n} \bigcup\{w \mid \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}$$

$$= \cup_{i=1}^{n} \bigcup\{s_i \mid \exists \vec{z_i}(\vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)\}, \quad \text{where } \vec{z_i} = (\vec{x_i}, \vec{y_i}) \backslash Var(s_i).$$

$$\mathit{cond\_finset} \equiv \bigwedge_{i=1}^{n} \exists u(set(u) \wedge \forall w(w \in u \leftrightarrow \exists \vec{x_i}, \vec{y_i}(w = s_i \wedge \vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)))$$

$$\mathit{cond\_args} \equiv \neg\exists \vec{x'_1}(nonmem(\vec{t'_1}) \wedge \vec{v} = \tilde{\vec{t'_1}}) \wedge \cdots \wedge \neg\exists \vec{x'_m}(nonmem(\vec{t'_m}) \wedge \vec{v} = \tilde{\vec{t'_m}})$$

When $f \in \Sigma_{nA}$, we have the following collect-all clause for f

$$f(\vec{v}) \approx s \leftarrow \mathit{cond\_finset} \wedge \mathit{cond\_args} \wedge \mathit{cond\_welldef}$$

where s, *cond_finset* and *cond_args* are as above and *cond_welldef* is as follows. Note that below D_i is just $nonmem(\vec{t_i})$. Also, as before, in all the four objects s, *cond_finset*, *cond_args* and *cond_welldef*, the free variables are exactly $\vec{v}$.

$$\mathit{cond\_welldef} \equiv \bigwedge_{i=1}^{n} \forall \vec{x_i}(D_i \wedge \vec{v} = \tilde{\vec{t_i}} \rightarrow \exists \vec{y_i} F_i)$$

The IFF-completions of various clauses are as in §8.2. We repeat the relevant parts below.

We first look at the completions for equational clauses. Let f be any function in Σ_F and suppose that f is defined through m equational clauses, $m \geq 1$, with the same notation as above. Then, each j-th equational clause of f

$$f(\tilde{\vec{t'_j}}) \approx s'_j \leftarrow F'_j, D'_j$$

translates to

$$f(\tilde{\vec{t'_j}}) \approx w' \leftrightarrow \exists \vec{y'_j}(w' = s'_j \wedge F'_j \wedge D'_j)$$

where $\vec{y'_j}$ are the rest of the variables of the j-th clause and w' is a variable new to the clause.

If there is more than one (say k many) equational clauses with the identical lhs argument $f(\tilde{\vec{t'_j}})$, then this group of clauses translates to the single IFF-completion

$$f(\tilde{\vec{t'_j}}) \approx w' \leftrightarrow (B_1 \vee \cdots \vee B_k)$$

where w' is a variable new to all the k clauses, and the disjuncts B are the bodies of clauses as above. For example, $B_1 \equiv \exists \vec{y'_j}(w' = s'_j \wedge F'_j \wedge D'_j)$.

Also, the following additional clause is added for all those $f \in \Sigma_{nU}$, to cover the definition of f for all argument values $\vec{v}$ not covered by the equational clauses of f.

$$f(\vec{v}) \not\approx w' \leftarrow \mathit{cond\_args}.$$

Next, we look at the completions for the collect-all clauses. For $f \in \Sigma_{nA}$, the subset clauses for f lead to

$$(f(\vec{v}) \approx w' \leftrightarrow w' = s \wedge \mathit{cond\_finset} \wedge \mathit{cond\_welldef}) \leftarrow \mathit{cond\_args}.$$

For $f \in \Sigma_A$, the subset clauses for f lead to

$$(f(\vec{v}) \approx w' \leftrightarrow w' = s \wedge \mathit{cond\_finset}) \leftarrow \mathit{cond\_args}.$$

Lastly, we look at the completions of relational clauses. For predicates $p \in \Sigma_{\overline{P}}$ with $n \geq 0$ relational clauses, we have their IFF-completions the same way as in logic programming. Suppose that p is defined through n relational clauses, $n \geq 0$, where the i-th relational clause $(1 \leq i \leq n)$ has the form:

$$p(\tilde{\vec{t_i}}) \leftarrow F_i, D_i$$

Let $\vec{x_i}$ be $Var(\tilde{\vec{t_i}})$. and let $\vec{y_i}$ be the rest of the variables in the i-th clause. Then we have the i-th clause to be logically equivalent to:

$$p(\vec{v}) \leftarrow \exists \vec{x_i}, \vec{y_i}(\vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)$$

where $\vec{v}$ are new variables to all the n relational clauses. Next, the IFF-completion gives rise to the following equivalence for p.

$$p(\vec{v}) \leftrightarrow (B_1 \vee \cdots \vee B_n)$$

where $B_i \equiv \exists \vec{x_i}, \vec{y_i}(\vec{v} = \tilde{\vec{t_i}} \wedge F_i \wedge D_i)$.

For a subset-equational program P, let P^{FD}_{IFF} be P^{FD} after replacing all its equational and relational clauses with their IFF-completions. Let $CAA_{IFF}(P)$ be the IFF-completions arising from the subset clauses.

Definition 9.2.1 The **IFF-completion** of a program P, called $comp_{IFF}(P)$, is given by

$$comp_{IFF}(P) = P^{FD}_{IFF} \cup CAA_{IFF}(P) \cup SetAx \cup FunAx$$

Queries and goals undergo just the the flattening and disjointness transformation.

For the program $\mathtt{P}_1$ and $\mathtt{P}_2$, we have their IFF-completions as follows.

P_1^F: $\quad P_1^F = P_1$

P_1^{FD}: $\quad permute\_set(\emptyset, [\,])$

$permute\_set(\{x/y\}, [x|z]) \leftarrow permute\_set(y, z), x \notin y$

$perms(x) \supsetapprox \{y\} \leftarrow permute\_set(x, y)$

$comp_{IFF}(P_1)$:

$$permute\_set(v_1, v_2) \leftrightarrow (v_1 = \emptyset \wedge v_2 = [\,]$$
$$\vee \exists x, y, z(v_1 = \{x/y\} \wedge v_2 = [x|z] \wedge permute\_set(y, z) \wedge x \notin y))$$

$$perms(x) \approx w \leftrightarrow (w = \bigcup\{\{y\} \mid permute\_set(x, y)\} \wedge cond\_finset)$$

P_2^F: $\quad setdiff(x, y) \supsetapprox \{z\} \leftarrow z \in x, z \notin y$

$disjoint(x, y) \leftarrow setdiff(x, y) \approx z, z = x$

P_2^{FD}: $\quad P_2^{FD} = P_2^F$

$comp_{IFF}(P_2)$:

$$setdiff(x, y) \approx w \leftrightarrow (w = \bigcup\{\{z\} \mid z \in x \wedge z \notin y\} \wedge cond\_finset)$$

$$disjoint(x, y) \leftrightarrow \exists z(setdiff(x, y) \approx z \wedge z = x)$$

9.3 Declarative Semantics

We define the declarative semantics as before in terms of logical consequence from the completion $comp_{IFF}(P)$.

Definition 9.3.1 The **declarative semantics** $\mathcal{D}_P$ of P is given by:

$$\mathcal{D}_P = \{A \mid comp_{IFF}(P) \models A, A \text{ a ground program atom}$$
$$\text{(of the form } f(\vec{t}) \approx t' \text{ or } f(\vec{t}) \gtrsim t' \text{ or } p(\vec{t}))\}$$

Again, our interest in atoms of the form $f(\vec{t}) \gtrsim t'$ is technical.

Clearly, if P is a subset-equational program, then its new semantics under the above definition is the same as the old enhanced semantics under Defn. 8.3.1. This is because $comp_{IFF}(P)$ is the same logical formula in both cases.

We now discuss some characteristics of subset-relational programs.

When relations are involved, deducing negative logical consequences are a must, unlike the case with subset-equational programs where it was an option. For example, consider P and $comp_{IFF}(P)$ below.

P: $\quad f(x) \gtrsim y \leftarrow p(x, y)$

$\quad p(x, \{1\})$

$\quad p(1, \{2\})$

$comp_{IFF}(P)$:

$\quad f(x) \gtrsim y \leftarrow p(x, y)$

$\quad f(x) \approx w \leftrightarrow w = \bigcup\{y \mid p(x, y)\} \wedge cond\_finset$

$\quad p(x, y) \leftrightarrow (y = \{1\} \vee x = 1 \wedge y = \{2\})$

From P, we see that $f(1) \gtrsim \{1\}$ and $f(1) \gtrsim \{2\}$ but we cannot deduce $f(1) \approx \{1, 2\}$ as a logical consequence since there are models of P in which $p(1, y)$ holds at argument values other than $y = \{1\}$ and $y = \{2\}$. However, $comp_{IFF}(P)$ does not permit such models and it is easy to see that

$\neg p(1,y) \leftrightarrow y \neq \{1\} \wedge y \neq \{2\}$. Hence *cond_finset* holds and the collect-all gives $f(1) \approx \{1,2\}$.

An issue, new when relations are considered, is that the condition *cond_finset* can easily not hold, i.e., relations can easily give rise to infinite sets. For this reason, we cannot omit *cond_finset* in specifying completions of programs. For example, consider P and $comp_{IFF}(P)$ below.

P: $\quad f(x) \supsetsim y \leftarrow p(x,y)$

$\quad\quad p(1,y)$

$comp_{IFF}(P)$:

$\quad f(x) \supsetsim y \leftarrow p(x,y)$

$\quad f(x) \approx w \leftrightarrow w = \bigcup\{y \mid p(x,y)\} \wedge cond\_finset$

$\quad p(x,y) \leftrightarrow x = 1$

Note that the free variables of *cond_finset* is just x and so we may write *cond_finset*(x). In evaluating $f(1)$, we have infinitely many valuations of y such that $f(1) \supsetsim y$ and the collect-all gives rise to an infinite set. The condition *cond_finset*(1) no longer holds and we cannot deduce $f(1) \approx w$ for any w. We can in fact show more, namely, that $\neg$*cond_finset*(1) holds in $comp_{IFF}(P)$ and hence $\forall w(f(1) \not\approx w)$ too. In this case,we have

$$cond\_finset(1) \equiv \exists u(set(u) \wedge \forall y(y \in u \leftrightarrow p(1,y)))$$

which is equivalent to $\exists u(set(u) \wedge \forall y(y \in u))$ since $p(1,y) \leftrightarrow true$. In other words, if *cond_finset* holds, then there is a universal set that contains all elements. But in ZF set theory, one can use this to show a contradiction via the use of the *axiom of separation* and the idea of *Russell's paradox* (see [Sup72] §1.3, §2.3 Thm. 41). Hence we have $\neg$*cond_finset*(1).

We note the following technicality that arises from using an untyped system. It is not always the case that infinitely many valuations of a variable involved in a collect-all gives rise to an infinite set. For example, the following program P gives rise to a finite set in evaluating $f(1)$, namely, $f(1) \approx \emptyset$, since individuals rather than sets are involved in the collect-all. One would not expect such situations in normal programming though.

P: $f(x) \gtrsim y \leftarrow p(x, y)$

$p(1, [y])$

Lastly, the definition of a correct answer has to be more general when relations are considered. Since non-ground terms may be involved in relational queries, a correct answer has to be a solved form rather than a substitution. The exact nature of the solved form is described in the next section, but we give the definition of a correct answer below.

Definition 9.3.2 Let $P \in \mathbf{L}_1$ and $G \in \mathbf{G}_1$ with Q its corresponding query. Let ζ be a solved form with $Dom(\zeta) = Var(G^{FD})$. Then ζ is a **correct answer** for $P \cup \{G\}$ if $comp_{IFF}(P) \models \forall(\zeta \rightarrow Q^{FD})$.

9.4 Unification and Disunification

As in logic programming, resolving with relations involves unification rather than matching, so too do we need unification at the heart of the resolution procedure in subset-relational programming. Further, since dscons terms can be involved, which implicitly represents non-membership constraints, we have to go beyond unification to disunification. In the literature (see [Com91]), the term disunification is used to include solving both positive and negative constraints. Also, as the program $\mathtt{P}_2$ in §9.1 suggests, it is natural to have positive and negative set constraints in bodies of programs.

Hence, we consider the disunification problem in this section and the process of turning them into solved forms. A disunification algorithm for finite sets together with a termination proof has already appeared in [DR93, DOPR93], and our description below is based on it, although in our context. By this we mean that our description is influenced by the fact that we have an untyped system and the *dscons* constructor while the algorithm in [DR93] uses a typed system for their set constructor and do not have a *dscons*.

Rather than list all the rewrite rules, we discuss only the main ideas behind them together with their representative rules. We first look at solving equations involving dscons terms, since they are the kinds of constraints that arise in the resolving of a goal with the head of a clause.

Consider a goal $\leftarrow p(\{1/x\}, 2)$ to be resolved with a clause $p(\{2\backslash z\}, [y]) \leftarrow q(z, y)$. The clause, after undergoing a disjointness transformation, can be put in the equivalent form

$$p(x_1, x_2) \leftarrow x_1 = \{2/z\} \wedge x_2 = [y] \wedge q(z, y) \wedge 2 \notin z.$$

Hence, we have the derivation

$$\begin{array}{l} \leftarrow p(\{1/x\}, 2) \\ \quad \Big| \\ \leftarrow \{1/x\} = \{2/z\}, 2 = [y], q(z, y), 2 \notin z \end{array}$$

Thus, we see that in the context of resolution a dscons term can be viewed as a notation for the corresponding scons term together with its *nonmem* conjunct. This also happens when dscons terms appear in the bodies of clauses.

Rather than solving a constraint containing a dscons term through separately solving the corresponding scons constraint and the *nonmem* conjunct, one could give rewrite rules directly for equations involving dscons terms as follows.

Consider Prop. 4.3.1(i), which we recall below.

$$\begin{array}{l} set(x) \wedge set(y) \rightarrow \big(\{x_1/x\} = \{y_1/y\} \leftrightarrow \\ \quad \big((x_1 \notin x \wedge y_1 \notin y \wedge x_1 = y_1 \wedge x = y) \\ \quad \vee(x_1 \notin x \wedge y_1 \notin y \wedge x_1 \neq y_1 \wedge \exists z(set(z) \wedge x = \{y_1/z\} \wedge y_1 \notin z \wedge y = \{x_1/z\} \wedge x_1 \notin z)) \\ \quad \vee(x_1 \notin x \wedge y_1 \in y \wedge y = \{x_1/x\}) \\ \quad \vee(x_1 \in x \wedge y_1 \notin y \wedge x = \{y_1/y\}) \\ \quad \vee(x_1 \in x \wedge y_1 \in y \wedge x = y)\big)\big) \end{array}$$

By considering $x_1 \notin x$ on both sides of the equivalence and dropping redundant conjuncts in the rhs we obtain an equivalence which forms the basis of the following non-deterministic rules.

$$K \cup \{\{s_1 \backslash s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{s_1 \notin s_2, s_1 = t_1, s_2 = t_2\}$$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

$$K \cup \{\{s_1 \backslash s_2\} = \{t_1/t_2\}\} \Rightarrow \exists z K \cup \{s_1 \neq t_1, \{s_1 \backslash z\} = t_2, s_2 = \{t_1 \backslash z\}, set(z)\}$$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

Here z is a new variable.

$$K \cup \{\{s_1 \backslash s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{s_1 = t_1, \{s_1 \backslash s_2\} = t_2, set(s_2)\}$$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

$$K \cup \{\{s_1 \backslash s_2\} = \{t_1/t_2\}\} \Rightarrow K \cup \{t_1 \in s_2, \{s_1 \backslash s_2\} = t_2\}$$

if it is not the case that $last(s_2) \equiv last(t_2) \equiv$ a variable.

Similarly, one can give rules for rewriting other cases of atoms involving dscons terms in $=$, *set* and $\in$ relations.

The rules for solving negative constraints are discussed below. But first, we frame the disunification problem and their solved forms. Unlike unification, in the case of disunification there is no one solved form that is suitable for all situations. Hence, one chooses solved forms according to the requirements of one's context so long as they satisfy the criteria of *solvability*, *simplicity* and *completeness*. Solvability means that any solved form has at least one solution. Simplicity means every solution can be easily obtained from a solved form. Completeness means that every disunification problem is equivalent to a finite disjunction of solved forms.

For example, one might expect disunification solved forms to involve universal quantifiers since unification solved forms involve existential quantifiers. However, universal quantifiers are unsuitable in the context of resolution and [DR93] gives a solved form that can do without it. We use this form below.

A *disunification problem* χ is a conjunction of literals of the form $\exists \vec{z}(L_1 \wedge \cdots \wedge L_n)$ where

$n \geq 0$ and each L_i is of the form $set(s)$, $s = t$, $s \in t$ or their negations. Also, the terms can involve the *dscons* constructor.

As before, the free variables of χ are denoted by $Var(\chi)$, and we will abbreviate a disunification problem as an existential multiset $K = \exists \vec{z}\{A_i\}_{i=1}^{n}$ and let $Var(K)$ be its free variables.

A *solved form* is a disunification problem of the form

$$\exists \vec{z}\{set(y_1), \ldots, set(y_m), indiv(y'_1), \ldots, indiv(y'_{m'}),$$
$$u_1 \neq s_1, \ldots, u_k \neq s_k, s'_1 \notin u'_1, \ldots, s'_{k'} \notin u'_{k'},$$
$$x_1 = t_1, \ldots, x_n = t_n\}$$

where $m, k, n \geq 0$, and the x_i's are distinct free variables that occur exactly once. Also, each existential variable occurs at least once in the multiset, and if any y_j is an existential variable, then it occurs among the $=$ and $\in$ literals. Further, u_i does not occur in s_i and u'_i does not occur in s'_i.

In the above definition, one would like to allow the *dscons* constructor in the terms. However, the solvability requirement is easily violated, e.g., in $\{set(y), x = \{1\backslash\{1\backslash y\}\}\}$. What additional criteria are required to ensure the solvability condition when dscons terms are allowed in solved forms is an issue that has to be addressed and is not treated here.

The equations of a solved form correspond to a substitution. It is convenient to apply a solved from ζ to an object in the manner of a substitution. By this we mean the substitution corresponding to ζ. Also, let ζ^- denote ζ without its equations.

We will continue to use the word unifier in the sense of a solved form representing some of the solutions of a disunification problem.

Turning now to the rules, we use the same strategy as in the past, namely, find suitable logical equivalences for their basis. One source of such equivalences are the equivalences $A \leftrightarrow B$ used for the unification case. We now use $\neg A \leftrightarrow \neg B$ instead. (Let us call $\neg A \leftrightarrow \neg B$ as the converse equivalence of $A \leftrightarrow B$.) Representative examples of such rules are as follows.

$K \cup \{indiv(\emptyset)\} \Rightarrow F$

$\exists z K \cup \{indiv(z)\} \Rightarrow K$

if $indiv(z)$ does not occur in K and $\Sigma_{\overline{C}} \neq \emptyset$.

$K \cup \{c \neq c\} \Rightarrow F$

if c is a 0-ary constructor.

$K \cup \{c(s_1, \ldots, s_m) \neq c(t_1, \ldots, t_m)\} \Rightarrow K \cup \{s_i \neq t_i\}$

if $m \geq 1$ and $1 \leq i \leq m$.

$K \cup \{s \notin \emptyset\} \Rightarrow K$

$K \cup \{s \notin \{t_1/t_2\}\} \Rightarrow K \cup \{indiv(t_2)\}$

$K \cup \{s \notin \{t_1/t_2\}\} \Rightarrow K \cup \{set(t_2), s \neq t_1, s \notin t_2\}$

We do not need rules for atoms of the form $x \neq t$ and $s \notin x$, where x does not occur in s, t, since these are already in solved form. Next is a rule that does not use the idea of a converse equivalence, for to do so would result in a universal variable. Instead, a simple venn-diagram visualization of the the two terms as sets is enough.

$K \cup \{x \neq t\} \Rightarrow K \cup \{indiv(x)\}$

$K \cup \{x \neq t\} \Rightarrow K \cup \{set(x), t_i \notin x\}$

if in both rules $x \in Var(t)$, $x \not\equiv t$,

$t \equiv \{t_1, \ldots, t_n/x\}$ with $n \geq 1$, and $x \notin Var(t_1, \ldots, t_n)$.

In the second rule $1 \leq i \leq n$.

Finally, we have the rules below for atoms for which it is again unsuitable to use the converse equivalence idea. This is because the B in $A \leftrightarrow B$ is a disjunction of conjunctions, so that the converse equivalence leads to a conjunction of disjunctions. So we reason differently, starting from basic ideas about sets, to obtain an equivalence having a disjunction of conjunctions. For example, we have $Set1 \neq Set2$ iff $Set1 \backslash Set2 \neq \emptyset \vee Set2 \backslash Set1 \neq \emptyset$.

$$K \cup \{\{s_1/s_2\} \neq \{t_1/t_2\}\} \Rightarrow \exists z K \cup \{z \in \{s_1/s_2\}, z \notin \{t_1/t_2\}, set(s_2), set(t_2)\}$$

Here z is a new variable.

$$K \cup \{\{s_1/s_2\} \neq \{t_1/t_2\}\} \Rightarrow \exists z K \cup \{z \in \{t_1/t_2\}, z \notin \{s_1/s_2\}, set(s_2), set(t_2)\}$$

Here z is a new variable.

We have omitted the rules for the cases of $indiv(s_2)$ or $indiv(t_2)$.

The above completes the description of a disunification algorithm. However, one could always seek more efficient algorithms since it is the heart of the resolution procedure. Note that the set unification problem has been shown to be NP-complete ([KN86, DOPR93]).

9.5 Operational Semantics

We give a few examples to convey the computation procedure in subset-relational programming. The derivations are now a combination of the tree structure usual in logic programming for relations and the vine-like structures of the previous chapter for functions. We continue to call the computation procedure as SLDUFF resolution.

As before, we compute with just the flattened forms of programs and goals. A successful computation is now obtained when a leaf node in a derivation tree denotes a solved form, of which a special case is the empty goal $\square$. Also, when function subgoals such as $f(\vec{t}) \approx x$ are to be resolved then matching of §7.4 is to be used and otherwise disunification of §9.4. Additionally, matching can be used for relations on those arguments for which the mode information knows the argument should be ground.

Consider the computation of $P_1 \cup \{\leftarrow perms(\{1,2\}) \approx x\}$. Due to lack of space, we show only one successful branch of the tree.

$$\leftarrow \underline{perms(\{1,2\}) \supseteq z_0}$$

$$\sigma_1 = \{x_1 \mapsto \{1,2\}, z_0 \mapsto \{y_1\}\}$$

$$\langle \sigma_1, \leftarrow \underline{permute\_set(\{1,2\}, y_1)} \rangle$$

$$\sigma_2 = \{x_2 \mapsto 1, y_2 \mapsto \{2\}, y_1 \mapsto [1|z_2]\} \qquad \sigma_2 = \{x_2 \mapsto 2, y_2 \mapsto \{1\}, y_1 \mapsto [2|z_2]\}$$

$$\langle \sigma_2 \circ \sigma_1, \leftarrow \underline{permute\_set(\{2\}, z_2)} \rangle \qquad \langle \sigma_2 \circ \sigma_1, \leftarrow \underline{permute\_set(\{1\}, z_2)} \rangle$$

$$\sigma_3 = \{x_3 \mapsto 1, y_3 \mapsto \emptyset, z_2 \mapsto [1|z_3]\}$$

$$\cdots \qquad \langle \sigma_3 \circ \sigma_2 \circ \sigma_1, \leftarrow \underline{permute\_set(\emptyset, z_3)} \rangle \qquad \cdots$$

$$\sigma_4 = \{z_3 \mapsto [\,]\}$$

$$\langle \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1, \square \rangle$$

We have $\sigma_4 \circ \sigma_3 \circ \sigma_2(z_0) = \{[2|[1|[\,]]]\} = \{[2,1]\}$

It is clear that the set collected indeed the set of permutations of the elements of $\{1,2\}$. Next, consider the computation of $P_1 \cup \{\leftarrow disjoint(\{1\},\{2\})\}$. It has the following derivation.

$$\langle \epsilon, \leftarrow \underline{disjoint(\{1\},\{2\})} \rangle$$

$$\sigma_1 = \{x_1 \mapsto \{1\}, y_1 \mapsto \{2\}\}$$

$$\langle \sigma_1, \leftarrow \underline{setdiff(\{1\},\{2\}) \approx z_1}, z_1 = \{1\} \rangle$$

$$\sigma_4 = \{z_1 \mapsto \{1\}\} \quad \text{(from tree below)}$$

$$\langle \sigma_4 \circ \sigma_1, \leftarrow \underline{\{1\} = \{1\}} \rangle$$

$$\zeta_5 = \epsilon \quad \text{(from constraint solving)}$$

$$\langle \zeta_5 \circ \sigma_4 \circ \sigma_1, \square \rangle$$

The set-collecting tree for $P \cup \{\leftarrow setdiff(\{1\},\{2\}) \approx z_1\}$ is as follows.

$$\leftarrow \underline{setdiff(\{1\},\{2\}) \supseteq x_0}$$

$$\sigma_2 = \{x_2 \mapsto \{1\}, y_2 \mapsto \{2\}, x_0 \mapsto \{z_2\}\}$$

$\langle \sigma_2, \leftarrow z_2 \in \{1\}, z_2 \notin \{2\} \rangle$

$\quad \mid \zeta_3 = \{z_2 = 1\} \quad$ (from constraint solving)

$\langle \zeta_3 \circ \sigma_2, \Box \rangle$

answer set $s = \bigcup \{\zeta_3 \circ \sigma_2(x_0)\} = \bigcup \{\{1\}\} = \{1\}$

In general, in resolving a relation, one obtains a solved form ζ as a unifier which need not be a susbtitution. Then, a typical derivation step looks as below where $p(\vec{t})$ has been resolved with a clause $p(\vec{s}) \leftarrow B_1$.

$\langle \theta, \leftarrow \underline{p(\vec{t})}, B \rangle$

$\quad \mid \zeta$ is a unifier of $\vec{t} = \vec{s}$

$\langle \zeta \circ \theta, \leftarrow \zeta^-, \zeta B_1, \zeta B \rangle$

We omit the existential variables from ζ^- since they would move to the front of the goal and become universal variables.

The derivation trees are naturally more complicated but the considerations are those for trees in logic programming and trees in the previous chapters. As for finite failure, there is one additional way by which it occurs, namely, when a set-collecting tree can collect a finite set, but its elements contain variables, thereby denoting an infinite set. This is shown below.

Consider the program used in §9.3 which we repeat below.

P: $\quad f(x) \supseteq y \leftarrow p(x, y)$

$\quad p(1, y)$

The computation for $P \cup \{\leftarrow f(1) \approx x\}$ leads to the following tree.

$\leftarrow \underline{f(1) \supseteq x_0}$

$\quad \mid \sigma_1 = \{x_1 \mapsto 1, x_0 \mapsto y_1\}$

$$\langle \sigma_1, \leftarrow \underline{p(1, y_1)} \rangle$$
$$\Big| \; \sigma_2 = \{y_1 \mapsto y_2\}$$
$$\langle \sigma_2 \circ \sigma_1, \Box \rangle$$

answer set $s = \bigcup\{\sigma_2 \circ \sigma_1(x_0)\} = \bigcup\{y_2\}$

Hence, the set collected is $\bigcup\{y_2\}$ and the tree must be considered finitely failed.

9.6 Soundness and Completeness

Soundness is usually a straightforward albeit extended mutual induction. It is largely a matter of stating the inductions properly and carrying them through. The one new aspect now is that the collected set in set-collecting trees may have variables which denote infinite sets and therefore finite failure for the set-collecting tree. For such cases, we need to deduce $\neg$*cond_finset* and a systematic means of doing so for the various cases that occur in finitely failed set-collecting trees has to be a part of the soundness proof.

For completeness, the usual means by which incompleteness occurs in the previous chapters occurs now too. These are the inability to use unification in resolving function symbols and the need to make a 'depth-first' compromise in place of 'breadth-first'. An additional way by which incompleteness can now occur is when a set-collecting tree diverges on account of an infinite set being collected, whereas declaratively we can deduce $\neg$*cond_finset* for the corresponding set abstraction. Hence, declaratively, we would have, say $\neg\exists x(f(\vec{t}) \approx x)$, for ground term $\vec{t}$, but operationally we would hang.

For example, consider program P below where f, g are aggregating and s denotes a unary constructor.

$P:$	$f(x) \gtrsim g(x)$	$P^F:$	$f(x) \gtrsim z \leftarrow g(x) \approx z$
	$g(x) \gtrsim \{y\} \leftarrow p(x, y)$		$g(x) \gtrsim \{y\} \leftarrow p(x, y)$

$p(x, 0)$	$p(x, 0)$
$p(x, s(y)) \leftarrow p(x, y)$	$p(x, s(y)) \leftarrow p(x, y)$

Then, a derivation for the goal $\leftarrow f(1) \approx x$ leads to the following trees.

$$\leftarrow \underline{f(1) \underset{\sim}{\supset} x_0}$$

$$\Big| \quad \sigma_1 = \{x_1 \mapsto 1, x_0 \mapsto z_1\}$$

$$\langle \sigma_1, \leftarrow \underline{g(1) \approx z_1} \rangle$$

Below is the set-collecting tree for $P \cup \{\leftarrow g(1) \approx z_1\}$ which diverges.

$$\leftarrow \underline{g(1) \underset{\sim}{\supset} z_0}$$

$$\Big| \quad \sigma_2 = \{x_2 \mapsto 1, z_0 \mapsto \{y_2\}\}$$

$$\langle \sigma_2, \leftarrow \underline{p(1, y_2)} \rangle$$

$$\sigma_3 = \{x_3 \mapsto 1, y_2 \mapsto 0\} \; / \qquad \backslash \; \sigma_4 = \{x_4 \mapsto 1, y_2 \mapsto s(y_4)\}$$

$$\langle \sigma_3 \circ \sigma_2, \Box \rangle \qquad \langle \sigma_4 \circ \sigma_2, \leftarrow \underline{p(1, y_4)} \rangle$$

$$\sigma_5 = \{x_5 \mapsto 1, y_4 \mapsto 0\} \; / \qquad \backslash$$

$$\langle \sigma_5 \circ \sigma_4 \circ \sigma_2, \Box \rangle \qquad \vdots$$

Hence, declaratively, we would have $f(1) \approx \emptyset$, but operationally we would diverge.

The above assumes that our set axioms are strong enough to deduce $\neg$*cond_finset* for those set abstractions denoting infinite sets. One needs to explore whether this is indeed the case, or for which infinite sets it is the case. Also, one has to estimate how important the role of negative logical consequence through infinite sets is to practical programming.

10 Conclusion and Further Work

In closing, we discuss the accomplishments and significance of the dissertation, after which we discuss further work beyond the dissertation.

10.1 Contributions of the Dissertation

This work set out to investigate the interaction of the subset assertions with equational and relational assertions by ascribing semantics to a graded sequence of programming languages called the subset-logic languages. We have succeeded in giving the syntax and semantics for the first level, namely, the subset-equational language, which examines the interaction of equational assertions and finite sets. This alone constituted a significant amount of work and comprises the chapters 3 to 8 of the dissertation. We have further outlined a syntax and semantics for the next level, namely, the subset-relational language in chapter 9. This language examines the interaction of subset assertions with equational and relational assertions, but excluding their negations. We consider them in the context of both positive and negative set constraints.

The significance of this work is as follows.

- It has provided a rigorous development for the use of set constructors in logic programming. Such a development is applicable to other logic languages proposing to incorporate finite sets.
- We have overcome the technical difficulties posed by the set constructors and the collect-all principle to give the subset-equational language the same elegant declarative semantics that definite clause programs possess in logic programming. We have also shown how the Clark completion can be adapted to subset-equational programs to provide answers in the

presence of incomplete information.

- Finally, we have outlined a logical-consequence semantics for the subset-relational language, again through the Clark completion, to give an initial treatment of the interaction of subset assertions with equations and relations.

More specifically, we note the following. We laid the logical foundations for set constructors through axioms and then used them to show that the set constructors indeed behave exactly like finite sets, to provide a framework for establishing the correctness of set-unification, and to define a Herbrand structure.

On a technical note, we found it appropriate to view unification in terms of logical consequence from the set axioms rather than as satisfaction in a particular algebra. Our terminology of unification (§4.2) is accordingly influenced by this viewpoint. Thus, minimality of a set of unifiers (Defn. 4.2.8) is based on the fact that the solutions expressed by any one unifier are not all expressed by the rest of the unifiers. This appears more natural than the usual notion of minimality based on absence of instances of unifiers (i.e., substitutions). It was not easy to see if the two definitions coincide except when the unifiers were essentially ground substitutions, in which case they did (Prop. 4.5.3).

We have improved on the semantics sketched for the subset-equational language in [JP89]. We have used the foundations of set constructors to fill in all the earlier gaps so that we could discuss logical-consequence semantics. We have clarified the form and meaning of the collect-all clause in the completions of programs through the conditions *cond_finset*, *cond_args* and *cond_welldef*. Thereby, we have also eliminated the need for the somewhat ad-hoc emptiness-as-failure priniciple.

In giving the fixpoint semantics (§7.6) we found that the usual conditions attendant on the T_P operator do not hold such as being a total operator on the powerset of the Herbrand Base. We found modifications of the standard theorems we feel are pleasing that permits the semantics to go through cleanly.

We have shown how the Clark completion semantics can be adapted to the subset-equational and subset-relational languages. In the case of the subset-equational language, it allows us to compute more queries than was possible through the semantics outlined in [JP89]. It does lead to incompleteness, however, but this seems to occur only on the uncommon cases of programs.

Lastly, we remark on two aspects of the semantics we have obtained. One is the rather complicated form of the semantics in terms of the rather large or sometimes formidable first-order formulae that we have to deal with. Some of this arises from formalising set issues and we take the same attitude that set theorists do with respect to axiomatic set theory, namely, that once one has become familiar with the machinery of formalisation, one can then reason and remain at an informal level.

The other aspect concerns our attitude towards logical-consequence semantics. Current trends in logical semantics pursues the model-theoretic approach in order to handle the drawbacks of the Clark completion semantics when negation is introduced in logic programming. Nevertheless, we feel it is important to know which parts of the subset-logic languages can be given a logical-consequence semantics, the logically impeccable approach. Besides, as Shepherdson remarks in [She92]: "The Clark completion of a program is one of the simplest declarative semantics which have been offered for negation as failure and is currently the most used".

10.2 Further Work

This work has been rather broad based, having to deal with many fronts such as set axioms, unification and the initial levels of the subset-logic languages. As such, it could not attend to all aspects of semantics such as finding syntactic restrictions that would lead to completeness or investigating model-theoretic semantics. However, our work provides a firm foundation from which further work on the subset-logic languages can proceed. We touch upon some of the avenues of further work below.

Firstly, the formalisation of the semantics sketched for the subset-relational language has to be carried out, based on the issues mentioned in chapter 9.

Next, giving a fixpoint semantics, both for the enhanced semantics of the subset-equational language and the subset-relational language would be desirable. This is likely to involve the construction at each iteration of both the success and failure sets instead of only the success set of before. Giving such a semantics would throw added light on the declarative semantics and would be necessary to show completeness for restricted versions of the languages.

Then there are the further levels of the subset-logic languages to be considered. At the next level is allowing negative relations and equations in the clause bodies. Here, one would need to evaluate the various model-theoretic approaches in the literature and apply it to our case.

Beyond this level are the language with choice operations and the language with infinite objects. In the former, we want to additionally consider atoms of the form $f(\vec{t}) \gtrsim t'$ in clause bodies, to non-deterministically choose some element of $f(\vec{t})$ or lazily generate all elements of $f(\vec{t})$ (see [Jay91]). In the latter we seek to obtain meaningful computation from infinite structures such as infinite sets and lists.

References

[AG91] Abiteboul, S. and Grumbach, S.: A Rule-Based Language with Functions and Sets, *ACM Trans. on Database Systems*, Vol.16, No. 1, Mar. 1991, pp. 1–30.

[Aït91] Aït-Kaci, H.: *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, 1991.

[BNST91] Beeri, C., Naqvi, S., Shmueli, O., and Tsur, S.: Set Constructors in a Logic Database Language, *Jnl. Logic Programming*, 1991:10:181–232.

[Bra90] Bratko, I.: *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1990.

[BRSS92] Beeri, C., Ramakrishnan, R., Srivastava, D., and Sudarshan, S.: The valid model semantics for logic programs, *Proc. of ACM Symp. on Princ. of Database Systems (PODS)*, Jun 1992, pp. 91–104.

[Cha88] Chan, D.: Constructive Negation Based on the Completed Database, *Proc. Fifth Int. Conf. on Logic Programming*, Seattle, Washington, Aug. 1988, pp. 111–125.

[Com91] Comon, H.: Disunification: A Survey, in: *Computational Logic: Essays in Honor of Alan Robinson*, MIT Press, 1991, pp. 322–359.

[DL86] DeGroot, D. and Lindstrom, G. (eds.): *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, New Jersey, 1986.

[DM79] Dershowitz, N. and Manna, Z.: Proving Termination with Multiset Orderings, *Commn. of the ACM*, Vol. 22, No. 8, Aug. 1979, pp. 465–476.

[DOPR91] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G.: {log}: A Logic Programming Language with Finite Sets, *Proc. Eight Int. Conf. on Logic Programming*, Paris, Jun. 1991, pp. 111–124.

[DOPR93] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G.: {log}: A Language for Programming in Logic with Finite Sets, submitted to *Jnl. of Logic Programming*, 1993.

[DR93] Dovier, A. and Rossi, G.-F.: Embedding extensional finite sets in CLP, *Proc. Fourth Int. Logic Programming Symposium*, MIT Press, 1993.

[End72] Enderton, H. B.: *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[Fit90] Fitting, M.: *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, New York, 1990.

[FBLD73] Fraenkel, A. A., Bar-Hillel, Y., Levy, A. and van Dalen, D.: *Foundations of Set Theory*, North-Holland, 1973.

[FLMP88] Falaschi, M., Levi, G., Martelli, M. and Palamidessi, C.: A New Declarative Semantics for Logic Languages, *Proc. Fifth Int. Conf. Logic Programming*, Seattle, Washington, Aug. 1988, pp. 993–1005.

[GL88] Gelfond, M. and Lifschitz, V.: The Stable Model Semantics for Logic Programming, *Proc. Fifth Int. Conf. on Logic Programming*, Seattle, Washington, Aug. 1988, pp. 1070–1080.

[GLMP91] Giovannetti, E., Levi, G., Moiso, C. and Palamidessi, C.: Kernel-LEAF: A Logic plus Functional Language, *Jnl. of Computer and System Sciences*, **42**, 139–185 (1991).

[GM87] Goguen, J. A. and Meseguer, J.: Models and Equality for Logical Programmming, *Proc. Int. Jnt. Conf. on Theory and Practice of Software Development* (TAPSOFT), Pisa, Italy, Mar. 1987, pp. 1–22.

[GS88] Gaifman, H. and Shapiro, E.: Fully Abstract Compositional Semantics for Logic Programs, *Proc. 16th ACM Symp. on Princ. of Programming Languages*, 1989, pp. 134–

142.

[HO82] Hoffmann, C. M. and O'Donnell, M. J.: Programming with Equations, *ACM Trans. Program. Lang. and Syst.*, Vol. 4, No.1, Jan. 1982, pp. 83–112.

[Höl89] Hölldobler, S.: *Foundations of Equational Logic Programming*, Lecture Notes in Artificial Intelligence 353, Springer-Verlag, Berlin 1989.

[Jay90] Jayaraman, B.: Towards a Broader Basis for Logic Programming, Tech. Report 90-08, Dept. of Computer Science, SUNY, Buffalo, 1990.

[Jay91] Jayaraman, B.: The SuRE Programming Framework, TR 91–11, SUNY at Buffalo, New York, Aug. 1991.

[JL87] Jaffar, J. and Lassez, J.-L.: Constraint Logic Programming, *Fourteenth Annual ACM Symp. on Princ. of Programming Languages*, Münich, Jan. 1987, pp. 111–119.

[JP89] Jayaraman, B. and Plaisted, D. A.: Programming with Equations, Subsets, and Relations, *Proc. North American Conf. Logic Programming*, Cleveland, Oct. 1989, pp. 1051–1068.

[JK91] Jouannaud, J.-P. and Kirchner, C.: Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification, in: *Computational Logic: Essays in Honor of Alan Robinson*, MIT Press, 1991, pp. 257–321.

[KN86] Kapur, D. and Narendran, P.: NP-Completeness of the Set Unification and Matching Problems, *Proc. of 8th Int. Conf. on Automated Deduction*, Oxford, England, Jul. 1986, Springer-Verlag, 1986.

[Kow79] Kowalski, R.: *Logic for Problem Solving*, North-Holland, Elsevier, New York, 1979.

[Kup90] Kuper, G. M.: Logic Programming with Sets, *Jnl. of Computer and System Sciences*, Vol. 41, No. 1, Aug. 1990, pp. 44–64.

[Llo87] Lloyd, J. W.: *Foundations of Logic Programming*, Springer-Verlag, 1987.

[Min88] Minker, J. (ed.): *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, California, 1988.

[MW85] Manna, Z. and Waldinger, R.: *The Logical Basis for Computer Programming: Vol. 1: Deductive Reasoning*, Addison-Wesley, 1985.

[Men79] Mendelson, E.: *Introduction to Mathematical Logic*, 2nd ed., Wadsworth, 1979.

[NT89] Naqvi, S. and Tsur, S.: *A Logical Language for Data and Knowledge Bases*, Computer Science Press, New York, 1989.

[ODon85] O'Donnell, M. J.: *Equational Logic as a Programming Language*, MIT Press, 1985.

[Prz88] Przymusinski, T. C.: Perfect Model Semantics, *Proc. Fifth Int. Conf. on Logic Programming*, Seattle, Washington, Aug. 1988, pp. 1081–1096.

[Prz89] Przymusinski, T. C.: On Constructive Negation in Logic Programming, *Proc. North Amer. Conf. on Logic Programming*, Cleveland, Oct. 1989.

[Rub73] Rubin, J. E.: Finite Sets, *Mathematics Magazine*, Vol. 46, Sep. 1973, No. 4, pp. 183–192.

[Ros89] Ross, K. A.: A Procedural Semantics for Well Founded Negation in Logic Programs, *Proc. of Eighth ACM Symp. on Princ. of Database Systems*, Philadelphia, Mar. 1989, pp. 22–33.

[Sch90] Schlipf, J.: Formalizing a Logic for Logic Programming, Tech. Report. No. CIS-TR-90-2, Computer Science Dept., Univ. of Cincinnati, Ohio, Feb. 1990.

[SDDS86] Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schonberg, E.: *Programming with Sets – An Introduction to SETL*, Springer-Verlag New York, 1986.

[She92] Shepherdson, J. C.: Logics for Negation as Failure, in: *Logic from Computer Science*, MSRI Publications 21, Springer-Verlag 1992.

[Sup57] Suppes, P.: *Introduction to Logic*, D. Van Nostrand Co., Princeton, New Jersey, 1957.

[Sup72] Suppes, P.: *Axiomatic Set Theory*, 1960, Dover edn., New York, 1972.

[TS86] Tamaki, H. and Sato, T.: OLD Resolution with Tabulation, *Proc. Third Int. Conf. on Logic Programming*, London, U. K., Jul. 1986, pp. 84–98.

[Tur89] Turi, D.: Logic Programs with Negation: Classes, Models, Interpreters, Report CS-R8943, Centre for Math. and Comp. Sc., Amsterdam, Oct. 1989.

[vGRS91] Van Gelder, A., Ross, K., and Schlipf, J. S.: The Well-Founded Semantics for General Logic Programs, *Jnl. of ACM*, Vol. 38, No. 3, Jul 1991, pp. 620–650.