Amalgamating Functional and Relational Programming through the Use of Equality Axioms

Keitaro Yukawa
Department of Computer Science

Research Report CS-87-61
October 1987

Amalgamating Functional and Relational Programming through the Use of Equality Axioms

by

Keitaro Yukawa

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1987

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The continuing need for a superior declarative programming language can be met to an important degree by amalgamating functional and relational programming, the two major methods of declarative programming. This frees the programmer of the awkwardness that may arise in either style alone, and opens a wider variety of programming paradigms in a single language. This dissertation presents a theoretical basis for the amalgamation of functional and relational programming within the framework of Horn clauses with equality, advancing the thesis that the amalgamation can be had by adding equality axioms to Horn clause relational programs.

We first discuss the semantics of logic programs with equality, namely a generalization of the model-theoretic and fixpoint semantics of logic programs due to van Emden and Kowalski to the case where logic programs may contain equations. Within this semantical framework, we reconstruct an example of the initial models of logic programs with equality, namely the standard models due to Goguen and Meseguer.

For computing with a logic program with equality, we propose the use of SLD-resolution for equality axioms, dispensing with any separate inference rules for equality. For the purpose of term evaluation, an SLD-refutation procedure requires the ability to test whether a given term is canonical; thus we propose SLD-resolution with the canonicality test.

In the presence of equations with conditions, the set of canonical terms as irreducible terms becomes undecidable. We therefore construct a semantical model whose domain is a decidable set of constructor terms which we regard as canonical, and then propose SLD-resolution with the constructor term test as a means of computing the denotation of a program determined by this new semantical model.

We incorporate higher-order functions into Horn clauses with equality by using the Schönfinkel-Curry function application operator, constructing a simple but adequate semantical model by means of functional domains, and showing that our equality axioms can be used to compute logic programs into which higher-order functions are incorporated.

Acknowledgements

I am grateful to my supervisor, Professor Maarten van Emden, for supporting me over the last five years, scientifically and financially; for giving me, with remarkable patience, plenty of time to ponder on logic, computing, and programming, even when I could not come up with anything substantial for several months; and for showing me an example of how scholarly work should go on.

Randy Goebel's influence on me has been indirect but firm. It was when I took his course on knowledge representation that I began to place logic programming in wider perspective and to view it as a tool for building intelligent machines—or, to use his term, rational machines. I thank Romas Aleliunas for warning me that higher-order functions are not as tamed an object as I first thought they were; David Poole for asking me several frustrating questions that prompted me to reconsider the issues I thought unimportant; Areski Nait-Abdallah for teaching me the fun and pains of fixed points; Stan Burris for steering me to the relevant work of logicians; Joseph Goguen for serving as the external examiner and for his detailed critical comments on the final draft of the thesis.

Finally I would like to thank my friend Mantis Cheng; without his collaboration on the system AP, this thesis would never have been possible.

To my parents

H.Y. & H.Y.

Not to allow the nesting of singular terms within singular terms within singular terms without limit, in polynomial fashion, and not to allow the facile substitution of complexes for variables and equals, would diminish the power of mathematics catastrophically, even though only practically and not in principle.

— W.V. Quine

But often it is more readable not to write equations between functions but rather equations between values for definitional purposes.

— D.S. Scott

Contents

Chapter 1

Functional and Relational Programming

In this chapter we examine the aims of functional and relational programming, justify the desire for the amalgamation of functional and relational programming, and state the purpose of this dissertation and provide a preview of it.

In Chapters 1 and 2, we use 'relational programming' for programming by Horn clauses, instead of the usual 'logic programming', which we reserve for programming by a certain logical system, i.e., programming in which a program is a sentence of the logical system and computation is deduction in the logical system. Logic programming in the sense of programming by Horn clauses will be restored in Chapters 3–8.

1.1 Aims of Functional and Relational Programming

The conventional programming languages in widespread use today, such as FORTRAN, Pascal, and C, have come to maturity; they are now being used to build commercial software systems as well as for education and experimentation in laboratories. There has been a

growing recognition, however, that the productivity of programmers using such conventional languages is stagnating, and these languages will be unsuitable for machines with a large number of processors running in parallel, made possible by the emerging VLSI technologies. The analysis has revealed that this is because of the inherently *imperative* nature of the conventional languages; they are the languages to tell the computer *how* to compute, usually in terms of change of the contents of the store and sequencing of the commands to be executed. The programmer is stymied by the burden of solving such *control* problems, and the rigid total ordering of commands is inappropriate for exploitation of parallelism.

As an alternative to imperative programming, the idea of *declarative* programming has emerged. A programmer states *what* is to be computed, with no concern about how, in a certain mathematical or logical notation easily understandable by humans. Declarative programming languages free programmers from the control problems and promise to increase programmer productivity. Since a declarative program does not commit itself to how it should be executed, parallelism can be exploited for its execution on a large scale.

As a step toward the ultimate declarative programming, two styles of programming have evolved: *functional* and *relational* programming. We shall now provide a brief review of both.

1.2 A Brief Review of Functional Programming

A program in a functional programming language typically consists of a set of *function definitions*, expressions intended by the user of the language to define and name functions to be computed. Computation in functional programming consists in *evaluation of expressions*.

From the viewpoint of declarative programming, the most important functional programming languages are those that use equational logic. Goguen [36] designed OBJ, a language based on many-sorted first-order equational logic, as a tool for testing algebraic abstract data type specifications. It was subsequently implemented by Tardo [41]. Later, it

was developed into a full-fledged functional language incorporating parameterized generic modules and subsorts [33]. O'Donnell [48,80] designed and implemented an "equational programming language", also based on first-order equational logic. A program in these languages is a set of universally quantified first-order equations that can be used as a *term rewriting system*. Computation in these languages is a special equational deduction called *reduction*. These languages have model-theoretic semantics.

The oldest functional programming language is Lisp [73,74]. Lisp's functions are defined over a domain of objects called *symbolic expressions* ("S-expressions"), using the five elementary functions *atom, eq, car, cdr, cons,* conditional expressions, and recursive function definition. The unique feature of Lisp is that programs themselves are S-expressions, so that any Lisp program can be manipulated as data of a Lisp program.

Lisp has spawned many descendants over the years, notably Scheme [94] and Lispkit Lisp [45]. In these languages, the recognized error of the original Lisp (i.e., the use of dynamic scoping instead of static scoping) was corrected, and it is possible to program with higher-order functions using lambda expressions, which is only possible to a limited degree in the original Lisp.

In his Turing Award Lecture [3], Backus bolstered the general interest in functional programming, introducing the language called FP. In FP, functions are defined from primitive functions using a set of combining forms called functional forms, without using variables. Backus also introduced the "algebra" of FP programs in order to transform, and prove the correctness of, FP programs.

Turner has designed and implemented three functional programming languages: SASL [100], KRC [99], and Miranda [97]. In all these languages, functions are defined by means of "recursion equations". Higher-order functions are defined purely equationally, without lambda expressions. Turner also invented a new way of implementing functional languages [98], a method of compiling function definitions into *combinators*, which had been introduced by Schönfinkel [90] and Curry [19]. For the last few years, this method has been the

subject of intensive research (e.g., [53,18]), in the hope of finding compilation methods producing optimized combinators and ways of efficiently reducing the produced combinators.

Burstall et al. designed HOPE [10] and Milner et al. designed ML [44,78]. Like Turner's languages, they are based on "recursion equations". In HOPE, higher-order functions are defined using lambda expressions. In ML, higher-order functions may be defined purely equationally or using lambda expressions.

1.3 A Brief Review of Relational Programming

Relational programming [63] emerged as an outgrowth of the research on resolution theorem proving spurred by Robinson's pioneer paper [85]; it is realized in the programming language Prolog first designed and implemented by Colmerauer and Roussel [88].

A Prolog program is a set of first-order sentences of the form:

$$\text{for all } x_1, ..., x_n, \quad A \text{ if } B_1 \text{ and } \cdots \text{ and } B_m,$$

where $x_1, ..., x_n$ are the variables in the atomic formulas A and B_i. Computation is activated by an attempt to prove as a theorem of the program a "goal" sentence of the form:

$$\text{there exist } y_1, ..., y_n \text{ such that } G_1 \text{ and } \cdots \text{ and } G_m,$$

where $y_1, ..., y_n$ are the variables in the atomic formulas G_i, and consists in an *SLD-refutation* of the program plus the negation of the goal sentence.

Kowalski and van Emden [25] studied three kinds of semantics of relational programs: model-theoretic, fixpoint, and operational, showing that they are equivalent. Warren [102] invented a compilation method of Prolog programs and showed that Prolog can be implemented efficiently.

1.4 Why Amalgamation?

Although neither functional nor relational programming has achieved the ultimate declarative programming, each has contributed in its own way. In functional programming, for example, there are well-understood theories and implementation techniques of higher-order functions, the so-called infinite data structures like streams, and delayed ("lazy") evaluation of the terms representing those infinite data structures. In relational programming, one can use, at least in principle, any arguments of a predicate as input or output in the form of *questions* and *answers*, a feature suitable for database applications.

In functional programming every object of computation must be cast in terms of functions even if functional notation is clumsy; e.g., relations which are not functions must be treated as truth-valued functions. In relational programming every object of computation must be cast in terms of relations even if relational notation is clumsy; e.g., defining functions in relational form is unacceptably awkward.

The notion of function and the notation to express functional application are ubiquitous in all technical thinking; programmers gain from thinking in terms of functions as much as any mathematicians, scientists, and engineers ever have. Yet relations, as is evidenced by our experience with relational databases, are an important formalism of computer science that cannot be replaced by functions.

Since the primitive notions and their notation mold the way the programmer thinks in formulating and solving problems, the proponents of either programming style have tended to adhere to the style on which they have been reared, and may be blinded to more effective programming paradigms—to a new way of conceiving and solving the problems engendered by the amalgamation of both programming styles. Even a programmer fluent in both styles has had eventually to choose between the two. The division of declarative programming into functional and relational has hampered progress in declarative programming.

To make further progress, we need a programming language that amalgamates functional

and relational programming—a language that allows a programmer to express the objects of computation in functional or relational style at will, to invent and use programming methods made possible only through the mixture of the both styles.

The purpose of this dissertation is to provide a theoretical framework within which such programming languages can be designed. The thesis of the dissertation is that the amalgamation of functional and relational programming can be had by adding equality axioms to Horn clause relational programs, and extending SLD-resolution with a canonicality test.

1.5 A Preview of the Dissertation

In Chapter 2 we provide a background for the amalgamation of functional and relational programming through the use of the equality relation. We explain why and how the equality relation can be used to amalgamate functional and relational programming, and justify why Horn clauses with equality are a promising language for the amalgamation. We then justify our use of equality axioms as opposed to use of special inference rules for equality. We review the literature on the proposed amalgamation and some fundamental work of logicians.

In Chapter 3 we give some technical preliminaries: the definitions of the concepts used in this dissertation as well as a survey of the previous work on which this dissertation depends.

In Chapter 4 we investigate the semantics of logic programs with equality. We generalize the model-theoretic and fixpoint semantics of logic programs studied by van Emden and Kowalski [25] to deal with logic programs with equality, adopting the E-interpretations for clausal logic with equality as defined by J.A. Robinson [84] and G. Robinson and Wos [83]. We prove the existence of the least E-models of logic programs (Theorem 4.6) and their equality to the standard models of logic programs constructed by Goguen and Meseguer [40] (Theorem 4.7). To provide a fixpoint semantics for logic programs with equality, we define a new function from E-interpretations to E-interpretations associated with a logic program with equality and prove that its least fixpoint is equal to the least E-model of the

logic program (Theorem 4.15). We compare our semantics with the one proposed by Jaffar et al. [56,58].

The purpose of Chapter 5 is to amalgamate term evaluation with logic programming by means of equality axioms and show that SLD-resolution of equality axioms is an alternative to narrowing. After arguing that the standard equality axioms are computationally infeasible, we replace the standard equality axioms by two sets of equality axioms where equations are operationally interpreted as confluence to a common term. We prove their correctness (Theorem 5.4), and completeness under ground confluence (Theorems 5.8 and 5.10): a condition in terms of reduction associated with logic programs. Although the two sets of equality axioms are computationally feasible, they are still inadequate for the purpose of term evaluation, because the search space typically contains refutations yielding computed answers that do not reduce terms to their canonical terms. To prune such refutations, we introduce SLD-resolution with the canonicality test, and show that it can be incorporated into the Prolog interpreter to evaluate terms efficiently (Theorems 5.15 and 5.17). Finally, we show that SLD-resolution of our equality axioms gives the operational effect of narrowing, thus demonstrating that SLD-resolution of our equality axioms is an alternative to narrowing.

In Chapter 6, motivated by the undecidability of canonical terms as irreducible terms, we propose a semantical model based on a decidable set of constructor terms, which we regard as canonical terms, for logic programs with equality. This semantical model will be constructed by means of a new continuous function associated with a logic program. We characterize the relations over constructor terms defined by a logic program as the least fixpoint of this function. To compute the least fixpoint of this new function, we propose SLD-resolution with the constructor term test, and prove its correctness (Theorem 6.8) and completeness under ground confluence (Theorem 6.10) with respect to the least fixpoint of the function.

In Chapter 7 we propose logic programming with the Curried Herbrand universe (LPCH),

a method to introduce higher-order functions within the framework of logic programs with equality without use of higher-order logic. In LPCH, every term is built up solely of variables, constant symbols, and the binary functor denoting the Schönfinkel-Curry function application operator. We construct functional domains from the domains of the least E-models of LPCH programs to provide a simple but adequate semantical model for LPCH where every ground term is taken to represent a unary function. We show that, when the equality predicate is interpreted as the intensional equality, the equality axioms shown in Chapter 5 can be used to compute programs in LPCH.

Finally in Chapter 8, we conclude, and suggest some possible directions of future research.

Appendix A contains the fundamentals of first-order logic that are necessary for understanding this dissertation. The index lists the important terms and acronyms appearing in this dissertation.

Chapter 2

Amalgamation and Equality

2.1 Why Horn Clauses with Equality?

We may view functional and relational programming as contributing to the realization of *logic programming*: programming in which a program is a sentence in a certain logical system and computation is deduction in the logical system.[1] Logic programming in such a sense is an instance of declarative programming and has the advantage that programs are intuitively ascertainable, computed answers can be justified in terms of logical semantics, and reasoning about programs can be done using the formalism of the logical system itself. Relational programming uses a restricted form of sentences in first-order logic, i.e., Horn clauses. Functional programming may use equational logic, as has been proposed by some researchers [33,41,48,80].

Goguen and Meseguer [40] recognized that the amalgamation of functional and relational programming is best done by amalgamating the logic that relational programming uses (i.e., Horn clause logic) and the logic that functional programming may use (i.e., equational

[1] Goguen [38] uses the term 'logical programming' for programming in which a program is a sentence in an *institution* and computation is deduction in the institution. Institution [39] is proposed as a mathematically rigorous definition of a logical system.

logic), proposing Horn clause logic with equality. Their proposal set a sound framework for amalgamation—a framework within which to make a further step toward the ultimate logic programming.

The semantic notion on which a functional programmer relies is that of *values of expressions*. Computation in functional programming languages consists in *evaluation of expressions*. The semantic notion on which a relational programmer relies is that of *truth* and *logical implication*. Computation in relational programming languages consists in *proof of theorems*. Superficially, this difference appears to draw a fundamental distinction between functional and relational programming. The distinction disappears, however, once we employ Horn clauses with equality. It is a logical implication of a program (i.e., a set of Horn clauses with equations) that a term and its value (in the sense of normal form) are *equal*. An evaluation of a term becomes a proof of the theorem that the term is equal to its value. By making explicit equational reasoning implicit in term evaluation, we can amalgamate functional and relational programming within the framework of Horn clauses with equality.

2.2 Why Equality Axioms?

Having argued that Horn clauses with equality are a promising language for the amalgamation, we must address the issue of *proof method* for Horn clause logic with equality. Since computation in a logic programming language *is* deduction in a logical system, it is imperative that a proposed proof method be efficiently implementable.

Horn clauses admit efficient computation by means of a special-purpose resolution rule called *SLD-resolution*.[2] Equations admit efficient computation by means of a special-purpose equational deduction called *term reduction* (or *term rewriting*).[3] Observing this,

[2]See Chapter 3, §3.3.2.

[3]See Chapter 3, §3.4.2.

Goguen and Meseguer [40] proposed as a proof method for Horn clauses with equality the combination of SLD-resolution and a special-purpose equational deduction called *narrowing*,[4] which is a generalized version of term reduction.

In this dissertation, we propose to adjoin equality axioms to Horn clauses with equations, using SLD-resolution as a sole proof method and dispensing with separate inference rules for equality such as narrowing. We propose this because

1. SLD-resolution of our equality axioms is at least as efficient as narrowing,
2. we dispense with the burden of implementing some separate inference rules for equality, and of interfacing them with SLD-resolution,
3. we can exploit Prolog implementation techniques,
4. our method generalizes to other useful binary relations such as inequality and partial orderings by axiomatizing those relations, rather than by inventing and implementing new inference rules for those relations.

2.3 Literature Review

In the literature on the amalgamation of functional and relational programming, five approaches may be distinguished:

1. combination of relational programming and Lisp,
2. introduction of logical variables or unification in functional programming,
3. use of conditional rewrite rules,
4. use of extended unification,

[4]See Chapter 3, §3.4.2.

5. use of Horn clauses with equality.

An example of (1) is LOGLISP, designed by Robinson and Sibert [87,86]. It is a programming language amalgamating LOGIC and LISP. LOGIC is a relational programming system implemented in LISP. In LOGLISP, the user can invoke LOGIC from LISP and vice versa. Other examples in this category are Scheme/L designed by Srivastava et al. [93], which combines relational programming and Scheme, and APPLOG designed by Cohen [15].

Abramson [1], Darlington et al. [20], Lindstrom [65], Sato and Sakurai [89], Smolka [92], and You and Subrahmanyam [103] proposed functional programming languages incorporating logical variables by means of unification. Reddy [82] investigated narrowing as the operational semantics of languages in this category. These languages are, however, too inexpressive for full amalgamation.

Fribourg [30,31], Dershowitz and Plaisted [21] proposed programming languages that combine functional and relational programming languages by using conditional rewrite rules. In these papers relations are treated as truth-valued functions, so that the SLD-resolution of relational programming can be obtained as a special case of reduction or narrowing. Our approach is the opposite: we treat equation-based computation (i.e. reduction and narrowing) as a special case (where the theory contains equality axioms) of SLD-resolution.

A more interesting approach to the amalgamation of functional and relational programming is the use of extended unification, i.e., replacement of unification by equation solving. In the context of relational programming, Colmerauer [17] was the first to take this approach in Prolog II, replacing unification by equation solving over a domain of infinite trees. Later he extended his method to deal with other relations, e.g., inequality [16]. Lloyd and van Emden [26] showed that Prolog II can be regarded as a logic programming language. Barbuti et al. [5], Bert and Echahed [8], Dincbas and Vanhentenryck [22], Furbach and Hölldobler [32], Kahn [59], Kornfeld [61], Newton [79], Subrahmanyam and You [95], and Tamaki [96] attempted to incorporate extended unification into relational programming. Elcock and Hoddinott [23,24] studied extended unification by means of the standard equality axioms.

Jaffar et al. [56,58] studied relational programs with extended unification over equational theories axiomatized in Horn equality clauses.

Goguen and Meseguer [40] proposed Eqlog, a language based on Horn clauses with equality. This language is given a model-theoretic semantics and subsumes more *ad hoc* languages mentioned above. Computation in Eqlog is done by the combination of narrowing and SLD-resolution. Gallier and Raatz [34,35] showed a correct and complete refutation method for Horn clauses with equality combining SLD-resolution and extended unification procedures. Levi et al. [4] proposed LEAF, also based on Horn clauses with equality; this language is more restrictive than Eqlog, being a subset of Horn clauses with equality. Computation in LEAF is done by translating a LEAF program into a canonical program to which are applied resolution and another inference rule called atom elimination.

Finally we mention a language outside of the five categories listed above: TABLOG proposed by Malachi et al. [68], a language based on first-order predicate logic with equality. It goes beyond Horn clause logic and allows negation, disjunction, and biconditional. Computation is accomplished by the deductive-tableau method. Although this language is more expressive than any of the above, its practicability as a programming language has not been demonstrated.

Mathematicians have done important related work.

A Soviet mathematician Mal'cev studied *algebraic systems* [69,70]. An algebraic system consists of a nonempty set S, a set of functions over S, and a set of relations over S. The two books cited above contain studies on algebraic systems axiomatized in Horn clauses, and specifically, in *quasi-identities*, which are Horn clauses containing only equations. Work on algebraic systems is closely related to the theory of models, and bears on the project of amalgamation, since any model of Horn clauses with equality can be thought of as an algebraic system in the sense of Mal'cev.

Selman [91] proposed a deductive system for what he calls *equation conjunction implication* languages, which are Horn clauses containing only equations, and showed its com-

pleteness. He also characterized the algebras definable in equation conjunction implication languages. McNulty [75] studied several aspects of "universal Horn logic": equivalence of structures, definability, consistency property, decision problems. He also showed a deductive system for universal Horn logic and proved its completeness. Makowsky [67] proved that a first-order theory admits an initial term model if and only if it is equivalent to a set of universal Horn sentences.

Chapter 3

Preliminaries

This chapter contains the definitions of the concepts used in the dissertation as well as a survey of the previous work on which this dissertation depends. We have to assume some familiarity with the fundamentals of first-order logic, which are included in Appendix A.

In this dissertation we shall call a statement due to the author a *theorem* or a *lemma*, and a statement due to the others a *proposition*.

3.1 Preliminaries on Substitution and Unification

In this section we review the most frequently used syntactic notions. We assume that a first-order language is given.

Definition 3.1 Two expressions e_1 and e_2 are said to be *identical*, written $e_1 \equiv e_2$, if they are the same string of symbols. ■

Definition 3.2 A term or formula is said to be *ground* if it contains no variables. ■

Definition 3.3 A *substitution* θ is a finite set of ordered pairs, written $\{x_1/t_1, ..., x_n/t_n\}$, where x_i is a variable, t_i is a term distinct from x_i, and all x_i's are distinct from one another.

θ is said to be a *ground substitution* if all t_i's are ground. ■

Definition 3.4 The *identity substitution* is the substitution given by the empty set. ■

Definition 3.5 Let α be a term or quantifier-free formula and $\theta = \{x_1/t_1, ..., x_n/t_n\}$ a substitution. The *instance* $\alpha\theta$ of α by θ is the term or formula obtained from α by simultaneously replacing every occurrence of x_i in α by t_i for all $1 \leq i \leq n$. If $\alpha\theta$ is ground, it is said to be a *ground instance* of α. ■

Proposition 3.1 *Let θ and σ be substitutions. Then there is a unique substitution γ such that for all terms or quantifier-free formulas α, $\alpha\gamma \equiv (\alpha\theta)\sigma$.* ■

Definition 3.6 The substitution γ given in Proposition 3.1 is called the *composition* of θ and σ, written $\theta\sigma$. ■

Definition 3.7 Two terms or quantifier-free formulas α_1 and α_2 are said to be a *variant* of each other if there exist substitutions θ_1 and θ_2 such that $\alpha_1\theta_1 \equiv \alpha_2$ and $\alpha_2\theta_2 \equiv \alpha_1$. ■

Definition 3.8 Let α be a term or quantifier-free formula, t an occurrence of a term in α,[1] and u a term. The *replacement* of t by u in α, written $\alpha[t \leftarrow u]$, is the term or formula obtained from α by replacing the occurrence t by the term u. ■

Definition 3.9 Let S be a finite set of terms or atomic formulas. A substitution θ is said to be a *unifier* for S if for all $x, y \in S, x\theta \equiv y\theta$. A unifier θ for S is said to be a most general unifier for S, if for all unifiers σ for S, there is a substitution γ such that $\sigma = \theta\gamma$. ■

For all finite sets of terms or atomic formulas, a most general unifier exists if a unifier exists and is unique up to variable renaming. Various efficient algorithms are known for computing most general unifiers for a finite set of terms or atomic formulas [7,71,81,85].

Definition 3.10 Let t and u be terms. A substitution θ is said to be a *matcher* of t to u if $t\theta \equiv u$. ■

[1]Note that t is an occurrence of a term, not the term itself. In this dissertation, a more rigorous definition for occurrence, such as by means of access paths, is not given.

Note that a matcher of t to u is not necessarily a unifier for t and u. E.g., $\{x/f(x)\}$ is a matcher of x to $f(x)$, but is not a unifier of x and $f(x)$; x and $f(x)$ are not unifiable.

3.2 Preliminaries on Clausal Logic

3.2.1 Syntax

Throughout this dissertation we use Kowalski's syntax [62].

A *clausal sentence* is a possibly infinite set of clauses. A *clause* is a pair of sets of atomic formulas (hereafter abbreviated as *atoms*) written as:

$$A_1, ..., A_m \leftarrow B_1, ..., B_n, \quad m \geq 0, n \geq 0.$$

The set $\{A_1, ..., A_m\}$ is the *conclusion* of the clause; $\{B_1, ..., B_n\}$ is the *premise* of the clause. A *Horn* clause is *positive* or *negative*. A *positive* Horn clause (or *definite* clause) is a clause where $m = 1$. A *negative* Horn clause (or *goal* clause) is a clause where $m = 0$ and $n \geq 1$. The *empty* clause is a clause where $m = 0$ and $n = 0$; it is written '$\square$'.

3.2.2 Semantics

Informally, a clausal sentence is to be understood as a conjunction of its clauses. A clause:

$$A_1, ..., A_m \leftarrow B_1, ..., B_n, \quad m \geq 1, n \geq 0, \tag{3.1}$$

is to be understood as:

$$\text{for all } x_1, ..., x_k, \ A_1 \text{ or } \cdots \text{ or } A_m \text{ if } B_1 \text{ and } \cdots \text{ and } B_n,$$

where $x_1, ..., x_k$ are the variables in the clause. A negative Horn clause:

$$\leftarrow B_1, ..., B_n, \quad n \geq 1, \tag{3.2}$$

is to be understood as:

$$\text{for all } x_1, ..., x_k, \text{ it is not the case that } B_1 \text{ and } \cdots \text{ and } B_n,$$

where $x_1, ..., x_k$ are the variables in the clause. The empty clause is to be understood as a contradiction.

It may be helpful to the readers not familiar with clausal form to remark that clauses (3.1) and (3.2) can be written as, respectively:

$$\forall x_1 \cdots \forall x_k (A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n)$$

and

$$\forall x_1 \cdots \forall x_k \neg (B_1 \wedge \cdots \wedge B_n)$$

in the standard syntax of first-order logic. Consequently, a clausal sentence can be regarded as a sentence in first-order logic and can be given the standard Tarskian model-theoretic semantics. However, the notion of *Herbrand model* plays an especially important role in clausal logic.

The *Herbrand universe* U_L of a first-order language L is the set of all ground terms of the language; if L has no constant symbols, an arbitrary one is added to it. The *Herbrand base* B_L of a language L is the set of all ground atoms of the language. A *Herbrand interpretation* of L is a subset of B_L. Given a clausal sentence S, we often identify the underlying language with the usual logical symbols plus the relation and function symbols appearing in S; in this case, we write U_S and B_S and speak of a Herbrand interpretation of S.

The definition of truth in Herbrand interpretations is given as follows.

Definition 3.11 Let I be a Herbrand interpretation of a language L.

1. A clausal sentence of L is true in I iff each of its clauses is true in I.

2. A clause is true in I iff each of its ground instances is true in I.

3. A ground clause

$$A_1, ..., A_m \leftarrow B_1, ..., B_n$$

is true in I iff at least one of $A_1, ..., A_m$ is true in I or at least one of $B_1, ..., B_m$ is not true in I.

4. A ground atom is true in I iff it is a member of I. ■

Definition 3.12 Let S be a clausal sentence of a language L. A *Herbrand model* of S is a Herbrand interpretation of L in which S is true. ■

The following proposition shows the special importance of Herbrand model in clausal logic.

Proposition 3.2 *A clausal sentence S has a model iff it has a Herbrand model.* ■

Finally, we note one more important proposition of clausal logic. We denote by $M(S)$ the set of all Herbrand models of a sentence S.

Proposition 3.3 *Let S be a clausal sentence and A a ground atom.*

$$A \in \bigcap M(S) \longleftrightarrow S \models A. \quad ■$$

3.3 Preliminaries on Logic Programs

The material contained in this section is taken from [2,25]. For a more comprehensive and detailed treatment of logic programs, see [66].

3.3.1 Semantics

A *logic program* is a set of positive Horn clauses. Logic programs have several important semantical properties not shared by a general clausal sentence.

Proposition 3.4 (Model intersection property) *Let $\mathcal{M}$ be a nonempty set of Herbrand models of a set S of Horn clauses. $\bigcap \mathcal{M}$ is a Herbrand model of S.* ■

This proposition implies that, for all consistent sets S of Horn clauses, $\bigcap M(S)$ is a Herbrand model of S. Clearly, $\bigcap M(S)$ is a subset of every Herbrand model. It is therefore called the *least* Herbrand model of S.

Proposition 3.5 (Existence of least Herbrand model) *Let S be a consistent set of Horn clauses. Then $\bigcap M(S) \in M(S)$.* ■

In addition to model-theoretic semantics, there is another important semantics of logic programs, namely *fixpoint semantics*. Associated with a logic program P is a function T_P from Herbrand interpretations to Herbrand interpretations. The denotation of P as determined by fixpoint semantics is taken to be the least fixpoint of T_P, which coincides with the least Herbrand model $\bigcap M(P)$.

Definition 3.13 Let P be a logic program. The function T_P associated with P from Herbrand interpretations to Herbrand interpretations is defined as follows:

$T_P(I) = \{A \in B_P$: there is a ground instance $A \leftarrow B_1, ..., B_n$, $n \geq 0$, of a clause in P such that $\{B_1, ..., B_n\} \subseteq I\}$. ■

Proposition 3.6 *The function T_P is continuous, i.e., for all increasing sequences*

$$I_0 \subseteq I_1 \subseteq \cdots \subseteq I_n \subseteq I_{n+1} \subseteq \cdots$$

of Herbrand interpretations,

$$T_P(\bigcup_{n=0}^{\infty} I_n) = \bigcup_{n=0}^{\infty} T_P(I_n). \quad ■$$

The power set of a Herbrand base B_P, namely the set of all Herbrand interpretations of P, forms a complete lattice under set inclusion $\subseteq$. It is well known that a continuous function f over a complete lattice has the least fixpoint $lfp(f)$ equal to $\bigcup_{n=0}^{\infty} f^n(\perp)$, where $\perp$ is the least element of the complete lattice.

Proposition 3.7 *Let P be a logic program.*

$$lfp(T_P) = \bigcup_{n=0}^{\infty} T_P^n(\emptyset). \quad \blacksquare$$

To link model-theoretic and fixpoint semantics, we characterize Herbrand models in terms of T_P.

Proposition 3.8 *Let P be a logic program and I a Herbrand interpretation. I is a Herbrand model of P iff $T_P(I) \subseteq I$.* ■

Proposition 3.8 implies that the least Herbrand model of a logic program is equal to the least fixpoint of the associated function.

Proposition 3.9 *Let P be a logic program.*

$$\bigcap M(P) = lfp(T_P). \quad \blacksquare$$

The following proposition summarizes important semantical properties of logic programs.

Proposition 3.10 *Let P be a logic program and A a ground atom.*

$$\{A \in B_P : P \models A\} = \bigcap M(P) = lfp(T_P) = \bigcup_{n=0}^{\infty} T_P^n(\emptyset). \quad \blacksquare$$

In the subsequent chapters, the following notation will be frequently used.

Definition 3.14 Let f be a monotonic function over a complete lattice and $\perp$ the least element of the lattice. We define:

$f \uparrow 0 = \perp$,
$f \uparrow n = f(f \uparrow (n-1))$, for positive integers n,
$f \uparrow \omega = \bigcup_{n=0}^{\infty} f^n(\perp)$. ■

3.3.2 SLD-Resolution

To compute the least Herbrand model of a logic program, a special-purpose resolution theorem prover called *SLD-resolution* is used. To demonstrate that $P \models A$ where A is a ground atom, one uses an *SLD-refutation procedure* to demonstrate that $P \cup \{\leftarrow A\}$ is unsatisfiable.

Definition 3.15 Let $G_i \equiv \leftarrow A_1, ..., A_m$, $C_{i+1} \equiv A \leftarrow B_1, ..., B_n$. G_{i+1} is said to be *derived* from G_i and C_{i+1} by a substitution θ_{i+1} if

1. $A\theta_{i+1} \equiv A_j\theta_{i+1}$ for some $j = 1, ..., m$ (hence θ_{i+1} is a unifier of A and A_j),
2. $G_{i+1} \equiv \leftarrow (A_1, ..., A_{j-1}, B_1, ..., B_n, A_{j+1}, ..., A_m)\theta_{i+1}$.

The atom A_j is called the *selected atom.* ■

Definition 3.16 Let P be a logic program and G a goal clause. An *SLD-derivation* of $P \cup \{G\}$ consists of

1. a (finite or infinite) sequence $G \equiv G_0, G_1, ...$ of goal clauses each with a selected atom,
2. a sequence $C_1, C_2, ...$ of variants of definite clauses in P (called the *input clauses* of the derivation),
3. a sequence $\theta_1, \theta_2, ...$ of substitutions,

such that for all $i \geq 0$, G_{i+1} is derived from G_i and C_{i+1} by θ_{i+1}. The variants C_i are such that for all $j \geq 1$, C_j contains no variables appearing in $G_0, ..., G_{j-1}$ or $C_1, ..., C_{j-1}$. ■

Definition 3.17 An *SLD-refutation* is a finite SLD-derivation that has the empty clause $\square$, which must be the last clause of the derivation. If $G_n \equiv \square$, the refutation has *length* n. ■

Definition 3.18 A *failed* SLD-derivation is a finite SLD-derivation ending with a

nonempty clause whose selected atom does not unify with any conclusion of a clause in the program. ■

The following proposition due to [2] is a weak form of the correctness and completeness of SLD-resolution for goal clauses containing only a single ground atom.

Proposition 3.11 *Let P be a logic program and A a ground atom. $P \models A$ iff there is an SLD-refutation of $P \cup \{\leftarrow A\}$.* ■

Definition 3.19 Let P be a logic program and G a goal clause. Let $\theta_1, \theta_2, ..., \theta_n$ be the sequence of substitutions used in an SLD-refutation of $P \cup \{G\}$. The composition $\theta = \theta_1 \cdots \theta_n$ restricted to the variables of G is called a *computed answer substitution* for $P \cup \{G\}$. ■

In logic programming, a goal clause is intended to be a question representing a computational problem and usually contains variables. A computed answer substitution for $P \cup \{G\}$ is interpreted as an answer to the question G. The following two propositions are due to [14].

Proposition 3.12 (Correctness of SLD-resolution) *Let P be a logic program, $G \equiv \leftarrow B_1, ..., B_n$ a goal clause, and θ a computed answer substitution for $P \cup \{G\}$. Then $P \models \forall[(B_1 \wedge \cdots \wedge B_n)\theta]$.*[2]

Proposition 3.13 (Completeness of SLD-resolution) *Let P be a logic program, $G \equiv \leftarrow B_1, ..., B_n$ a goal clause, and θ be a substitution for the variables of G such that $P \models \forall[(B_1 \wedge \cdots \wedge B_n)\theta]$. Then there is an SLD-refutation of $P \cup \{G\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ.* ■

[2] We express the universal and existential closure of a formula by prefixing $\forall$ and $\exists$ respectively.

3.4 Preliminaries on Equational Logic and Term Rewriting Systems

The material contained in this section is taken from [42] and [52].

3.4.1 Algebras and Equational Logic

Throughout this dissertation, the term 'equational logic' refers to that part of first-order logic which deals with theories consisting of equations.

Definition 3.20 An *equation* is an atomic formula of the form $=(t, u)$; for convenience, it is written as $t = u$. ■

Definition 3.21 An *algebra* $\mathcal{A}$ is a pair $\langle A, F\rangle$ where A is a nonempty set called the *domain, universe*, or *carrier* of $\mathcal{A}$ and F is a family of n-ary operations $f : A^n \to A, n \geq 0$. ■

An equational language is a first-order language whose sole relation symbol is the equality predicate '=' that is a logical symbol denoting the identity relation over the domain. An interpretation I of an equational language L therefore consists of a universe $|I|$ and an assignment of n-ary functions $f^I : |I|^n \to |I|$ to the n-ary function symbols f in L. Thus any interpretation I of an equational language determines the corresponding algebra $\langle |I|, F\rangle$ where $F = \{f^I : f \text{ is a function symbol in } L\}$. Henceforth, each equation in a set of equations is assumed to be universally quantified with respect to its variables.

Definition 3.22 Let E be a set of equations. If $E \models \forall(t = u)$, the equation $t = u$ is said to be *valid* in E. ■

Definition 3.23 Let L be an equational language and E a set of equations in L. Then $\cong_E$ is the finest congruence relation over the set of terms in L containing $\{\langle t\theta, u\theta\rangle : \theta \text{ is a substitution and } t = u \in E\}$. ■

Proposition 3.14 *Let E be a set of equations. Then the equation $t = u$ is valid in E iff $t \cong_E u$.* ■

Definition 3.24 Let L be an equational language and E a set of equations. Then $\cong_{EG}$ is the congruence relation over the Herbrand universe U_L defined by $t \cong_{EG} u \leftrightarrow E \models t = u$. ■

Clearly, $\cong_{EG}$ is $\cong_E$ restricted to U_L. The relation $\cong_{EG}$ is used to construct an initial model of a set of equations. (For the definition of initiality, see Appendix A.)

Proposition 3.15 *Let L be an equational language and E a set of equations. Let I be an interpretation of L such that $|I| = U_L/\cong_{EG}$ and $f^I([t_1]_{\cong_{EG}}, ..., [t_n]_{\cong_{EG}}) = [f(t_1, ..., t_n)]_{\cong_{EG}}$ for all n-place function symbols f in L. Then I is an initial model of E.* ■

In universal algebra (e.g. [9]), the Herbrand universe of an equational language is called a *term algebra* generated by the empty set of variables. The initial model I in Proposition 3.15 is in fact the *quotient algebra* of that term algebra by the congruence relation $\cong_{EQ}$.

3.4.2 Term Rewriting Systems, Reduction, and Narrowing

An important application of equations to computer science is to use them as directed rewrite rules, e.g., functional programming, symbolic computation, and equational theorem proving.

Definition 3.25 A *term rewriting system* is a set of equations such that for all equations $t = u$ in the set, all the variables occurring in u occur in t. ■

Definition 3.26 Let E be a term rewriting system. The *reduction* relation $\rightarrow_E$ associated with E is defined on the set of terms as follows. For all terms t and u, $t \rightarrow_E u$ iff there exist

1. an equation $t' = u'$ in E,

2. an occurrence s of a subterm of t,

3. a matcher θ of t' to s,

such that $u \equiv t[s \leftarrow u'\theta]$. The term t is said to be reducible at occurrence s. ■

Definition 3.27 Let E be a term rewriting system. The *narrowing* relation $\Rightarrow_E$ associated with E is defined on the set of terms as follows [54]. For all terms t and u, $t \Rightarrow_E u$ iff there exist

1. a variant $t' = u'$ of an equation in E that has no variables in common with t,
2. an occurrence s of a non-variable subterm of t,
3. a unifier θ of t' and s,

such that $u \equiv t\theta[s\theta \leftarrow u'\theta]$. The term t is said to be narrowable at occurrence s. ■

For example, if $E = \{0 + x = x, s(x) + y = s(x + y)\}$, then $u + 0$ is not in the domain of $\longrightarrow_E$ and $u + 0 \Rightarrow_E 0$. For ground terms, reduction and narrowing coincide. $\rightarrow_E$ is not always a subset of $\Rightarrow_E$. E.g., let $E = \{x = a\}$. Then $f(y) \rightarrow_E f(a)$ but $f(y) \not\Rightarrow_E f(a)$.

Let $\xrightarrow{*}_E$ be the transitive-reflexive closure of $\rightarrow_E$. A term t is an *E-canonical* (or *E-normal*) term iff there is no term u such that $t \rightarrow_E u$. For all terms t and u, u is an E-canonical form of t iff u is E-canonical and $t \xrightarrow{*}_E u$. We omit the prefix E- when the term rewriting system is apparent from the context.

Definition 3.28 $\rightarrow_E$ is *Noetherian* or *terminating* iff for all terms t, there is no infinite reduction sequence starting from t. ■

Definition 3.29 $\rightarrow_E$ is *confluent* iff for all terms t, u, s, $s \xrightarrow{*}_E t$ and $s \xrightarrow{*}_E u$ imply that there is a term v such that $t \xrightarrow{*}_E v$ and $u \xrightarrow{*}_E v$. ■

A term rewriting system E is said to be Noetherian (confluent) iff $\rightarrow_E$ is Noetherian (confluent), respectively.

Definition 3.30 A Noetherian and confluent term rewriting system is said to be *canonical*. ■

Both confluence and the Noetherian property are undecidable properties of term rewriting systems.

The following propositions show the importance of confluence and the Noetherian property of term rewriting systems.

Proposition 3.16 *Let E be a Noetherian term rewriting system. Then for all terms t, any reduction sequence starting from t leads to a canonical form of t.* ■

Proposition 3.17 *Let E be a confluent term rewriting system. Then for all terms t, a canonical form of t, if it exists, is unique.* ■

Proposition 3.18 *Let E be a canonical term rewriting system. Then for all terms t, any reduction sequence starting from t leads to the unique canonical form of t.* ■

Proposition 3.19 *Let E be a canonical term rewriting system. Then $E \models \forall(t = u)$ iff the canonical forms of t and u are the same. The validity problem in a canonical term rewriting system is decidable.* ■

Proposition 3.19 shows that term reduction is a complete inference system for the validity problem in canonical term rewriting systems. The following two propositions are the correctness and completeness of narrowing for equation solving in canonical term rewriting systems. They are taken from [54].

Proposition 3.20 (Correctness of narrowing) *Let E be a canonical term rewriting system. Suppose that $\theta_1, ..., \theta_n$ is the sequence of substitutions used in a finite narrowing sequence:*

$$h(t_0, u_0) \Rightarrow_E h(t_1, u_1) \Rightarrow_E \cdots \Rightarrow_E h(t_n, u_n)$$

such that t_n and u_n are unifiable by a unifier σ, where h is a function symbol not occurring in t_0 or u_0 or E. Let $\theta = \theta_1 \cdots \theta_n \sigma$. Then $E \models \forall[(t_0 = u_0)\theta]$. ■

Let E be a set of equations, θ_1 and θ_2 substitutions, and V a set of variables. We write $(\theta_1 =_E \theta_2)[V]$ iff for all $x \in V, E \models \forall(x\theta_1 = x\theta_2)$.

Proposition 3.21 (Completeness of narrowing) *Let E be a canonical term rewriting system and V be the set of variables appearing in two terms t_0 and u_0. Let σ be a substitution for V such that $E \models \forall[(t_0 = u_0)\sigma]$. Then there is a finite narrowing sequence:*

$$h(t_0, u_0) \Rightarrow_E h(t_1, u_1) \Rightarrow_E \cdots \Rightarrow_E h(t_n, u_n)$$

with the substitutions $\theta_1, ..., \theta_n$ such that t_n and u_n are unifiable by some unifier δ, where h is a function symbol not occurring in t_0 or u_0 or E, and (letting θ be $\theta_1 \cdots \theta_n\delta$ restricted to V) $(\sigma =_E \theta\gamma)[V]$ for some substitution γ. ■

3.4.3 Reduction Strategies

When a term rewriting system is used for functional programming, reduction is used to evaluate terms. Reduction strategies therefore have important effects on the efficiency and termination of term evaluation. We define two well-known reduction strategies: innermost and outermost. When t is a subterm of a term u, we say that u is a *superterm* of t.

Definition 3.31 A reduction rule is said to be *innermost* if, given a reducible term t, it always reduces a subterm of t whose proper subterms are all irreducible. ■

Definition 3.32 A reduction rule is said to be *outermost* if, given a reducible term t, it always reduces a subterm u of t such that t is irreducible at any proper superterm of u. ■

Chapter 4

Semantics of Logic Programs with Equality

4.1 Introduction

In the standard literature on the semantics of logic programs [2,25,66], two kinds of semantics have been studied: model-theoretic and fixpoint semantics. Model-theoretic semantics gives meaning to logic programs as sentences in first-order logic. In fixpoint semantics, a logic program is regarded as a function from Herbrand interpretations to Herbrand interpretations whose least fixpoint is taken to be the denotation of the program. Moreover, they determine the same denotation: the least Herbrand model of a logic program coincides with the least fixpoint of the function associated with the logic program (cf. § 3.3.1). This semantical duality has proved indispensable in the semantical analysis of logic programs; in particular fixpoint semantics has been used to discover and prove various properties of the least Herbrand models and of SLD-resolution.

In this chapter we show that the semantical duality of logic programs can be carried over to logic programs with equality by using the special kind of interpretation for clausal logic with equality as defined by J.A. Robinson [84] and G. Robinson and Wos [83]. This

will be done in such a way that if a logic program contains no equations, the proposed model-theoretic and fixpoint semantics reduces to the one shown in [25], with an inessential difference necessitated by the presence of equality. Goguen and Meseguer [40] have shown that logic programs with equality possess *initial models*, by constructing an example of initial models called the *standard models*. The initial models of a logic program are determined uniquely up to isomorphism, and thus are representation-independent. Our treatment of model-theoretic semantics can be seen as a reconstruction of the standard models within the framework of [25], relating them to the fixpoint semantics that we propose. Jaffar et al. [58,56] have studied a semantics of a restricted class of logic programs with equality. Apart from this restriction, their semantics is motivated by a viewpoint different from ours. Consequently, there are differences in formulation. A comparison will be made in § 4.5.

Throughout this dissertation, Eq will be the following set of equality axioms: reflexivity, symmetry, transitivity, and substitutivity for function and relation symbols.

$$
\begin{aligned}
Eq = \ \{ \ & x = x \leftarrow && (4.1)\\
, \ & x = y \leftarrow y = x && (4.2)\\
, \ & x = z \leftarrow x = y, y = z && (4.3)\\
, \ & f(x_1, ..., x_i, ..., x_{n_f}) = f(x_1, ..., y_i, ..., x_{n_f}) \leftarrow x_i = y_i && (4.4)\\
, \ & p(x_1, ..., y_i, ..., x_{n_p}) \leftarrow x_i = y_i, p(x_1, ..., x_i, ..., x_{n_p}) && (4.5)\\
\}. &
\end{aligned}
$$

(4.4) is included for all non-constant function symbols f with arity n_f in the language and the i-th argument, for all $1 \leq i \leq n_f$. (4.5) is included for all non-equality relation symbols p with arity n_p in the language and for the i-th argument, for all $1 \leq i \leq n_p$. Eq is a standard axiomatization of equality.

Hereafter, $\models_E$ will be used for logical implication under the truth definition where '=' has the fixed interpretation of the identity relation (i.e., '=' is a logical symbol), while $\models$ will be used for logical implication under the truth definition where '=' receives no special

interpretation at all (i.e., '=' is a relation symbol open to interpretation).[1] For example, $\models_E \forall x(x = x)$ but $\not\models \forall x(x = x)$. The following proposition shows that the two are related through the standard equality axioms Eq.

Proposition 4.1 *Let S be a set of sentences and φ a sentence. Then*

$$S \cup Eq \models \varphi \quad \longleftrightarrow \quad S \models_E \varphi. \ \blacksquare$$

4.2 Clausal Logic with Equality

The purpose of this section is to describe the special kind of interpretation for clausal logic with equality defined by J.A. Robinson [84] and G. Robinson and Wos [83]. Some definitions and theorems used in later parts of the dissertation will also be given.

4.2.1 Syntax

The syntax of clausal logic with equality is the same as that of clausal logic, except that there is a distinguished predicate '=' denoting the identity relation over a domain. A positive Horn clause whose conclusion is an equation will be called an *equational clause*.

We will assume that '=' is a distinguished predicate of *every* language. The Herbrand base of a language therefore always contains all ground instances of all equations. When the convention is adopted of identifying the underlying language with the usual logical symbols plus the relation and function symbols appearing in a clausal sentence, all ground instances of all equations must be included in the Herbrand base, even if '=' does not appear in the sentence at all.

[1] Most textbooks on mathematical logic use $\models$ for our $\models_E$.

4.2.2 Semantics

In clausal logic without equality, the importance of Herbrand interpretations lies in the fact that a clausal sentence has a model if and only if it has a Herbrand model [46]. We naturally seek a special kind of interpretation that plays a role analogous to that of Herbrand interpretations in clausal logic without equality. Two such interpretations have been proposed. One is by J.A. Robinson [84], the other by G. Robinson and Wos [83]. They are two different manifestations of the same method using a congruence relation associated with the equality predicate, which will now be described.

An interpretation I of a first-order language L

1. specifies a nonempty set $|I|$ as the universe of discourse,
2. assigns an n-ary relation $p^I \subseteq |I|^n$ to each n-place relation symbol p,
3. assigns a member c^I of $|I|$ to each constant symbol c,
4. assigns an n-ary function f^I from $|I|^n$ to $|I|$ to each n-place function symbol f.

A Herbrand interpretation I of a language L is an interpretation of L such that

1. $|I|$ is the Herbrand universe U_L,
2. $c^I = c$ for all constant symbols c,
3. f^I is the function such that

$$f^I(t_1, ..., t_n) = f(t_1, ..., t_n)$$

for all n-place function symbols f.

Given a language L, only the assignment of relations to the relation symbols varies from one Herbrand interpretation to another. Each Herbrand interpretation I therefore determines uniquely the corresponding subset I' of the Herbrand base B_L:

$$I' = \{p(t_1, ..., t_n) \in B_L : \langle t_1, ..., t_n \rangle \in p^I\}.$$

Conversely, each subset I' of B_L determines uniquely the corresponding Herbrand interpretation I in which the assignment of relations to the relation symbols is:

$$p^I = \{\langle t_1, ..., t_n \rangle : p(t_1, ..., t_n) \in I'\}.$$

Because of this existence of a bijective mapping between the Herbrand interpretations and the subsets of the Herbrand base, it is customary to identify a Herbrand interpretation with the corresponding subset of the Herbrand base and to use a simplified definition of truth (§3.2.2).

In clausal logic with equality, the special kind of interpretation as defined by J.A. Robinson [84] (henceforth *E-interpretation*) plays a role analogous to that of the Herbrand interpretations. Just as a Herbrand interpretation can be identified with the corresponding subset of the Herbrand base, an E-interpretation can be identified with the corresponding subset of the Herbrand base that forms a *congruence closure*. Such was the line taken by G. Robinson and Wos [83].

J.A. Robinson's E-Interpretations

J.A. Robinson's E-interpretations are defined using *partitions* of the Herbrand universe.

A *partition* π of a set S is a set of nonempty subsets of S with the property that for all $x \in S$, x is a member of exactly one $B \in \pi$. The members of a partition π are called the *blocks* of π. We denote by $[x]_\pi$ the block of which x is a member, omitting the subscript 'π' when the partition is apparent from the context.

Let L be a language and Π_L be the set of all partitions π of the Herbrand universe U_L such that for all n-place function symbols f, if terms t_i and u_i are members of the same block of π for all $1 \leq i \leq n$, then $f(t_1, ..., t_n)$ and $f(u_1, ..., u_n)$ are members of the same block of π. An E-interpretation I of a language L is an interpretation of L such that

1. $|I|$ is a $\pi \in \Pi_L$,

2. $c^I = [c]_\pi$ for all constant symbols c,

3. f^I is the function such that

$$f^I([t_1]_\pi, ..., [t_n]_\pi) = [f(t_1, ..., t_n)]_\pi$$

for all n-place function symbols f. Note that the right-hand side is unique because of the condition on the partitions in Π_L.

The interpretation of the relation symbols varies from E-interpretation to E-interpretation. An *E-model* of a sentence S is an E-interpretation in which S is true. (See Appendix A for the definition of truth in non-Herbrand interpretations.)

Example 4.1 Let S be the clausal sentence $\{a = b \leftarrow\}$. There are two E-interpretations I and J: I with the partition $\{\{a\}, \{b\}\}$ as universe, J with the partition $\{\{a, b\}\}$ as universe. Note that '=' denotes the identity relation. I is not an E-model of S, since $a^I = \{a\} \neq b^I = \{b\}$. J is an E-model of S, since $a^J = b^J = \{a, b\}$. ■

Let S be any clausal sentence and M any model of S. Let $\cong_M$ be the binary relation over the Herbrand universe U_S such that $t \cong_M u$ iff $t = u$ is true in M, namely iff t and u receive the identical denotation in M. The relation $\cong_M$ is easily seen to be a congruence relation over U_S, and therefore the set of all congruence classes induced by $\cong_M$, $U_S/\cong_M$, is a partition in Π_S. Let M' be the E-interpretation defined by

1. $|M'| = U_S/\cong_M$,

2. $p^{M'} = \{\langle [t_1]_{\cong_M}, ..., [t_n]_{\cong_M}\rangle : p(t_1, ..., t_n)$ is true in $M\}$, for all relation symbols p.

Note that the assignment of the constant and function symbols is fixed by $U_S/\cong_M$. It is routine to check that M' is an E-model of S. Hence we have a proposition due to [84].

Proposition 4.2 *A clausal sentence S has a model iff it has an E-model.* ■

This theorem is a generalization of a theorem due to Herbrand (Proposition 3.2) to clausal logic with equality, and implies that theorem by identifying with a Herbrand interpretation

an E-interpretation in which the universe is the partition whose blocks are all singleton sets containing a member of the Herbrand universe.

G. Robinson and Wos's E-interpretations

Given a language L, only the partition π assigned as the universe and the assignment of relations over π to the relation symbols vary from one E-interpretation to another. Now there is a bijective mapping between the E-interpretations (in the sense of J.A. Robinson) and those subsets of the Herbrand base which form a *congruence closure*.[2]

Definition 4.1 Let L be a first-order language and I a subset of B_L. I is said to be a *congruence closure* iff it satisfies the following conditions:

1. for all $t \in U_L$, $t = t \in I$,
2. for all $t, u \in U_L$, if $t = u \in I$ then $u = t \in I$,
3. for all $s, t, u \in U_L$, if $s = t \in I$ and $t = u \in I$ then $s = u \in I$,
4. for all function symbols f, if $t_i = u_i \in I$ for some $1 \leq i \leq n_f$, then $f(t_1, ..., t_i, ..., t_{n_f}) = f(t_1, ..., u_i, ..., t_{n_f}) \in I$,
5. for all non-equality predicates p, if $t_i = u_i \in I$ for some $1 \leq i \leq n_p$, and $p(t_1, ..., t_i, ..., t_{n_p}) \in I$, then $p(t_1, ..., u_i, ..., t_{n_p}) \in I$. ■

Definition 4.2 Let I be a subset of B_L that is a congruence closure. We define on U_L the *congruence relation associated with* I, $\cong_I$, by

$$\forall t, u \in U_L, \;\; t \cong_I u \longleftrightarrow t = u \in I. \quad ■$$

The existence of a bijection between the E-interpretations and the congruence closures is seen as follows. Let I be an E-interpretation with a partition π as universe. Let I' be the subset of B_L:

[2]The term 'congruence closure' was not used by Robinson and Wos and is introduced by the author.

$$
\begin{aligned}
I' &= \{A \in B_L : A \text{ is true in } I\} \\
&= \{t = u : t \text{ and } u \text{ are members of the same block of } \pi\} \cup \\
&\quad \{p(t_1, ..., t_n) : \langle [t_1]_\pi, ..., [t_n]_\pi \rangle \in p^I\}.
\end{aligned}
$$

It is easy to see that I' is a uniquely determined congruence closure.

Conversely, let I be a congruence closure and $\cong_I$ the associated congruence relation. Clearly, the set of all congruence classes induced by $\cong_I$, $U_L/\cong_I$, is a partition in Π_L. We define the E-interpretation I' corresponding to I:

1. $|I'| = U_L/\cong_I$,
2. $p^I = \{\langle [t_1]_{\cong_I}, ..., [t_n]_{\cong_I} \rangle : p(t_1, ..., t_n) \in I\}$.

Thus, there is indeed a bijective mapping between the E-interpretations and those subsets of the Herbrand base which form a congruence closure.

Example 4.2 Let S be the clausal sentence $\{a = b \leftarrow\}$ in Example 4.1. The congruence closure $\{a = a, b = b\}$ corresponds to the E-interpretation I. The congruence closure $\{a = a, b = b, a = b, b = a\}$ corresponds to the E-interpretation J. ■

Now that there is a bijective mapping between the E-interpretations and those subsets of the Herbrand base which form a congruence closure, we identify an E-interpretation with the corresponding congruence closure, just as we identify a Herbrand interpretation with the corresponding subset of the Herbrand base. This was the line taken by Robinson and Wos [83]. Moreover, for all ground equations $t = u$ and E-interpretations I with $|I| = \pi$,

$t = u$ is true in I iff
$[t]_\pi = [u]_\pi$ iff
$t = u$ is a member of the congruence closure corresponding to I,

and for all non-equational ground atoms $p(t_1, ..., t_n)$ and E-interpretations I with $|I| = \pi$,

$p(t_1, ..., t_n)$ is true in I iff

$\langle [t_1]_\pi, ..., [t_n]_\pi \rangle \in p^I$ iff

$p(t_1, ..., t_n)$ is a member of the congruence closure corresponding to I.

Hence a clausal sentence S is true in an E-interpretation I iff S is true in the congruence closure corresponding to I regarded as a Herbrand interpretation. To determine the truth of a clausal sentence S in an E-interpretation I, it suffices to determine the truth of S in the corresponding congruence closure using the simplified truth definition for Herbrand interpretations. The identification of an E-interpretation with the corresponding congruence closure therefore brings a special benefit of being able to treat an E-interpretation simply as a subset of the Herbrand base that is a congruence closure and to use the simplified truth definition for Herbrand interpretations. Since this identification greatly facilitates the formal development of our semantical theory of logic programs with equality (§4.4), we shall adopt the identification throughout the rest of this dissertation, unless otherwise stated.

We now redefine E-interpretations, truth, and E-models.

Definition 4.3 Let L be a first-order language. An *E-interpretation* of L is a subset of B_L which is a congruence closure. ■

We will speak of an E-interpretation of a clausal sentence S when the convention is adopted of identifying the underlying language with the usual logical symbols plus the relation and function symbols appearing in S. The definition of truth in E-interpretations is the same as that for clauses without equality.

Definition 4.4 Let S be a clausal sentence of a language L. An *E-model* of S is an E-interpretation of L in which S is true. ■

We have completed the description of E-interpretations due to J.A. Robinson and to G. Robinson and Wos. In the rest of this chapter, we discuss our contribution: a generalization of the semantical theory in [25] to deal with equality by means of E-interpretations.

E-interpretations and E-models can be characterized in terms of the standard equality

axioms Eq.

Theorem 4.1 *Let L be a language and I a subset of B_L.*

I is an E-interpretation of L $\longleftrightarrow$
I is a Herbrand model of Eq $\longleftrightarrow$
$I = T_{Eq}(I)$.

Proof The first equivalence is obvious. We prove the second equivalence.

if: Suppose that $I = T_{Eq}(I)$. By Proposition 3.8, I is a Herbrand model of Eq.

only if: Suppose that I is a Herbrand model of Eq. By Proposition 3.8, $T_{Eq}(I) \subseteq I$. Thus it suffices to show that $I \subseteq T_{Eq}(I)$. Suppose that $t = u \in I$. Since I is a Herbrand model of Eq, $t = t \in I$ by the reflexivity axiom (4.1). The ground clause $t = u \leftarrow t = t, t = u$ is a ground instance of the transitivity axiom (4.3). Hence $t = u \in T_{Eq}(I)$. Suppose that a non-equational atom $p(t_1, ..., t_n) \in I$. Since I is a Herbrand model of Eq, $t_1 = t_1 \in I$. By the substitutivity axioms for the relation symbols (4.5), $p(t_1, ..., t_n) \in T_{Eq}(I)$. Thus $I \subseteq T_{Eq}(I)$. ■

Theorem 4.2 *Let S be a clausal sentence and I a subset of B_S. I is an E-model of S iff I is a Herbrand model of $S \cup Eq$.*

Proof

I is an E-model of S $\longleftrightarrow$
I is a Herbrand model of S and is an E-interpretation $\longleftrightarrow$ (by Theorem 4.1)
I is a Herbrand model of S and of Eq $\longleftrightarrow$
I is a Herbrand model of $S \cup Eq$. ■

To conclude this section, we state two theorems for later use.

Theorem 4.3 *Let I be a nonempty set of E-interpretations. $\bigcap I$ is an E-interpretation.*

Proof By Theorem 4.1, I is a nonempty set of Herbrand models of Eq. By the model intersection property of Horn clauses (Proposition 3.4), $\bigcap I$ is a Herbrand model of Eq. By Theorem 4.1, $\bigcap I$ is an E-interpretation. ■

We denote by $M_E(S)$ the set of all E-models of a sentence S. The following theorem is a generalization of the first theorem in §5 of [25] (cf. Proposition 3.3) to clausal logic with equality.

Theorem 4.4 *Let S be a clausal sentence and A a ground atom.*

$$A \in \bigcap M_E(S) \longleftrightarrow S \models_E A.$$

Proof

$A \in \bigcap M_E(S) \longleftrightarrow$ (by Theorem 4.2)
$A \in \bigcap M(S \cup Eq) \longleftrightarrow$ (by Proposition 3.3)
$S \cup Eq \models A \longleftrightarrow$ (by Proposition 4.1)
$S \models_E A.$ ■

4.3 The Standard Model

Goguen and Meseguer [40] have shown that a logic program with equality possesses an initial model, which is determined uniquely up to isomorphism; because of this uniqueness up to isomorphism, an initial model of a logic program is proposed as its intended model which is representation-independent. To show the existence of an initial model, Goguen and Meseguer constructed an example of the initial models, called the *standard models*. In [40], the standard model of a logic program is constructed in the same way as an initial model of a set of equations (i.e., the quotient algebra of a term algebra by the congruence relation determined by the set of equations; see Proposition 3.15), namely by replacing the Herbrand universe by the quotient of the Herbrand universe with respect to the congruence relation determined by all ground equations logically implied by the logic program.

Let S be a clausal sentence. We define the congruence relation $\cong_S$ over U_S by:

$$\forall t, u \in U_S, \;\; t \cong_S u \longleftrightarrow S \models_E t = u.$$

Let M_S be an interpretation where

1. $|M_S| = U_S/\cong_S$,
2. $p^{M_S} = \{\langle [t_1]_{\cong_S}, ..., [t_n]_{\cong_S}\rangle : S \models_E p(t_1, ..., t_n)\}$ for all non-equality relation symbols p.
3. $c^{M_S} = [c]_{\cong_S}$ for all constant symbols c,
4. $f^{M_S}([t_1]_{\cong_S}, ..., [t_n]_{\cong_S}) = [f(t_1, ..., t_n)]_{\cong_S}$ for all function symbols f.

The following proposition is due to Goguen and Meseguer [40], where its proof can be found.

Proposition 4.3 *Let S be a consistent set of Horn clauses. M_S is an initial model of S.*

■

4.4 Semantics of Logic Programs with Equality

4.4.1 Model Intersection Property and Least E-Models

We state two semantical properties peculiar to sets of Horn clauses.

Theorem 4.5 (Model intersection property) *Let $\mathcal{M}$ be a nonempty set of E-models of a set S of Horn clauses. $\bigcap \mathcal{M}$ is an E-model of S.*

Proof By Theorem 4.2, $\mathcal{M}$ is a nonempty set of Herbrand models of $S \cup Eq$. By the model intersection property of Horn clauses (Proposition 3.4), $\bigcap \mathcal{M}$ is a Herbrand model of $S \cup Eq$. By Theorem 4.2, $\bigcap \mathcal{M}$ is an E-model of S. ■

This theorem implies that, for all consistent sets S of Horn clauses, $\bigcap M_E(S)$ is an E-model of S.

Theorem 4.6 (Existence of least E-model) *Let S be a consistent set of Horn clauses. Then $\bigcap M_E(S) \in M_E(S)$.*

Proof By Theorem 4.5. ■

Clearly, $\bigcap M_E(S)$ is a subset of every E-model of S. We therefore call $\bigcap M_E(S)$ the *least* E-model of S. The following theorem shows that $\bigcap M_E(S)$ is a reconstruction of the standard model by Goguen and Meseguer [40].

Theorem 4.7 *Let S be a clausal sentence. Let $\cong_{\cap M_E(S)}$ be the congruence relation associated with $\bigcap M_E(S)$, namely the one defined by:*

$$\forall t, u \in U_S, \;\; t \cong_{\cap M_E(S)} u \longleftrightarrow t = u \in \bigcap M_E(S).$$

Then $\cong_S$ is equal to $\cong_{\cap M_E(S)}$. Recall that

$$\forall t, u \in U_S, \;\; t \cong_S u \longleftrightarrow S \models_E t = u.$$

Proof

$t \cong_S u \longleftrightarrow$

$S \models_E t = u \longleftrightarrow$ (by Theorem 4.4)

$t = u \in \bigcap M_E(S) \longleftrightarrow$

$t \cong_{\cap M_E(S)} u.$ ■

4.4.2 Fixpoint Semantics of Logic Programs

In logic programs without equality, associated with a logic program P is a function T_P from Herbrand interpretations to Herbrand interpretations. The denotation of P as determined by *fixpoint semantics* is taken to be the least fixpoint of T_P, which coincides with the least Herbrand model $\bigcap M(P)$.

In this section we propose a new function F_P from E-interpretations to E-interpretations associated with a logic program P to provide a fixpoint semantics for logic programs with equality. The goal is to define F_P so that the least fixpoint of F_P coincides with the least E-model of P and so that F_P can be used to prove various properties of the least E-models and of proof methods for logic programs with equality.

To provide fixpoint semantics, we need a suitable partial ordering over the set of all E-interpretations. Here, taking advantage of the identification of the E-interpretations and the corresponding congruence closures, we simply use the set inclusion relation between congruence closures, which turns out to be a suitable partial ordering.

Let I_1, I_2 be congruence closures and I_1', I_2' be the corresponding E-interpretations (in the sense of J.A. Robinson), respectively. Since I_1 and I_2 are the sets of ground atoms (including equations) true in I_1' and I_2' respectively, we have that $I_1 \subseteq I_2$ iff

$\forall A \in B_L$, if A is true in I_1' then A is true in I_2'.

This allows us to translate the set inclusion relation between congruence closures into the relation between the corresponding E-interpretations (in the sense of J.A. Robinson).

Before defining F_P, it is necessary to prove the existence of the function that maps an arbitrary subset I of B_P to the *least* E-interpretation containing I. It is well known in elementary algebra that for every binary relation R over a set, the least equivalence relation containing R exists and is equal to the intersection of all equivalence relations containing R. A similar result holds for congruence relations over the Herbrand universe. We prove the existence of the least congruence closure containing I, for any given subset I of the Herbrand base. For any subset I of the Herbrand base, let $I' = \{A \leftarrow : A \in I\}$.

Theorem 4.8 *Let L be a language and $I \subseteq B_L$. The least congruence closure containing I exists and is equal to $\bigcap M_E(I')$.*

Proof For all $J \subseteq B_L$, J is a congruence closure containing I iff J is an E-interpretation containing I iff J is an E-model of I'. By Theorem 4.6, $\bigcap M_E(I')$ is the least E-model of I'

and hence is the least congruence closure containing I. ■

$\bigcap M_E(I')$ can be characterized by T_{Eq}.

Theorem 4.9 *Let L be a language and $I \subseteq B_L$.*

$$\bigcap M_E(I') = \bigcup_{n=0}^{\infty} T_{Eq}^n(I \cup \{t = t : t \in U_L\}).$$

Proof

$\bigcap M_E(I') =$ (by Theorem 4.2)
$\bigcap M(I' \cup Eq) =$ (by Proposition 3.10)
$\bigcup_{n=0}^{\infty} T_{I' \cup Eq}^n(\emptyset)$.

We now prove by induction that for all $n \geq 0$,

$$T_{I' \cup Eq}^{n+1}(\emptyset) = T_{Eq}^n(I \cup \{t = t : t \in U_L\}). \tag{4.6}$$

basis: $T_{I' \cup Eq}(\emptyset) = I \cup \{t = t : t \in U_L\} = T_{Eq}^0(I \cup \{t = t : t \in U_L\})$.

induction step: Let $J = I \cup \{t = t : t \in U_L\}$. We note that

$$J = T_{Eq}^0(J) \subseteq \cdots \subseteq T_{Eq}^i(J) \subseteq T_{Eq}^{i+1}(J) \subseteq \cdots. \tag{4.7}$$

(4.7) is easily proved by induction. By (4.7), for all $i \geq 0$

$$I \subseteq T_{Eq}^i(J).$$

Hence, for all $i \geq 0$

$$T_{I' \cup Eq}(T_{Eq}^i(J)) = T_{Eq}(T_{Eq}^i(J)). \tag{4.8}$$

Suppose that (4.6) is true for all $n \leq m$. Consider the case where $n = m + 1$.

$T_{I' \cup Eq}^{m+2}(\emptyset) =$
$T_{I' \cup Eq}(T_{I' \cup Eq}^{m+1}(\emptyset)) =$ (by the induction hypothesis)
$T_{I' \cup Eq}(T_{Eq}^m(I \cup \{t = t : t \in U_L\})) =$ (by (4.8))
$T_{Eq}^{m+1}(I \cup \{t = t : t \in U_L\})$.

This completes the proof of (4.6). So by (4.6),

$$\bigcup_{n=0}^{\infty} T_{I'\cup Eq}^{n+1}(\emptyset) = \bigcup_{n=0}^{\infty} T_{Eq}^{n}(I \cup \{t = t : t \in U_L\}).$$

So,

$$\bigcap M_E(I') = \bigcup_{n=0}^{\infty} T_{Eq}^{n}(I \cup \{t = t : t \in U_L\}). \quad \blacksquare$$

The above two theorems tell us that, given a subset I of the Herbrand base, there is the least superset of I which is a congruence closure. Hence, there is the function which maps a subset I of the Herbrand base to the least superset of I that is a congruence closure.

Definition 4.5 Let L be a language and I a subset of B_L. The function *congruence closure of* I, $cl(I)$, is the one that maps I to the least superset of I which is a congruence closure, i.e.,

$$cl(I) = \bigcap M_E(I') = \bigcup_{n=0}^{\infty} T_{Eq}^{n}(I \cup \{t = t : t \in U_L\}). \quad \blacksquare$$

Theorem 4.10 *Let L be a language. The function $cl(I)$ is continuous, i.e., for all increasing sequences*

$$I_0 \subseteq I_1 \subseteq \cdots \subseteq I_n \subseteq I_{n+1} \subseteq \cdots$$

of subsets of B_L,

$$cl(\bigcup_{n=0}^{\infty} I_n) = \bigcup_{n=0}^{\infty} cl(I_n).$$

Proof Let REF be $\{t = t : t \in U_L\}$. We prove that

$$\bigcup_{n=0}^{\infty} T_{Eq}^{n}((\bigcup_{j=0}^{\infty} I_j) \cup REF) = \bigcup_{j=0}^{\infty}(\bigcup_{n=0}^{\infty} T_{Eq}^{n}(I_j \cup REF)).$$

Obviously,

$$(\bigcup_{j=0}^{\infty} I_j) \cup REF = \bigcup_{j=0}^{\infty}(I_j \cup REF).$$

Since T_{Eq} is continuous, the n-fold composition of T_{Eq}, T_{Eq}^{n}, is continuous for all $n \geq 0$. So,

$$T_{Eq}^{n}((\textstyle\bigcup_{j=0}^{\infty} I_j) \cup REF) =$$
$$T_{Eq}^{n}(\textstyle\bigcup_{j=0}^{\infty}(I_j \cup REF)) =$$
$$\textstyle\bigcup_{j=0}^{\infty} T_{Eq}^{n}(I_j \cup REF).$$

So,

$$\bigcup_{n=0}^{\infty} T_{Eq}^{n}((\bigcup_{j=0}^{\infty} I_j) \cup REF) =$$
$$\bigcup_{n=0}^{\infty} \bigcup_{j=0}^{\infty} T_{Eq}^{n}(I_j \cup REF) =$$
$$\bigcup_{j=0}^{\infty} \bigcup_{n=0}^{\infty} T_{Eq}^{n}(I_j \cup REF). \quad \blacksquare$$

The set of all E-interpretations of a language forms a complete lattice under the set inclusion relation $\subseteq$.

Theorem 4.11 *The set of all E-interpretations of a language L forms a complete lattice under the set inclusion relation $\subseteq$, with $\{t = t : t \in U_L\}$ as the least element and B_L as the greatest element.*

Proof Clearly, $\subseteq$ is a partial ordering on the set of all E-interpretations. Let $\mathcal{I}$ be any set of E-interpretations. The greatest lower bound of $\mathcal{I}$ is $\bigcap \mathcal{I}$, because $\bigcap \mathcal{I}$ is an E-interpretation by Theorem 4.3. The least upper bound of $\mathcal{I}$ is $cl(\bigcup \mathcal{I})$. To see this, consider an upper bound I_0 of $\mathcal{I}$. I_0 is an E-interpretation and $I \subseteq I_0$ for all $I \in \mathcal{I}$. Hence, $\bigcup \mathcal{I} \subseteq I_0$ and I_0 is a congruence closure. By Definition 4.5, $cl(\bigcup \mathcal{I}) \subseteq I_0$. $\blacksquare$

Consider, for example, the least E-interpretation $\{t = t : t \in U_L\}$. This corresponds to the E-interpretation (in the sense of J.A. Robinson) in which the universe is the partition whose blocks are all singleton sets each containing a member of the Herbrand universe, and no non-equality relations hold among any objects whatever. The greatest E-interpretation B_L corresponds to the E-interpretation (in the sense of J.A. Robinson) in which the universe is the partition containing only one element which is the denotation of every ground term, and all relations hold among this single object.

We are now in a position to define the function F_P associated with a logic program P.

Definition 4.6 Let P be a logic program. The function F_P associated with P from E-interpretations to E-interpretations is defined as follows. Let I be an E-interpretation.

$$F_P(I) = cl(T_P(I)). \quad \blacksquare$$

Given an E-interpretation I, $T_P(I)$ contains all ground atoms obtainable from I by a one-step application of modus ponens; $T_P(I)$ by itself is not in general an E-interpretation. The function cl then closes $T_P(I)$ with respect to equality, extending $T_P(I)$ to the least E-interpretation containing it. Hence the equality relation is "built into" F_P, so that, e.g., if atoms $s = t$ and $t = u$ are obtained from I by an application of T_P, then $F_P(I)$ contains $s = u$, regardless of how $s = u$ is actually deduced.

Theorem 4.12 *The function F_P is continuous.*

Proof By definition, F_P is the composition of two continuous functions cl and T_P. Hence, F_P is continuous. ■

It is well known that a continuous function f over a complete lattice has the least fixpoint $lfp(f)$ equal to $\bigcup_{n=0}^{\infty} f^n(\perp)$, where $\perp$ is the least element of the lattice. Since the least element of the lattice of E-interpretations is $\{t = t : t \in U_P\}$, $lfp(F_P)$ is equal to $\bigcup_{n=0}^{\infty} F_P^n(\{t = t : t \in U_P\})$, which we take to be the denotation of P as determined by fixpoint semantics.

Theorem 4.13 *Let P be a logic program.*

$$lfp(F_P) = \bigcup_{n=0}^{\infty} F_P^n(\{t = t : t \in U_P\}). \quad ■$$

To link model-theoretic and fixpoint semantics, we characterize E-models in terms of F_P. The following theorem is a generalization of the theorem in §7 of [25] (cf. Proposition 3.8) to logic programs with equality.

Theorem 4.14 *Let P be a logic program and I an E-interpretation. I is an E-model of P iff $F_P(I) \subseteq I$.*

Proof *only if*: Suppose that I is an E-model of P. By Theorem 4.2, I is a Herbrand model of $P \cup Eq$. So, I is a Herbrand model of P. So, $T_P(I) \subseteq I$ by Proposition 3.8. By

Theorem 4.10, the function cl is continuous and hence monotonic, implying that $cl(T_P(I)) \subseteq cl(I) = I$. Hence, $F_P(I) \subseteq I$.

if: Suppose that I is not an E-model of P. P is not true in I. So, there is a ground instance:

$$A \leftarrow B_1, ..., B_m$$

of a clause in P such that it is not true in I. Hence, $A \notin I$ and $B_i \in I$ for all $1 \leq i \leq m$. But, $A \in T_P(I) \subseteq F_P(I)$. Thus, $F_P(I) \not\subseteq I$. ■

Finally, we can prove the desired result, i.e., that the least E-model of a logic program is equal to the least fixpoint of the associated function.

Theorem 4.15 *Let P be a logic program.*

$$\bigcap M_E(P) = lfp(F_P).$$

Proof

$\bigcap M_E(P) =$ (by Theorem 4.14)
$\bigcap\{I : I$ is an E-interpretation and $F_P(I) \subseteq I\} =$
$lfp(F_P)$. ■

By Theorems 4.4, 4.13, and 4.15, we get

Theorem 4.16 *Let P be a logic program and A a ground atom.*

$$\{A : P \models_E A\} = \bigcap M_E(P) = lfp(F_P) = \bigcup_{n=0}^{\infty} F_P^n(\{t = t : t \in U_P\}). \quad ■$$

Suppose that no equations appear in a logic program P. Then it is clear that

$$\bigcap M_E(P) = \bigcap M(P) \cup \{t = t : t \in U_P\}$$

and

$$F_P(I) = T_P(I) \cup \{t = t : t \in U_P\}.$$

The necessity to add $\{t = t : t \in U_P\}$ arises from the fact that $\models_E \forall(x = x)$; hence the two equations above show that our treatment of the semantics of logic programs is a generalization of the one in [25] to the case where a logic program may contain equations.

4.5 A Comparison with the Semantics of Jaffar et al.

In this section we compare our semantics of logic programs with equality with that studied by Jaffar et al. [58,56]. Jaffar et al. [58] studied the semantics of logic programs augmented with Horn clause equality theory (see below). Later, they proposed a *logic programming language scheme* [56] based on the theory in [58]—a scheme for logic programming languages whose instances are obtained by supplying unification complete equality theories for the scheme's parameter.

A *logic program* as defined by Jaffar et al. [58,56] is a pair (P, E) where P is a *definite clause logic program* and E is a consistent *Horn clause equality theory*. A definite clause logic program is a finite set of definite clauses:

$$A \leftarrow e_1, ..., e_n, B_1, ..., B_m$$

where A and B_i $(1 \leq i \leq m)$ are atoms other than equations, and e_j $(1 \leq j \leq n)$ are equations. A Horn clause equality theory is a possibly infinite set of *Horn equality clauses*. A Horn equality clause is a Horn clause containing only equations.

Let (P, E) be a logic program. E induces the associated congruence relation $\cong_E$ over $U_{(P,E)}$ defined by:

$$\forall t, u \in U_{(P,E)}, \;\; t \cong_E u \longleftrightarrow E \models_E t = u.$$

The *E-base* of (P, E) is defined to be the set

$$\{p([t_1]_{\cong_E}, ..., [t_n]_{\cong_E}) : [t_i]_{\cong_E} \in U_{(P,E)}/\cong_E \text{ and } p \text{ is a non-equality predicate}\}.$$

An *E-interpretation* (in their sense) is a subset of the E-base. Truth in E-interpretation is defined in an obvious way, leading to the existence of least E-models.

The function $T_{(P,E)}$ associated with (P, E) from E-interpretations to E-interpretations is defined by

$$T_{(P,E)}(I) = \{p([t_1]_{\simeq_E}, ..., [t_l]_{\simeq_E}) : \text{there is a ground instance } p(u_1, ..., u_l) \leftarrow e_1, ..., e_n, B_1, ..., B_m \text{ of a clause in } P \text{ such that } [t_i]_{\simeq_E} = [u_i]_{\simeq_E} \text{ for all } 1 \leq i \leq l \text{ and } E \models_E e_j \text{ for all } 1 \leq j \leq n \text{ and } [B_k]_{\simeq_E} \in I \text{ for all } 1 \leq k \leq m\}.$$

$T_{(P,E)}$ is continuous and its least fixpoint is equal to $\bigcup_{n=0}^{\infty} T^n_{(P,E)}(\emptyset)$, which coincides with the least E-model of (P, E).

It is clear that their least E-models coincide with our least E-models, for all logic programs in their sense. (cf. § 4.4.1). There are, however, differences between their E-interpretations and the function $T_{(P,E)}$, and our E-interpretations and the function F_P. It seems that in their logic programs, a Horn clause equality theory E is intended to specify a certain predetermined algebra which is to be superimposed by a user program P [55,57]. Since the Horn clause equality theory is independent of P and hence its models can be constructed without regard to P, it is possible to define E-interpretations and $T_{(P,E)}$ in terms of the quotient domain predetermined by E.

In our logic programs, equations and non-equational atoms have the same status: equations can appear anywhere in a program and equality and other relations can be defined mutually recursively. The E-interpretations are defined using partitions over the Herbrand universe, and the function F_P is defined in such a way as to accumulate logically implied equations (namely the congruence relation) as well as other atoms, rather than to define them in terms of the congruence relation predetermined by all ground equations logically implied by P.

Chapter 5

Amalgamation

5.1 Introduction

Having studied the semantics of logic programs with equality, we address the question how to compute with them. Although the standard equality axioms qualify as a logic program, their use as such is computationally infeasible because the SLD-resolution search space contains many refutations yielding useless answers and many infinite derivations. We therefore replace the standard equality axioms by two sets of equality axioms where equations are operationally interpreted as *confluence to a common term*. After proving the correctness of the two sets of equality axioms, we prove their completeness under a condition given in terms of *reduction* associated with logic programs, namely that of *ground confluence*. Though computationally feasible, the two sets of equality axioms are still inadequate to the purpose of term evaluation, because the search space contains refutations yielding answers that do not reduce terms to their canonical forms. To amalgamate term evaluation with logic programming by pruning such refutations, we propose *SLD-resolution with the canonicality test* and show that its search spaces are amenable to search by the Prolog interpreter. Finally, we show that SLD-resolution of our equality axioms gives the operational effect of narrowing, thus demonstrating that it is an alternative to narrowing.

5.2 Computational Intractability of the Standard Equality Axioms

The standard set Eq of equality axioms:

$$
\begin{aligned}
Eq = \quad \{ \quad & x = x \leftarrow && (5.1)\\
, \quad & x = y \leftarrow y = x && (5.2)\\
, \quad & x = z \leftarrow x = y, y = z && (5.3)\\
, \quad & f(x_1, ..., x_i, ..., x_{n_f}) = f(x_1, ..., y_i, ..., x_{n_f}) \leftarrow x_i = y_i && (5.4)\\
, \quad & p(x_1, ..., y_i, ..., x_{n_p}) \leftarrow x_i = y_i, p(x_1, ..., x_i, ..., x_{n_p}) && (5.5)\\
\}, &
\end{aligned}
$$

is a set of positive Horn clauses.[1] Hence, by the correctness and completeness of SLD-resolution, for all sets P of positive Horn clauses and for all goal clauses G, there is an SLD-refutation of

$$P \cup Eq \cup \{G\} \tag{5.6}$$

if and only if (5.6) is unsatisfiable. However, the SLD-resolution search space of (5.6) typically contains many refutations yielding useless answers and numerous infinite derivations. It is not amenable to efficient search by SLD-refutation procedures using, like Prolog interpreters, the depth-first search strategy.

Take, for example, the following logic program P defining addition:

$$
\begin{aligned}
P = \quad \{ \quad & 0 + x = x \leftarrow && (5.7)\\
, \quad & s(x) + y = s(x + y) \leftarrow && (5.8)\\
\}, &
\end{aligned}
$$

and the standard equality axioms Eq:

$$Eq = \quad \{ \quad x = x \leftarrow \tag{5.9}$$

[1]Note that (5.4) and (5.5) are an axiom schema; see page 30.

$$, \quad x = y \leftarrow y = x \tag{5.10}$$

$$, \quad x = z \leftarrow x = y, y = z \tag{5.11}$$

$$, \quad s(x) = s(y) \leftarrow x = y \tag{5.12}$$

$$, \quad x_1 + x_2 = y_1 + x_2 \leftarrow x_1 = y_1 \tag{5.13}$$

$$, \quad x_1 + x_2 = x_1 + y_2 \leftarrow x_2 = y_2 \tag{5.14}$$

$$\}.$$

Suppose that $s(0)+0$ is to be "evaluated", i.e., a substitution $\theta = \{x/t\}$ is to be found such that

$$P \cup Eq \models (s(0) + 0 = x)\theta.$$

There are many answers, some of which are not informative, such as $\{x/s(0) + 0\}$ and $\{x/s(0+0)\}$. Typically, the user wants x to be instantiated to a canonical form of $s(0) + 0$. However, canonical forms have no special status in the SLD-refutation procedure so that the user can do no better than to force the procedure to generate answers until a canonical form is encountered. Consider one of many SLD-refutations of $P \cup Eq \cup \{\leftarrow s(0) + 0 = x\}$.

$\leftarrow s(0) + 0 = x$

(5.11) |

$\leftarrow s(0) + 0 = y_1, y_1 = x$

(5.8) $|\{y_1/s(0+0)\}$

$\leftarrow s(0+0) = x$

(5.11) |

$\leftarrow s(0+0) = y_2, y_2 = x$

(5.12) $|\{y_2/s(y_3)\}$

$\leftarrow 0+0 = y_3, s(y_3) = x$

(5.7) $| \{y_3/0\}$

$\leftarrow s(0) = x$

(5.9) $|\{x/s(0)\}$

$\square$

$\{x/s(0)\}$

This refutation, which reduces the term to its canonical form, is found by the right choice of the equality axioms and the equations defining functions, among a large number of alternatives. The need for the right choice of the axioms and equations also arises in equation solving, as the following SLD-refutation of $P \cup Eq \cup \{\leftarrow x + s(0) = s(s(0))\}$ shows.

$\leftarrow x + s(0) = s(s(0))$

(5.11) |

$\leftarrow x + s(0) = y_1, y_1 = s(s(0))$

(5.8) $|\{x/s(x_1), y_1/s(x_1 + s(0))\}$

$\leftarrow s(x_1 + s(0)) = s(s(0))$

(5.12) |

$\leftarrow x_1 + s(0) = s(0)$

(5.7) $|\{x_1/0\}$

$\square$

$\{x/s(0)\}$

5.3 The Equality Axioms

We have seen that the standard equality axioms Eq are computationally intractable and that there is need for controlled search. What are adequate controls, i.e., rules for the choice of the equality axioms and equations? It seems that there is no single set of all-purpose controls; evaluation of terms and equation solving, for example, may require radically different search strategies.

For the purpose of functional programming, with which this dissertation is concerned, equations are used *directionally* as left-to-right rewrite rules. An adequate set of control

rules for that purpose is therefore one which, when acted upon SLD-refutation procedures, performs rewriting of terms efficiently.

How are we to implement such controlled rules? One possibility is to express the required control in a metalanguage and to impose it on the standard equality axioms and a logic program. We adopt a different method, which we regard as being more in spirit of logic programming: to use different predicates denoting the same relation, namely equality. The different predicates are so used as to result in a small SLD-resolution search space, while preserving the correctness of the clauses with respect to the intended relation.

In the following, two sets of equality axioms, called Eq_1 and Eq_2, will be presented. The explanation of the axioms will follow immediately after the presentation of the axioms. In §5.9, Eq_1 and Eq_2 will be modified into a form executable by a Prolog interpreter augmented with the ability to test for canonical forms. When executed by such a Prolog interpreter, the two different substitutivity axioms in Eq_1 and Eq_2 give rise to different evaluation rules.

As explained above, the predicates of the form eq_i, $1 \leq i \leq 3$, will be used. They all denote the same relation: equality.

The Equality Axioms $\mathbf{Eq_1}$

The equality axioms Eq_1 is defined as follows.

$$
\begin{aligned}
Eq_1 = \{ \quad & eq_1(x,y) \leftarrow eq_2(x,z), eq_2(y,z) && (5.15)\\
, \quad & eq_2(x,x) \leftarrow && (5.16)\\
, \quad & eq_2(x,z) \leftarrow eq_3(x,y), eq_2(y,z) && (5.17)\\
, \quad & eq_3(x,y) \leftarrow x = y && (5.18)\\
, \quad & eq_3(f(x_1,\ldots,x_i,\ldots,x_{n_f}), f(x_1,\ldots,y_i,\ldots,x_{n_f})) \leftarrow eq_3(x_i,y_i) && (5.19)\\
\}. &&
\end{aligned}
$$

(5.19) is included for all non-constant function symbols f in the language and the i-th argument, for all $1 \leq i \leq n_f$.

The Equality Axioms $\mathbf{Eq_2}$

The equality axioms Eq_2 is defined as follows.

$$
\begin{aligned}
Eq_2 = \quad \{ \quad & eq_1(x,y) \leftarrow eq_2(x,z), eq_2(y,z) && (5.20)\\
, \quad & eq_2(x,x) \leftarrow && (5.21)\\
, \quad & eq_2(x,z) \leftarrow eq_3(x,y), eq_2(y,z) && (5.22)\\
, \quad & eq_3(x,y) \leftarrow x = y && (5.23)\\
, \quad & eq_3(f(x_1,\ldots,x_{n_f}), f(y_1,\ldots,y_{n_f})) \leftarrow \\
& \qquad eq_2(x_1,y_1),\ldots,eq_2(x_{n_f},y_{n_f}) && (5.24)\\
\}. &
\end{aligned}
$$

(5.24) is included for all non-constant function symbols in the language.

The predicate '=' is the one that appears in the conclusions of equational clauses. Hence the relation eq_3 as determined by (5.18) is one-step reduction at outermost functors. The substitutivity axiom (5.19) adds one-step reduction at subterms, however deeply nested, to one-step reduction at outermost functors. The relation eq_3 in Eq_1 is therefore one-step reduction.[2] The clauses (5.16) and (5.17) say that eq_2 is the reflexive-transitive closure of eq_3, hence of one-step reduction. In Eq_2, the substitutivity axiom (5.24) uses eq_2 and so eq_3 is no longer one-step reduction; eq_2 comes out to be the reflexive-transitive closure of one-step reduction. The clauses (5.15) and (5.20) say that the relation eq_1 holds of x and y if x and y reduce to a common term. These two clauses add some equation solving capabilities to Eq_1 and Eq_2 while preserving the directional use of equations. In fact, because of the addition of these clauses, it becomes possible to generalize the reduction relation as usually defined for term rewriting systems to logic programs, and to prove the completeness of Eq_1 and Eq_2 under *ground confluence* (§§5.7 and 5.8).

[2] A precise definition of reduction for logic programs will be given in §5.5.

5.4 Transformed Programs

To use Eq_1 and Eq_2 for computing with a logic program with equality, we use the *transformed program* of it. First, it is necessary to define *homogeneous forms* of logic programs [26].

Definition 5.1 The *homogeneous form* of a non-equational clause[3]

$$p(t_1, ..., t_n) \leftarrow B_1, ..., B_m \quad (m, n \geq 0)$$

is

$$p(x_1, ..., x_n) \leftarrow x_1 = t_1, ..., x_n = t_n, B_1, ..., B_m$$

where $x_1, ..., x_n$ are n different variables not occurring in the original clause. The homogeneous form of an equational clause is itself. ■

Definition 5.2 The *homogeneous form* of a logic program is the set of the homogeneous forms of its clauses. ■

Homogeneous forms of logic programs dispense with the need for the substitutivity axioms for predicates.[4]

Example 5.1 Shown below is a logic program defining the addition function and the relation *less than or equal to*.

$$\begin{array}{rll} P = & \{ & 0 + x = x \leftarrow \\ & , & s(x) + y = s(x + y) \leftarrow \\ & , & leq(0, u) \leftarrow \\ & , & leq(s(u), s(v)) \leftarrow leq(u, v) \\ & \}. & \end{array}$$

[3]Recall that an equational clause is a positive Horn clause whose conclusion is an equation.

[4]Precisely speaking, it means that the least Herbrand model of $P \cup Eq$ coincides with that of $P_{hom} \cup Eq'$, where P_{hom} is the homogeneous form of P and Eq' is Eq minus all the substitutivity axioms for the predicates, not that $P \cup Eq$ is logically equivalent to $P_{hom} \cup Eq'$, which is in general false.

Its homogeneous form is

$$
\begin{aligned}
P' = \ \{ \ & 0 + x = x \leftarrow \\
, \ & s(x) + y = s(x + y) \leftarrow \\
, \ & leq(x, y) \leftarrow x = 0, y = u \\
, \ & leq(x, y) \leftarrow x = s(u), y = s(v), leq(u, v) \\
\}. & \qquad \blacksquare
\end{aligned}
$$

To use Eq_1 or Eq_2, we replace by $eq_1(M, N)$ each equation $M = N$ in the condition of a clause in the homogeneous form.

Definition 5.3 The *transformed equation* of an equation $M = N$ is $eq_1(M, N)$. The *transformed atom* of a non-equational atom A is A itself. The *transformed clause* C_T of a Horn clause C:

$$A \leftarrow B_1, ..., B_m$$

where A may be absent, is obtained from C by replacing every B_i by its transformed atom. The *transformed program* P_T of a logic program P is the set of the transformed clauses of the clauses in the homogeneous form of P. ■

Example 5.2 The transformed program P_T of P shown in Example 5.1 is:

$$
\begin{aligned}
P_T = \ \{ \ & 0 + x = x \leftarrow \\
, \ & s(x) + y = s(x + y) \leftarrow \\
, \ & leq(x, y) \leftarrow eq_1(x, 0), eq_1(y, u) \\
, \ & leq(x, y) \leftarrow eq_1(x, s(u)), eq_1(y, s(v)), leq(u, v) \\
\}. & \qquad \blacksquare
\end{aligned}
$$

Given a logic program P and a goal clause G, computation is achieved by the SLD-resolution of $P_T \cup Eq_1 \cup \{G_T\}$ or of $P_T \cup Eq_2 \cup \{G_T\}$, where P_T is the transformed program of P and G_T is the transformed goal clause of G. In the following, the subscript $-_T$ will be used for transformed atoms, clauses, and programs.

5.5 Reduction Associated with Logic Programs

Under what condition are transformed programs and Eq_1 or Eq_2 complete? In other words, under what condition does eq_1 in $P_T \cup Eq_1$ or $P_T \cup Eq_2$ coincide with $=$ in the least E-model of P? An answer will be provided in terms of a condition on the *reduction relation* associated with a logic program, namely the condition of *ground confluence* (see below). In this section we define the reduction relation associated with a logic program.

In term rewriting systems, a ground term t is reducible if there is a subterm of t matching the left-hand side of some equation (§3.4.2, Definition 3.26). The same definition is not useful in the case of logic programs because of the presence of equational clauses:

$$M = N \leftarrow B_1, ..., B_m. \tag{5.25}$$

Even if a subterm of t matches M by a matcher θ, we would not like to call t reducible unless all $B_i\theta$'s are satisfied. This observation suggests the following definition of reduction in logic programs: a ground term t is reducible using (5.25) if a subterm of t matches M by a matcher θ and

$$P \models_E \exists(B_1 \wedge \cdots \wedge B_m)\theta. \tag{5.26}$$

But this definition induces a computational difficulty; if some B_i is an equation $M' = N'$ then (5.26) requires, in general, two-way rewriting to see if $(M' = N')\theta$ holds. This difficulty has indeed been observed by Kaplan [60] in the context of conditional rewrite rules.[5] We agree with Kaplan that the operational interpretation of an equation that is computationally feasible yet as close semantically to equality as possible is *confluence to a common term*. This is the reason for our adopting the clauses (5.15) and (5.20) in Eq_1 and Eq_2.

Such an interpretation of equations seems to lead to a complicated definition of reduction, as in [60] in the case of conditional rewrite rules. Now the interpretation of an equation

[5] A conditional rewrite rule in the sense of Kaplan is a positive Horn clause:

$$M = N \leftarrow M_1 = N_1, ..., M_n = N_n$$

such that all the variables occurring in M_i, N_i for all $1 \leq i \leq n$, and N occur in M.

as confluence to a common term is what eq_1 in Eq_1 and Eq_2 is. The desired reduction relation can therefore be defined in terms of transformed programs and Eq_1 or Eq_2. We use Eq_1.

Let P be a logic program. The *reduction* relation $\rightarrow_P$ associated with P is defined on the set of all *ground* terms, i.e., the Herbrand universe of P, as follows.

Definition 5.4 For all ground terms M and N,

$$M \rightarrow_P N \overset{def}{\longleftrightarrow} P_T \cup Eq_1 \models eq_3(M, N),$$

where P_T is the transformed program of P. ■

Definition 5.5 For all ground terms M and N,

$$M \leftrightarrow_P N \overset{def}{\longleftrightarrow} M \rightarrow_P N \text{ or } N \rightarrow_P M. \quad ■$$

Let $\overset{*}{\rightarrow}_P$ ($\overset{*}{\leftrightarrow}_P$) be the transitive-reflexive closure of $\rightarrow_P$ ($\leftrightarrow_P$). We omit the subscript when the logic program is apparent from the context.

Definition 5.6 $\rightarrow_P$ is said to be *ground confluent* iff for all ground terms M, N, and S, $S \overset{*}{\rightarrow}_P M$ and $S \overset{*}{\rightarrow}_P N$ imply that there is a ground term T such that $M \overset{*}{\rightarrow}_P T$ and $N \overset{*}{\rightarrow}_P T$. A logic program P is said to be ground confluent iff $\rightarrow_P$ is ground confluent. ■

The following property is a well-known property of confluence, often called the Church-Rosser property.

Proposition 5.1 *Suppose that $\rightarrow_P$ is ground confluent. Then $M \overset{*}{\leftrightarrow}_P N$ iff there is a ground term L such that $M \overset{*}{\rightarrow}_P L$ and $N \overset{*}{\rightarrow}_P L$.* ■

Definition 5.7 $\rightarrow_P$ is said to be *ground Noetherian* (or *ground terminating*) iff for all ground terms M, there is no infinite reduction sequence starting from M. A logic program P is said to be ground Noetherian iff $\rightarrow_P$ is ground Noetherian. ■

Definition 5.8 A ground term M is a *P-canonical* (or *P-normal*) term iff there is no

term N such that $M \rightarrow_P N$. For all ground terms M and N, N is a P-canonical form of M iff N is P-canonical and $M \xrightarrow{*}_P N$. ■

We omit the prefix P- when the logic program is apparent from the context. (We also omit 'ground' when no confusion should arise.) When a logic program is a term rewriting system, the reduction $\rightarrow_P$ coincides with the usual reduction for term rewriting systems [49,52], restricted to ground terms. When a logic program is a set of conditional rewrite rules as defined by Kaplan [60], $\rightarrow_P$ coincides with the reduction as defined by Kaplan for conditional rewrite rules, restricted to ground terms.

The following theorem shows that $\xrightarrow{*}_P$ and eq_2 in Eq_1 coincide.

Theorem 5.1 *Let P be a logic program. Then for all ground terms M and N,*

$$M \xrightarrow{*}_P N \quad \longleftrightarrow \quad P_T \cup Eq_1 \models eq_2(M, N).$$

Proof ($\longrightarrow$) By induction on the n of $\xrightarrow{n}_P$.

basis: Suppose $M \xrightarrow{0} N$. Then $M \equiv N$. Hence by (5.16), $P_T \cup Eq_1 \models eq_2(M, N)$.

induction step: Suppose the theorem is true for reduction of less than $m+1$ steps, $m \geq 0$. Consider M and N such that $M \xrightarrow{m+1} N$. Then $M \rightarrow Q \xrightarrow{m} N$ for some ground term Q. By Def. 5.4,

$$P_T \cup Eq_1 \models eq_3(M, Q).$$

By the induction hypothesis,

$$P_T \cup Eq_1 \models eq_2(Q, N).$$

Hence by (5.17),

$$P_T \cup Eq_1 \models eq_2(M, N).$$

($\longleftarrow$) By induction on the n of $T_{P_T \cup Eq_1} \uparrow n$.

basis: Suppose $eq_2(M, N) \in T_{P_T \cup Eq_1} \uparrow 1$. Then it must be an instance of (5.16) and hence $M \equiv N$. Thus $M \xrightarrow{0} N$.

induction step: Suppose that the theorem is true for the atoms $eq_2(M,N) \in T_{P_T \cup Eq_1} \uparrow n$ where $n \leq m$. Consider $eq_2(M,N) \in T_{P_T \cup Eq_1} \uparrow (m+1)$. If $eq_2(M,N)$ is an instance of (5.16) then $M \equiv N$ and therefore $M \xrightarrow{0} N$. Otherwise there is a ground instance:

$$eq_2(M,N) \leftarrow eq_3(M,Q), eq_2(Q,N)$$

of the clause (5.17) such that $eq_3(M,Q), eq_2(Q,N) \in T_{P_T \cup Eq_1} \uparrow m$. By Def. 5.4, $M \rightarrow Q$. By the induction hypothesis, $Q \xrightarrow{*} N$. Thus $M \xrightarrow{*} N$. ■

5.6 Correctness of $\mathbf{Eq_1}$, $\mathbf{Eq_2}$, and Transformed Programs

Before proving the correctness of $P_T \cup Eq_1$ with respect to $P \cup Eq$, we need a lemma. Let P_{hom} be the homogeneous form of a logic program P.

Lemma 5.1 *Let P be a logic program. Then*

$$P \cup Eq \models P_{hom}.$$

Proof It suffices to show that every clause in P_{hom} is logically implied by $P \cup Eq$. Any equational clause in P_{hom} is logically implied by $P \cup Eq$ because it is in P. Consider a non-equational clause in P_{hom}:

$$p(x_1, ..., x_n) \leftarrow x_1 = t_1, ..., x_n = t_n, B_1, ..., B_n.$$

Clearly,

$$P \models [p(t_1, ..., t_n) \leftarrow B_1, ..., B_n].$$

By the substitutivity axioms for predicates (5.5),

$$Eq \models [p(x_1, ..., x_n) \leftarrow t_1 = x_1, ..., t_n = x_n, p(t_1, ..., t_n)].$$

Thus,

$$P \cup Eq \models [p(x_1, ..., x_n) \leftarrow x_1 = t_1, ..., x_n = t_n, B_1, ..., B_n]. \quad ■$$

We are now in a position to prove the correctness of $P_T \cup Eq_1$ with respect to $P \cup Eq$. We express by the following sentences that the intended interpretation of 'eq_i' $(1 \leq i \leq 3)$ is equality:

$$\forall x \forall y (eq_1(x, y) \leftrightarrow x = y), \tag{5.27}$$

$$\forall x \forall y (eq_2(x, y) \leftrightarrow x = y), \tag{5.28}$$

$$\forall x \forall y (eq_3(x, y) \leftrightarrow x = y). \tag{5.29}$$

Theorem 5.2 *Let P be a logic program. Then*

$$P \cup Eq \cup \{(5.27), (5.28), (5.29)\} \models P_T \cup Eq_1.$$

Proof By Lemma 5.1, $P \cup Eq \models P_{hom} \cup Eq$. Hence it suffices to show that

$$P_{hom} \cup Eq \cup \{(5.27), (5.28), (5.29)\} \models P_T \cup Eq_1.$$

Since P_T is obtained from P_{hom} by replacing $M = N$ in the conclusions by $eq_1(M, N)$, $P_{hom} \cup \{(5.27)\} \models P_T$. Hence it suffices to show that

$$Eq \cup \{(5.27), (5.28), (5.29)\} \models Eq_1.$$

It is easy to see that ((5.1) = reflexivity, (5.2) = symmetry, (5.3) = transitivity, (5.4) = functor substitutivity):

$$\{(5.2), (5.3), (5.27), (5.28)\} \models (5.15),$$

$$\{(5.1), (5.28)\} \models (5.16),$$

$$\{(5.3), (5.28), (5.29)\} \models (5.17),$$

$$\{(5.29)\} \models (5.18),$$

$$\{(5.4), (5.29)\} \models (5.19).$$

Thus

$$Eq \cup \{(5.27), (5.28), (5.29)\} \models Eq_1.$$

Thus the theorem holds. ■

Corollary 5.1 *Let P be a logic program. For all ground atoms A other than those of the form $eq_i(M, N)$ $(1 \leq i \leq 3)$,*

$$P_T \cup Eq_1 \models A \quad \longrightarrow \quad P \cup Eq \models A$$

and for all ground terms M and N,

$$P_T \cup Eq_1 \models eq_1(M, N) \quad \longrightarrow \quad P \cup Eq \models M = N.$$

Proof Suppose

$$P_T \cup Eq_1 \models A.$$

By Theorem 5.2,

$$P \cup Eq \cup \{(5.27), (5.28), (5.29)\} \models A.$$

Since the predicates eq_i $(1 \leq i \leq 3)$ do not occur in A or P or Eq,

$$P \cup Eq \models A.$$

Suppose

$$P_T \cup Eq_1 \models eq_1(M, N).$$

By Theorem 5.2,

$$P \cup Eq \cup \{(5.27), (5.28), (5.29)\} \models eq_1(M, N).$$

Hence $eq_1(M, N)$ is true in every model of $P \cup Eq \cup \{(5.27), (5.28), (5.29)\}$. But in every model of $P \cup Eq \cup \{(5.27), (5.28), (5.29)\}$, the relations eq_1 and $=$ are the same, because of (5.27). So $M = N$ is true in every model of $P \cup Eq \cup \{(5.27), (5.28), (5.29)\}$, and hence

$$P \cup Eq \cup \{(5.27), (5.28), (5.29)\} \models M = N.$$

Since the predicates eq_i $(1 \leq i \leq 3)$ do not occur in $M = N$ or P or Eq,

$$P \cup Eq \models M = N. \quad \blacksquare$$

Theorem 5.3 *Let P be a logic program. For all atoms A other than those of the form $eq_i(M, N)$ $(1 \leq i \leq 3)$,*

$$P_T \cup Eq_1 \models \forall A \quad \longrightarrow \quad P \cup Eq \models \forall A$$

and for all terms M and N,

$$P_T \cup Eq_1 \models \forall(eq_1(M, N)) \quad \longrightarrow \quad P \cup Eq \models \forall(M = N).$$

Proof Suppose that $P_T \cup Eq_1 \models \forall A$. Let $x_1, ..., x_n$ be all the variables occurring in A. Let $a_1, ..., a_n$ be some distinct constant symbols not occurring in P_T or A. Let $\theta = \{x_1/a_1, ..., x_n/a_n\}$. Then $A\theta$ is ground and

$$P_T \cup Eq_1 \models A\theta.$$

By Corollary 5.1,

$$P \cup Eq \models A\theta.$$

Hence by the completeness of SLD-resolution, there is an SLD-refutation of $P \cup Eq \cup \{\leftarrow A\theta\}$. Since a_i's do not occur in P or Eq or A, by textually replacing a_i by x_i $(1 \leq i \leq n)$ in this refutation, we obtain an SLD-refutation of $P \cup Eq \cup \{\leftarrow A\}$ with the identity substitution as the computed answer substitution. Hence by the correctness of SLD-resolution,

$$P \cup Eq \models \forall A.$$

The case for equations is proved analogously. ■

Theorem 5.4 (Correctness of $\mathbf{Eq_1}$ and transformed programs) *Let P be a logic program and G a goal clause $\leftarrow B_1, ..., B_m$. Let θ be any computed answer substitution for $P_T \cup Eq_1 \cup \{G_T\}$. Then*

$$P \models_E \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

Proof Let $G_T \equiv \leftarrow B_1^T, ..., B_m^T$. Let θ be a computed answer substitution for $P_T \cup Eq_1 \cup \{G_T\}$. By the correctness of SLD-resolution,

$$P_T \cup Eq_1 \models \forall(B_1^T \wedge \cdots \wedge B_m^T)\theta.$$

So $P_T \cup Eq_1 \models \forall(B_i^T\theta)$ for all $1 \leq i \leq m$. By Theorem 5.3, $P \cup Eq \models \forall(B_i\theta)$. So

$$P \cup Eq \models \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

By Proposition 4.1,

$$P \models_E \forall(B_1 \wedge \cdots \wedge B_m)\theta. \quad \blacksquare$$

The correctness of $P_T \cup Eq_2$ with respect to $P \cup Eq$ can be proved analogously.

5.7 Completeness of $\mathbf{Eq_1}$ and Transformed Programs under Ground Confluence

The goal of this section is to prove the completeness under ground confluence of $P_T \cup Eq_1$ with respect to $P \cup Eq$. The first theorem to be proved is the converse of Corollary 5.1 on the assumption that P is ground confluent, i.e., that for all non-equational ground atoms A,

$$P \cup Eq \models A \quad \longrightarrow \quad P_T \cup Eq_1 \models A$$

and for all ground terms M and N,

$$P \cup Eq \models M = N \quad \longrightarrow \quad P_T \cup Eq_1 \models eq_1(M, N).$$

By Proposition 4.1 and Theorem 4.16, it suffices to show that

$$A \in F_P \uparrow \omega \quad \longrightarrow \quad P_T \cup Eq_1 \models A$$

and

$$M = N \in F_P \uparrow \omega \quad \longrightarrow \quad P_T \cup Eq_1 \models eq_1(M, N).$$

We first need a lemma.

Lemma 5.2 *Let P be a logic program and $I \subseteq B_P$. Suppose that $M = N \in F_P(I)$. Then there are ground terms K_i*

$$M \equiv K_0, ..., K_i, ..., K_n \equiv N, \quad n \geq 0,$$

such that for every pair of K_i and K_{i+1}, $0 \leq i \leq n-1$, $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$ for some ground terms s and t.

Proof By Definitions 4.5 and 4.6,

$$F_P(I) = \bigcup_{n=0}^{\infty} T_{Eq}^n(T_P(I) \cup \{t = t : t \in U_P\}).$$

We prove the lemma by induction on the power n of T_{Eq} above.

basis: Suppose that $M = N \in T_{Eq}^0(T_P(I) \cup \{t = t : t \in U_P\}) = T_P(I) \cup \{t = t : t \in U_P\}$. If $M = N \in T_P(I)$ then we can take $K_0 \equiv M, K_1 \equiv N, s \equiv M$, and $t \equiv N$. If $M = N \in \{t = t : t \in U_P\}$ then $M \equiv N$ and we can take $K_0 \equiv M \equiv N$.

induction step: Suppose that the lemma is true for all $n \leq m$. Consider M and N such that

$$M = N \in T_{Eq}^{m+1}(T_P(I) \cup \{t = t : t \in U_P\}).$$

Then there is a ground instance:

$$M = N \leftarrow B_1, ..., B_k \tag{5.30}$$

of a clause in Eq such that

$$\{B_1, ..., B_k\} \subseteq T_{Eq}^m(T_P(I) \cup \{t = t : t \in U_P\}).$$

Four cases must be considered.

case 1: Suppose (5.30) is an instance of the reflexivity axiom. We can take $K_0 \equiv M \equiv N$.

case 2: Suppose (5.30) is an instance of the symmetry axiom,

$$M = N \leftarrow N = M.$$

By the induction hypothesis, there are ground terms

$$N \equiv K_0, ..., K_n \equiv M$$

such that $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$. Hence there are terms

$$M \equiv K_n, ..., K_0 \equiv N$$

such that $K_i \equiv K_{i+1}[t \leftarrow s]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$.

case 3: Suppose (5.30) is an instance of the transitivity axiom,

$$M = N \leftarrow M = L, L = N.$$

By the induction hypothesis, there are ground terms

$$M \equiv K_0, ..., K_l \equiv L \equiv K_l, ..., K_n \equiv N$$

such that $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$.

case 4: Suppose (5.30) is an instance of a substitutivity axiom for functors,

$$f(..., M', ...) = f(..., N', ...) \leftarrow M' = N'$$

where $M \equiv f(..., M', ...)$ and $N \equiv f(..., N', ...)$. By the induction hypothesis, there are ground terms

$$M' \equiv K_0, ..., K_n \equiv N'$$

such that $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$. Then there are ground terms

$$f(..., M', ...) \equiv f(..., K_0, ...), ..., f(..., K_n, ...) \equiv f(..., N', ...)$$

such that $f(..., K_{i+1}, ...) \equiv f(..., K_i, ...)[s \leftarrow t]$ and either $s = t \in T_P(I)$ or $t = s \in T_P(I)$. This completes the induction step.

The lemma therefore holds. ■

Theorem 5.5 *Let P be a ground confluent logic program. For all non-equational ground atoms A,*

$$A \in F_P \uparrow \omega \quad \longrightarrow \quad P_T \cup Eq_1 \models A$$

and for all ground terms M and N,

$$M = N \in F_P \uparrow \omega \quad \longrightarrow \quad P_T \cup Eq_1 \models eq_1(M, N).$$

Proof By induction on the n of $F_P \uparrow n$.

basis: We note $F_P \uparrow 0 = \{t = t : t \in U_P\}$. Clearly, $P_T \cup Eq_1 \models eq_1(t,t)$ for all $t \in U_P$, because of (5.15) and (5.16).

induction step: Suppose that the theorem is true for all $n \leq m$. We treat equations first. Suppose that $M = N \in F_P \uparrow (m+1)$. By Lemma 5.2, there are ground terms K_i

$$M \equiv K_0, ..., K_n \equiv N, \quad n \geq 0,$$

such that for every pair of K_i and K_{i+1}, $0 \leq i \leq n-1$, $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(F_P \uparrow m)$ or $t = s \in T_P(F_P \uparrow m)$ for some ground terms s and t.

Suppose that $s = t \in T_P(F_P \uparrow m)$. Then there is a ground instance:

$$s = t \leftarrow B_1, ..., B_k$$

of a clause in P such that $\{B_1, ..., B_k\} \subseteq F_P \uparrow m$. By the induction hypothesis, $P_T \cup Eq_1 \models B_i^T$ for all $1 \leq i \leq k$, where B_i^T is the transformed atom of B_i. Hence $P_T \cup Eq_1 \models s = t$. So $P_T \cup Eq_1 \models eq_3(s,t)$ by (5.18) and hence $P_T \cup Eq_1 \models eq_3(K_i, K_{i+1})$ by (5.19). Using a similar argument, we see that $t = s \in T_P(F_P \uparrow m)$ implies $P_T \cup Eq_1 \models eq_3(K_{i+1}, K_i)$.

So the terms K_i's are such that $P_T \cup Eq_1 \models eq_3(K_i, K_{i+1})$ or $P_T \cup Eq_1 \models eq_3(K_{i+1}, K_i)$. Hence by Definition 5.4, the terms are such that $K_i \rightarrow_P K_{i+1}$ or $K_{i+1} \rightarrow_P K_i$. By Definition 5.5, $K_i \leftrightarrow_P K_{i+1}$ for all $1 \leq i \leq n-1$, hence $M \equiv K_0 \overset{*}{\leftrightarrow}_P K_n \equiv N$. By the ground confluence of P and Proposition 5.1, there is a ground term L such that $M \overset{*}{\rightarrow}_P L$ and $N \overset{*}{\rightarrow}_P L$. By

Theorem 5.1, $P_T \cup Eq_1 \models eq_2(M, L)$ and $P_T \cup Eq_1 \models eq_2(N, L)$. By the clause (5.15) in Eq_1, $P_T \cup Eq_1 \models eq_1(M, N)$.

Next, we treat non-equational atoms. Suppose that $p(t_1, ..., t_k) \in F_P \uparrow (m+1)$. Since $F_P \uparrow (m+1) = F_P(F_P \uparrow m) = cl(T_P(F_P \uparrow m))$, there is a ground instance:

$$p(u_1, ..., u_k) \leftarrow B_1, ..., B_l \tag{5.31}$$

of a clause in P such that $t_i = u_i \in F_P \uparrow (m+1)$ for all $1 \leq i \leq k$ and $\{B_1, ..., B_l\} \subseteq F_P \uparrow m$. By the proof above for the case of equations, $P_T \cup Eq_1 \models eq_1(t_i, u_i)$. By the induction hypothesis, $P_T \cup Eq_1 \models B_i^T$ for all $1 \leq i \leq l$, where B_i^T is the transformed atom of B_i. The ground clause:

$$p(t_1, ..., t_k) \leftarrow eq_1(t_1, u_1), ..., eq_1(t_k, u_k), B_1^T, ..., B_l^T$$

is a ground instance of a clause in P_T, because (5.31) is a ground instance of a clause in P. Hence $P_T \cup Eq_1 \models p(t_1, ..., t_k)$. This completes the induction step.

Thus the theorem holds. ■

Theorem 5.6 *Let P be a ground confluent logic program. For all non-equational ground atoms A,*

$$P \cup Eq \models A \quad \longrightarrow \quad P_T \cup Eq_1 \models A$$

and for all ground terms M and N,

$$P \cup Eq \models M = N \quad \longrightarrow \quad P_T \cup Eq_1 \models eq_1(M, N).$$

Proof By Proposition 4.1, Theorems 4.16 and 5.5. ■

Theorem 5.7 *Let P be a ground confluent logic program. For all non-equational atoms A,*

$$P \cup Eq \models \forall A \quad \longrightarrow \quad P_T \cup Eq_1 \models \forall A$$

and for all terms M and N,

$$P \cup Eq \models \forall(M = N) \quad \longrightarrow \quad P_T \cup Eq_1 \models \forall(eq_1(M, N)).$$

Proof Let $x_1, ..., x_n$ be all the variables occurring in A. Let $a_1, ..., a_n$ be some distinct constant symbols not occurring in P or A. Let $\theta = \{x_1/a_1, ..., x_n/a_n\}$. Then $A\theta$ is ground and

$$P \cup Eq \models A\theta.$$

By Theorem 5.6,

$$P_T \cup Eq_1 \models A\theta.$$

Hence by the completeness of SLD-resolution, there is an SLD-refutation of $P_T \cup Eq_1 \cup \{\leftarrow A\theta\}$. Since a_i's do not occur in P_T or Eq_1 or A, by textually replacing a_i by x_i $(1 \leq i \leq n)$ in this refutation, we obtain an SLD-refutation of $P_T \cup Eq_1 \cup \{\leftarrow A\}$ with the identity substitution as the computed answer substitution. Hence by the correctness of SLD-resolution,

$$P_T \cup Eq_1 \models \forall A.$$

The case for equations is proved analogously. ■

We are now in a position to prove the completeness of Eq_1 and transformed programs under ground confluence, in a form similar to Clark's strong completeness of SLD-resolution [14].

Theorem 5.8 (Completeness under ground confluence of $\mathbf{Eq_1}$ and transformed programs) *Let P be a ground confluent logic program and G a goal clause $\leftarrow B_1, ..., B_m$. Let θ be any substitution for the variables of G such that*

$$P \models_E \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

Then there is an SLD-refutation of $P_T \cup Eq_1 \cup \{G_T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ.

Proof By Proposition 4.1,

$$P \cup Eq \models \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

So $P \cup Eq \models \forall(B_i\theta)$ for all $1 \leq i \leq m$. Hence by Theorem 5.7, $P_T \cup Eq_1 \models \forall(B_i^T\theta)$ for all $1 \leq i \leq m$, where B_i^T is the transformed atom of B_i. So

$$P_T \cup Eq_1 \models \forall(B_i^T \wedge \cdots \wedge B_m^T)\theta.$$

By the completeness of SLD-resolution, there is an SLD-refutation of $P_T \cup Eq_1 \cup \{G_T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ. ■

5.8 Completeness of $\mathbf{Eq_2}$ and Transformed Programs under Ground Confluence

The completeness under ground confluence of $P_T \cup Eq_2$ with respect to $P \cup Eq$ is an immediate consequence of the completeness of Eq_1 and the following lemma, a proof of which is omitted because it is lengthy but straightforward.

Lemma 5.3 *Let P be a logic program. For all ground atoms A other than those of the form $eq_i(M, N)$ $(1 \leq i \leq 3)$,*

$$P_T \cup Eq_1 \models A \longleftrightarrow P_T \cup Eq_2 \models A$$

and for all ground terms M and N,

$$P_T \cup Eq_1 \models eq_1(M, N) \longleftrightarrow P_T \cup Eq_2 \models eq_1(M, N). \ ■$$

Theorem 5.9 *Let P be a ground confluent logic program. For all non-equational ground atoms A,*

$$P \cup Eq \models A \longrightarrow P_T \cup Eq_2 \models A$$

and for all ground terms M and N,

$$P \cup Eq \models M = N \longrightarrow P_T \cup Eq_2 \models eq_1(M, N).$$

Proof By Theorem 5.6 and Lemma 5.3. ■

Theorem 5.10 (Completeness under ground confluence of $\mathbf{Eq_2}$ and transformed programs) *Let P be a ground confluent logic program and G a goal clause $\leftarrow B_1, ..., B_m$. Let θ be any substitution for the variables of G such that*

$$P \models_E \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

Then there is an SLD-refutation of $P_T \cup Eq_1 \cup \{G_T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ.

Proof By Theorem 5.9 and the same argument that derived Theorem 5.8 from Theorem 5.6. ■

5.9 Canonicality

The equality axioms Eq_1 or Eq_2 adjoined to a transformed program are not in general complete with respect to $P \cup Eq$; we have shown a sufficient condition for completeness: ground confluence. However, the SLD-resolution search space of $P_T \cup Eq_1 \cup \{G_T\}$ and of $P_T \cup Eq_2 \cup \{G_T\}$ is considerably smaller than that of $P \cup Eq \cup \{G\}$. The search space is reduced by interpreting an equation operationally as confluence to a common term. Still Eq_1 and Eq_2 are inadequate for the purpose of term evaluation; the search space spanned by Eq_1 or Eq_2 for a goal clause $\leftarrow eq_1(t, x)$ or $\leftarrow eq_1(x, t)$, where t is a ground term to be evaluated, has in general refutations instantiating x to a term that is equal to t but not canonical. For term evaluation, the refutation procedure must avoid such refutations by using the reflexivity axiom (5.16) or (5.21) only when a term has been reduced to a canonical term.

To prune refutations yielding such useless answers for the purpose of term evaluation, we propose *SLD-resolution with the canonicality test* (hereafter called *SLDC-resolution*):

SLD-resolution with the ability to test, given a term t, whether there is a ground instance of t that is canonical or noncanonical. The test will be done when and only when an atom of the form $eq_2(s,t)$ has been resolved.

The function of SLDC-resolution is best described by means of the following two logic programs, which are derived from the equality axioms shown in [27]. Let Eq_1' be the following logic program.

$$
\begin{aligned}
Eq_1' = \ \{ \ & eq_1(x,y) \leftarrow eq_2(x,z), eq_2(y,z) && (5.32)\\
, \ & eq_2(x,x) \leftarrow canonical(x) && (5.33)\\
, \ & eq_2(x,z) \leftarrow non\_canonical(x), eq_3(x,y), eq_2(y,z) && (5.34)\\
, \ & eq_3(x,y) \leftarrow x = y && (5.35)\\
, \ & eq_3(f(x_1,...,x_i,...,x_{n_f}), f(x_1,...,y_i,...,x_{n_f})) \leftarrow eq_3(x_i,y_i) && (5.36)\\
\}. &
\end{aligned}
$$

(5.36) is included for all non-constant function symbols f in the language and the i-th argument, for all $1 \leq i \leq n_f$.

Let Eq_2' be the following logic program.

$$
\begin{aligned}
Eq_2' = \ \{ \ & eq_1(x,y) \leftarrow eq_2(x,z), eq_2(y,z) && (5.37)\\
, \ & eq_2(x,x) \leftarrow canonical(x), && (5.38)\\
, \ & eq_2(x,z) \leftarrow non\_canonical(x), eq_3(x,y), eq_2(y,z) && (5.39)\\
, \ & eq_3(x,y) \leftarrow x = y && (5.40)\\
, \ & eq_3(f(x_1,...,x_{n_f}), f(y_1,...,y_{n_f})) \leftarrow \\
& \qquad eq_2(x_1,y_1),...,eq_2(x_{n_f},y_{n_f}) && (5.41)\\
\}. &
\end{aligned}
$$

(5.41) is included for all non-constant function symbols in the language.

There are no clauses for the predicates '*canonical*' and '*non_canonical*', which denote, respectively, the ground canonical terms and the ground noncanonical terms. Their proce-

dural interpretation is: when an atom of the form *canonical*(t) or *non_canonical*(t) is to be resolved,

1. *canonical*(t) succeeds iff there is a ground instance of t that is canonical, *without instantiating* t,
2. *non_canonical*(t) succeeds iff t is not a variable and there is a ground instance of t that is noncanonical, *without instantiating* t.

The non-variable checking done by the '*non_canonical*' predicate is to prevent infinite derivations that are likely to be caused by an attempt to reduce a variable. Since a term t is never instantiated, the effect of the two predicates is strictly to control the choice between the clauses (5.16) and (5.17) ((5.21) and (5.22)). This circumstance and the correctness of Eq_1 and Eq_2 (§5.6) guarantee that Eq_1' and Eq_2' are still correct with respect to Eq even with the above procedural interpretation of *canonical*(t) and *non_canonical*(t) where t is never instantiated. An SLDC-refutation procedure is defined as an SLD-refutation procedure using Eq_1' or Eq_2'. An SLD-derivation (SLD-refutation) using Eq_1' or Eq_2' will be called an SLDC-derivation (SLDC-refutation).

5.10 The SLDC-Resolution Search Space of $\mathbf{Eq_1'}$ and $\mathbf{Eq_2'}$

SLDC-resolution does not have completeness under ground confluence. The gain, from the viewpoint of term evaluation, is that the search space is further reduced so that it contains no refutations yielding a noncanonical term. In this section we substantiate this claim and show that Eq_1' and Eq_2' are amenable to interpretation by a Prolog interpreter into which the effect of the predicates '*canonical*' and '*non_canonical*' are incorporated.

First, we prove that any SLDC-refutation via R, where R is the computation rule always selecting the leftmost atom, of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$, where M is ground, instantiates x to a term N that is a canonical form of M if N is ground (Theorem 5.12). We need an auxiliary theorem about the relation eq_2.

Theorem 5.11 *Let P be a logic program. Let θ be any computed answer substitution obtained from an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_2(M,N)\}$ via R, where R is the computation rule always selecting the leftmost atom, and M and N are any terms. Then there is a ground substitution σ of $M\theta$ and $N\theta$ such that $N\theta\sigma$ is a canonical form of $M\theta\sigma$.*

Proof By induction on the length of an SLDC-refutation.

basis: Suppose that the length of a refutation is two. Then the clause (5.33) is used. Hence $M\theta \equiv N\theta$ and it has a canonical ground instance.

induction step: Suppose that the theorem holds for the SLDC-refutations of the length less than or equal to m, $m \geq 2$. Consider a refutation of the length $m+1$. The clause (5.34) must be the first input clause. Hence the goal $\leftarrow eq_3(M,L), eq_2(L,N)$ appears in the refutation, where L is some term. Let θ_1 be a computed answer substitution for $P_T \cup Eq_1' \cup \{\leftarrow eq_3(M,L)\}$. Since the relation eq_3 is the one-step reduction, $M\theta_1\gamma \rightarrow_P L\theta_1\gamma$ for any ground substitution γ of $M\theta_1$ and $L\theta_1$. The next goal to be solved is $\leftarrow eq_2(L\theta_1, N\theta_1)$. Let θ_2 be a computed answer substitution for $P_T \cup Eq_1' \cup \{\leftarrow eq_2(L\theta_1, N\theta_1)\}$. The length of this refutation is less than m. By the induction hypothesis, there is a ground substitution σ of $L\theta_1\theta_2$ and $N\theta_1\theta_2$ such that $L\theta_1\theta_2\sigma \xrightarrow{*}_P N\theta_1\theta_2\sigma$ and $N\theta_1\theta_2\sigma$ is canonical. The substitution σ can be extended to a substitution σ' such that $M\theta_1\theta_2\sigma'$ is ground and $M\theta_1\theta_2\sigma' \rightarrow_P L\theta_1\theta_2\sigma'$. Hence $M\theta_1\theta_2\sigma' \xrightarrow{*}_P N\theta_1\theta_2\sigma'$ and $N\theta_1\theta_2\sigma'$ is canonical. As $\theta_1\theta_2$ is a computed answer substitution for $\leftarrow eq_1(M,N)$, σ' is the desired substitution. ■

Corollary 5.2 *Let P be a logic program. Let $\theta = \{x/N\}$ be any computed answer substitution obtained from an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_2(M,x)\}$ via R where R is the computation rule always selecting the leftmost atom and M is a ground term. If N is ground then it is a canonical form of M.*

Proof By Theorem 5.11. ■

Theorem 5.12 *Let P be a logic program. Let $\theta = \{x/N\}$ be any computed answer substitution obtained from an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M,x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow$*

$eq_1(x, M)\}$ via R where R is the computation rule always selecting the leftmost atom and M is a ground term. If N is ground then it is a canonical form of M.

Proof The goal $\leftarrow eq_1(M, x)$ derives the goal $\leftarrow eq_2(M, y), eq_2(x, y)$. Let $\sigma = \{y/N\}$ be a computed answer substitution for $P_T \cup Eq_1' \cup \{\leftarrow eq_2(M, y)\}$. By Theorem 5.11, there is a ground instance of N that is a canonical form of M. The next goal is $\leftarrow eq_2(x, N)$. By the clause (5.33), x is instantiated to N. Clearly, if N is ground then it is a canonical form of M. The goal $\leftarrow eq_2(x, N)$ resolves with the clause (5.34) but the non-variable test fails because x is a variable. The case for $\leftarrow eq_1(x, M)$ can be proved similarly. ■

Theorem 5.12 shows that if an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ is found instantiating x to a ground term N, then N is a canonical form of M. It would be desirable if the Prolog interpreter could always find an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ and of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ for any ground terminating logic program P and for any ground term M. This is seen to be impossible, because solving an atom in the premise of a clause may lead to an infinite derivation, even if the program is ground terminating. Hereafter, therefore, only logic programs that are ground terminating sets of equations will be considered.

Lemma 5.4 *Let P be a ground terminating logic program which is a set of equations. For all equations $M = N$ in P, all the variables occurring in N occur in M.*

Proof Suppose otherwise and let $M = \ldots x \ldots$ be some equation in P where x is a variable not occurring in M. Let θ be a substitution instantiating all the variables in the equation except x to some ground terms. Then it is clear that there is an infinite reduction sequence

$$M\theta \rightarrow_P (\ldots M \ldots)\theta \rightarrow_P (\ldots(\ldots M \ldots)\theta \ldots)\theta \rightarrow_P \ldots .$$

This contradicts the assumption that P is ground terminating. ■

Theorem 5.13 *Let P be a ground terminating logic program which is a set of equations. Any SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ via*

R, where R is the computation rule always selecting the leftmost atom and M is a ground term, instantiates x to a canonical form of M.

Proof By Lemma 5.4 and the fact that the relation eq_3 is the one-step reduction, solving the goal $\leftarrow eq_3(M, x)$, where M is a ground, instantiates x to a ground term N such that $M \rightarrow_P N$. Using this fact, it is straightforward to prove that any SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ via R instantiates x to a ground term. By Theorem 5.12, it is a canonical form of M. ■

In what follows in this section, we assume that the effect of the predicates '*canonical*' and '*non_canonical*' is incorporated into the Prolog interpreter. Theorem 5.13 guarantees that any SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ or of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ via the Prolog interpreter's computation rule instantiates x to a canonical form of M for a ground terminating set P of equations. We prove that the Prolog interpreter finds one (Theorem 5.15).

Theorem 5.14 *Let P be a ground terminating logic program which is a finite set of equations. The Prolog interpreter finds an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_2(M, x)\}$ where M is ground, and instantiates x to a canonical form of M.*

Proof We assign to each ground term M a natural number by the following function α:

> $\alpha(M)$ = the sum of the numbers of the steps of all the possible reduction sequences of M.

The finiteness of P and Lemma 5.4 guarantee that for all ground terms M, there are at most finitely many ground terms N such that $M \rightarrow_P N$. Because of this fact and ground termination, the function α is well defined, i.e., every ground term is assigned a unique natural number. We now prove the theorem by induction on $\alpha(M)$.

basis: If $\alpha(M) = 0$, M is canonical. By (5.33), the Prolog interpreter finds a refutation and

instantiates x to M.

induction step: Suppose that the theorem holds for all terms M such that $\alpha(M) \leq m, m \geq 0$. Consider a term M such that $\alpha(M) = m + 1$. M must be noncanonical. Hence the canonicality test fails and the following goal clause is derived:

$$\leftarrow eq_3(M, y), eq_2(y, x).$$

Since M is noncanonical and the relation eq_3 is the one-step reduction, the goal $\leftarrow eq_3(M, y)$ succeeds and instantiates y to a ground term N such that $M \rightarrow_P N$, by Lemma 5.4. Clearly $\alpha(N) < \alpha(M)$ because M has now reduced to N. By the induction hypothesis, the goal $\leftarrow eq_2(N, x)$ succeeds and instantiates x to a canonical form of N, i.e., of M. ■

Theorem 5.15 *Let P be a ground terminating logic program which is a finite set of equations. The Prolog interpreter finds an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(M, x)\}$ and of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$, where M is ground, and instantiates x to a canonical form of M.*

Proof The goal $\leftarrow eq_1(M, x)$ derives the goal $\leftarrow eq_2(M, y), eq_2(x, y)$. By Theorem 5.14, $\leftarrow eq_2(M, y)$ succeeds and instantiates y to a canonical form N of M. Thus the next goal is $eq_2(x, N)$. By the clause (5.32), x is instantiated to N. The case for $P_T \cup Eq_1' \cup \{\leftarrow eq_1(x, M)\}$ is proved similarly. ■

Theorems 5.11-5.15 and Corollary 5.2 remain true when 'Eq_2'' is substituted for 'Eq_1''. For Eq_2', Theorems 5.14 and 5.15 can be strengthened to guarantee that the length of all the failed branches of the first SLDC-refutations by the Prolog interpreter is one.

Theorem 5.16 *Let P be a ground terminating logic program which is a finite set of equations. The Prolog interpreter finds an SLDC-refutation of $P_T \cup Eq_2' \cup \{\leftarrow eq_2(M, x)\}$, where M is ground, and instantiates x to a canonical form of M. Moreover, all the failed branches constructed in the course of finding the first refutation are caused by failure of canonicality test (5.38) or nonexistence of matching equations (5.40). The length of any failed branch constructed in the course of finding that refutation is one.*

Proof We assign to each ground term M a natural number by the following function α:

> $\alpha(M)$ = the sum of the numbers of the steps of all the possible reduction sequences of M plus the sum of the heights of all the terms appearing in those reduction sequences.[6]

The finiteness of P and Lemma 5.4 guarantee that for all ground terms M, there are at most finitely many ground terms N such that $M \rightarrow_P N$. Because of this fact and ground termination, the function α is well defined, i.e., every ground term is assigned a unique natural number. We now prove the theorem by induction on $\alpha(M)$.

basis: If $\alpha(M) = 0$, M is a constant canonical term. By (5.38), the Prolog interpreter instantiates x to M. There are no failed branches.

induction step: Suppose that the theorem holds for all terms M such that $\alpha(M) \leq m, m \geq 0$. Consider a term M such that $\alpha(M) = m + 1$. If M is canonical, by (5.38) x is instantiated to M. No failed branches are constructed. If M is noncanonical, then by (5.39) the Prolog interpreter derives the goal:

$$\leftarrow eq_3(M, y), eq_2(y, x),$$

after failing the canonicality test and constructing a failed branch of length one. Suppose that M unifies with the left-hand side of an equation $K = L$ by a most general unifier θ. M is $K\theta$. By (5.40), the next goal to be solved is $\leftarrow eq_2(L\theta, x)$. Clearly, $\alpha(L\theta) < \alpha(M)$ because M has now reduced to $L\theta$. By the induction hypothesis, the Prolog interpreter instantiates x to a canonical form of $L\theta$, i.e., of M, and all the failed branches constructed in that process satisfy the condition stated in the theorem. Thus the induction step is checked for the case where M unifies with the left-hand side of an equation.

Suppose that M does not unify with any left-hand side. By (5.41), the next goal is

[6] The height of a term t, $h(t)$, is defined as follows. For a variable or a constant symbol t, $h(t) = 0$. $h(f(t_1, ..., t_n)) = (\max_{1 \leq i \leq n} h(t_i)) + 1$.

derived (letting M be $f(M_1, ..., M_n)$):

$$\leftarrow eq_2(M_1, y_1), ..., eq_2(M_n, y_n), eq_1(f(y_1, ..., y_n), x)$$

after constructing a failed branch of length one. Clearly, $\alpha(M_i) < \alpha(M)$ for all $1 \leq i \leq n$. By the induction hypothesis, each y_i is instantiated to a canonical form N_i of M_i, and all the failed branches constructed in that process satisfy the condition stated in the theorem. Hence the next goal to be solved is:

$$\leftarrow eq_1(f(N_1, ..., N_n), x). \tag{5.42}$$

Since M is noncanonical and irreducible at the top level, at least one M_i is noncanonical, hence at least one N_i differs from M_i. Thus $f(M_1, ..., M_n)$ requires at least one step to reduce to $f(N_1, ..., N_n)$, implying that $\alpha(f(N_1, ..., N_n)) < \alpha(f(N_i, ..., N_n))$. By the induction hypothesis, solving (5.42) instantiates x to a canonical form of $f(N_1, ..., N_n)$, i.e., of M, and all the failed branches constructed in that process satisfy the condition stated in the theorem. This completes the induction step. Thus the theorem holds. ■

Theorem 5.17 *Let P be a ground terminating logic program which is a finite set of equations. The Prolog interpreter finds an SLDC-refutation of $P_T \cup Eq_2' \cup \{\leftarrow eq_1(M, x)\}$ and of $P_T \cup Eq_2' \cup \{\leftarrow eq_1(x, M)\}$, where M is ground, and instantiates x to a canonical form of M. Moreover, all the failed branches constructed in the course of finding the first refutation are caused by failure of canonicality test (5.38) or nonexistence of matching equations (5.40). The length of any failed branch constructed in the course of finding that refutation is one.*

Proof The goal $\leftarrow eq_1(M, x)$ derives the goal $\leftarrow eq_2(M, y), eq_2(x, y)$. By Theorem 5.16, the first goal succeeds and instantiates y to a canonical form N of M and all the failed branches constructed in that process satisfy the condition stated in the theorem. Hence the next goal to be solved is $\leftarrow eq_2(x, N)$. By (5.38), x is instantiated to N. The case for $P_T \cup Eq_2' \cup \{\leftarrow eq_1(x, M)\}$ is proved similarly. ■

For Eq_1', we cannot guarantee that the length of failed branches is one (in Theorem 5.15);

when an SLDC-refutation procedure searches for a unifying subterm (the clauses (5.35) and (5.36)), it may construct long failed branches for a term being reduced that is deeply nested or has many arguments.

If an SLDC-refutation procedure with Eq_1' uses (5.35) first before (5.36), as is the case when the Prolog interpreter is used, it gives the effect of outermost reduction[7] (also called normal order reduction or "lazy" reduction). If an SLDC-refutation procedure with Eq_2' uses (5.41) first whenever one of the arguments of f is noncanonical, it gives the effect of innermost reduction[8] (also called applicative order reduction). If an SLDC-refutation procedure with Eq_2' uses (5.40) first, we have another reduction strategy which, as far as we know, has no name in the literature. Outermost reduction can be much less efficient than innermost reduction for two reasons. First, an SLDC-refutation procedure needs to search, within the term being reduced, for an outermost unifying subterm (the clauses (5.35) and (5.36)). Second, it may introduce recomputation of identical subterms because a one-step reduction may create multiple occurrences of an unevaluated term. The first may be overcome by a pattern-matching algorithm for efficiently searching an outermost unifying subterm (e.g., [80]). The second may be overcome by an implementation technique such as that used by Turner [98], in which multiple occurrences of a subterm are represented by pointers to a shared data structure for the subterm, so that a reduction of any one will result in a simultaneous reduction of all. These potential causes for inefficiency do not exist for innermost reduction; since the arguments of a term are reduced before the term is reduced, any argument is reduced only once, avoiding recomputation of identical subterms, and in the clauses (5.40) and (5.41), there is no room for search for a unifying subterm, as opposed to the clauses (5.35) and (5.36). The advantage of using Eq_2' for innermost reduction is therefore the efficiency of term evaluation.

The advantage of using Eq_1' over using Eq_2' is that Eq_1' is more "complete" in this sense: there are goal clauses G for which no SLDC-refutation with Eq_2' exists but one with Eq_1'

[7]See Definition 3.32.

[8]See Definition 3.31.

does exist. An example is provided by the so-called "infinite data structures".

Let P be the following logic program.

$$\begin{aligned} P = \ \{ \ & int(x) = cons(x, int(s(x))) \leftarrow \\ , \ & car(cons(x,y)) = x \leftarrow \\ , \ & cdr(cons(x,y)) = y \leftarrow \\ \}. & \end{aligned}$$

The first clause defines an "infinite list" of integers, which can be handled by Eq_1' but not by Eq_2'. E.g., an SLDC-refutation of $P_T \cup Eq_1' \cup \{\leftarrow eq_1(car(cdr(int(0))), x)\}$ instantiating x to $s(0)$ exists and can be found by the Prolog interpreter, but no SLDC-refutation of $P_T \cup Eq_2' \cup \{\leftarrow eq_1(car(cdr(int(0))), x)\}$ exists.

5.11 A Comparison with Narrowing

Let E be a canonical term rewriting system. Since E is then confluent, it is ground confluent. The completeness of Eq_1 and Eq_2 under ground confluence therefore tells us that our equality axioms are at least as powerful as narrowing in equation solving. In this section we show that SLD-resolution using Eq_1 indeed gives the operational effect of narrowing.

Theorem 5.18 shows that SLD-refutation of $E \cup Eq_1 \cup \{\leftarrow eq_3(t, x)\}$ gives the effect of narrowing t one step.

Theorem 5.18 *Let E be a term rewriting system. Suppose that $t \Rightarrow_E u$ with the substitution θ. Let θ_1 be θ restricted to the variables of t. Then there is an SLD-refutation of $E \cup Eq_1 \cup \{\leftarrow eq_3(t, x)\}$, where x is a variable not occurring in t, with the computed answer substitution $\theta_2 = \theta_1 \cup \{x/u\}$.*

Proof Suppose that $t \equiv ...s...$ narrows to $u \equiv (...u'...)\theta$ using a variant $t' = u'$ of an equation in E such that $s\theta = t'\theta$. The goal $\leftarrow eq_3(...s..., x)$ derives the goal $\leftarrow eq_3(s, x_1)$ by

repeatedly using the substitutivity axioms (5.19), with the composition of the substitutions used in that derivation being $\{x/...x_1...\}$ where x_1 is a new variable. The goal $\leftarrow eq_3(s, x_1)$ derives the goal $\leftarrow s = x_1$ using (5.18). Since $s\theta \equiv t'\theta$, this goal resolves with $t' = u'$, using the substitution $\theta \cup \{x_1/u'\theta\}$. Hence the computed answer substitution for the original goal is $\theta \cup \{x/(...u'...)\theta\}$ restricted to the variables of $\leftarrow eq_3(t, x)$, namely $\theta_1 \cup \{x/u\}$. ■

Theorem 5.19 shows that SLD-resolution of $E \cup Eq_1 \cup \{\leftarrow eq_2(t, x)\}$ gives the effect of narrowing t zero or more steps.

Theorem 5.19 *Let E be a term rewriting system. Suppose that $t \overset{*}{\Rightarrow}_E u$ with θ being the composition of the substitutions used in narrowing t to u, restricted to the variables of t. Then there is an SLD-refutation of $E \cup Eq_1 \cup \{\leftarrow eq_2(t, x)\}$, where x is a variable not occurring in t, with the computed answer substitution $\theta' = \theta \cup \{x/u\}$.*

Proof By induction on the lengths of narrowing sequences, using Theorem 5.18. ■

To solve an equation $t_0 = u_0$ in a canonical term rewriting system E using narrowing, one finds a narrowing sequence:

$$h(t_0, u_0) \Rightarrow_E h(t_1, u_1) \Rightarrow_E \cdots \Rightarrow_E h(t_n, u_n) \tag{5.43}$$

such that t_n and u_n are unifiable [54] (cf. Propositions 3.20 and 3.21). Each narrowing step $h(t_i, u_i) \Rightarrow_E h(t_{i+1}, u_{i+1})$ in (5.43) corresponds to an SLD-derivation from $\leftarrow eq_2(t_i, x), eq_2(u_i, x)$ to $\leftarrow eq_2(t_{i+1}, x), eq_2(u_{i+1}, x)$ where x is a variable not occurring in t_i or u_i, in the following sense.

If $h(t_i, u_i) \Rightarrow_E h(t_{i+1}, u_{i+1})$ then either $t_i \Rightarrow_E t_{i+1}$ or $u_i \Rightarrow_E u_{i+1}$ but not both, because h is a function symbol not occurring in E. Suppose that $t_i \Rightarrow_E t_{i+1}$ with the substitution θ. Let θ_1 be θ restricted to the variables of t_i. Then clearly $u_{i+1} \equiv u_i\theta_1$. The goal

$$\leftarrow eq_2(t_i, x), eq_2(u_i, x)$$

derives the goal

$$\leftarrow eq_3(t_i, y), eq_2(y, x), eq_2(u_i, x)$$

using (5.17). By Theorem 5.18, $E \cup Eq_1 \cup \{\leftarrow eq_3(t_i, y)\}$ has a refutation with $\theta_2 = \theta_1 \cup \{y/t_{i+1}\}$. Hence the next goal is

$$\leftarrow eq_2(t_{i+1}, x), eq_2(u_i\theta_1, x),$$

which is

$$\leftarrow eq_2(t_{i+1}, x), eq_2(u_{i+1}, x).$$

A similar argument applies to the case where $u_i \Rightarrow_E u_{i+1}$.

The last step of unifying t_n and u_n by a unifier δ corresponds to the SLD-refutation of $E \cup Eq_1 \cup \{\leftarrow eq_2(t_n, x), eq_2(u_n, x)\}$ using (5.16) twice, instantiating x to $t_n\delta \equiv u_n\delta$.

Finally, finding a narrowing sequence (5.43) corresponds to finding an SLD-refutation of $E \cup Eq_1 \cup \{\leftarrow eq_1(t_0, u_0)\}$, as the goal $\leftarrow eq_1(t_0, u_0)$ derives the goal $\leftarrow eq_2(t_0, x), eq_2(u_0, x)$.

Chapter 6

A Semantical Model Based on Constructor Terms

6.1 Introduction

In the previous chapter we have seen the necessity to use the canonicality predicate to prune refutations yielding answers that do not reduce terms to canonical forms. The canonicality predicate turns out to be undecidable. We therefore abandon irreducible terms as canonical terms and adopt a decidable set of *constructor terms*—terms denoting data objects. An effect of adopting constructor terms as canonical terms is that a logic program must be regarded as defining relations over the set of constructor terms. We characterize the relations over constructor terms defined by a logic program P as the least fixpoint of a new continuous function associated with P. To compute the least fixpoint of this new function, we propose a set of equality axioms with the constructor term predicate, and a more realistic method—*SLD-resolution with the constructor term test.*

6.2 Undecidability of Canonical Terms

We have seen in §5.10 that the search space of SLDC-resolution for term evaluation does not contain any refutations yielding noncanonical terms. To prune such useless refutations, it was essential to equip SLD-refutation procedures with the supposed ability to test, given a term t, whether or not there is a ground instance of t that is canonical or noncanonical. In fact, there is no algorithm for such a test: there is a logic program for which the predicate $canonical(t)$ is undecidable[1] and $non\_canonical(t)$ is partially decidable. (For the definition of the two predicates, see §5.9)

Kaplan [60] has shown that there is a set of *conditional rewrite rules*[2] for which it is partially decidable whether a ground term t is noncanonical. When a logic program is a set of conditional rewrite rules as defined by Kaplan, the reduction relation for the logic program (cf. Def. 5.4) coincides with the reduction relation for the conditional rewrite rules as defined by Kaplan, restricted to ground terms. Kaplan's result therefore implies that it is at best partially decidable whether a ground term is noncanonical for a logic program. The problem that the predicate $non\_canonical(t)$ has to solve, i.e., to test whether t is not a variable and there is a ground instance of t that is noncanonical, is then seen to be partially decidable at best. It is in fact partially decidable; there is an effective procedure that terminates if there is a ground instance of t that is noncanonical, namely any complete SLD-refutation procedure searching for a refutation of $P_T \cup Eq_1 \cup \{\leftarrow eq_3(t, x)\}$, where x is a variable not occurring in t. Kaplan's result implies also that it is undecidable whether a ground term is canonical, since the set of ground canonical terms is the complement of the set of ground noncanonical terms. Hence the problem that the predicate $canonical(t)$ has to solve, i.e., to test whether there is a ground instance of t that is canonical, is undecidable.

Since the two predicates are essential to reducing search spaces, this difficulty must be

[1]By an undecidable predicate or problem we mean, in this chapter, a predicate or problem which is neither decidable nor partially decidable.

[2]See the footnote on page 58 for the definition.

circumvented. One way is to restrict ourselves to a class of logic programs for which the two predicates are decidable, e.g., the class of logic programs in which all equational clauses are just equations without conditions. In this chapter we take a different approach by abandoning, for the purpose of term evaluation, the notion of canonicality as irreducibility and seek another notion of what canonicality consists in.

6.3 Canonicality = Irreducibility?

It is customary in logic, mathematics, and computer science to talk of *canonical* terms (or *normal* forms) as terms fully evaluated or reduced. In λ-calculus and term rewriting systems, for example, rules of reduction are laid down, and then a term is said to be canonical if there are no terms to which the term is reducible. The reduction rules themselves are used so that whenever a term M reduces to a term N, N is more informative than M about the value of M; irreducible terms are the terms that are most informative about the values of respective terms. 'Canonical' is used in the sense of having the maximal information about the values of terms.

When a certain formal system is used for the purpose of functional programming, an additional condition may well be imposed on what are taken to be canonical terms: the condition that canonical terms denote *data objects*. Data objects are what programs manipulate: numbers, strings of characters, lists, sequences, trees, etc. When this stronger notion of canonicality is adopted, an irreducible term that fails to denote a data object does not qualify as a canonical term. Typically, one defines functions over the data objects constructible in one's programming language, and it is the terms denoting such data objects that one is prepared to call canonical. And it is to the terms denoting such data objects that noncanonical terms evaluate.

In some formal systems, the set of irreducible terms and the set of terms denoting data objects may coincide. Martin-Löf [72] states, after defining canonical terms to be irreducible terms in his system of intuitionistic type theory:

> What I call canonical and noncanonical forms of expression correspond to the constructors and selectors, respectively, of Landin(1964).[3] In the context of programming, they might also aptly be called data and program forms, respectively.

We will see in the next section that such is not the case for logic programs with equations.

The additional condition that canonical terms denote data objects may be unreasonable for some applications such as theorem proving in equational theories or symbolic computation. When testing the validity of an equation in an equational theory represented by a canonical term rewriting system, for example, one needs to test if two irreducible terms are identical (cf. Proposition 3.19); here the question whether they denote data objects is irrelevant.

We are concerned in this chapter with the use of equations for defining functions and ignore other uses of equations. In such use of equations, there are good grounds for imposing the additional condition that a canonical term not only be irreducible but also denote a data object.

6.4 Constructor Terms as Canonical Terms

We argued in §5.9 that, to reduce the search space for the purpose of term evaluation, it is necessary to test whether a term has a canonical ground instance and whether it has a noncanonical ground instance. We advanced two reasons for not adopting, for the purpose of functional programming, the usual notion of canonicality as irreducibility: the undecidability of canonical terms and the programmer's notion of canonical terms as terms denoting data objects.

In a logic program without equations, the function symbols are used to denote *data constructors*, i.e., functions used to construct composite data objects. The ground terms,

[3] The reference is to [64].

i.e., the members of a Herbrand universe, are used for data structuring and constitute the intended domain of the data objects. Relations over this domain defined by the logic program are computed. Functions are not objects of computation.

In a logic program with equations defining functions, the function symbols are used to denote the data constructors as well as the functions that are computed. Here we have two kinds of function symbols with conspicuously different uses.

Example 6.1 Consider the following logic program defining addition and the function *length* of lists:

$$
\begin{array}{rll}
P = & \{ & 0 + x = x \leftarrow \\
 & , & s(x) + y = s(x + y) \leftarrow \\
 & , & length(nil) = 0 \leftarrow \\
 & , & length(x.y) = s(length(y)) \leftarrow \\
 & \}. &
\end{array}
$$

The intended use of the function symbols '0', 's', 'nil', and '.' is to construct data objects, namely natural numbers and lists, whereas the intended use of '+' and '$length$' is to define the two functions and to evaluate terms. ■

Henceforth we assume that the set of function symbols Σ of a given language is partitioned into the two disjoint decidable sets Σ_C and Σ_D. A function symbol in Σ_C will be called a *constructor symbol*; a function symbol in Σ_D will be called a *defined symbol*. A ground term containing only constructor symbols will be called a *constructor term*. We assume that for all logic programs P, the set of constructor terms is a subset of the set of P-canonical terms. It is intended that constructor symbols are used for construction of data objects and defined symbols for function definitions and evaluation.[4]

An example of the partition [80] is to define a defined symbol to be a function symbol that appears as the outermost function symbol of the left-hand side of the equation in the

[4]For a study of equational theories with such a partition of function symbols, see [37,51].

conclusion of a clause, and a constructor symbol to be one that is not a defined symbol. Clearly all the constructor terms according to this definition are irreducible, if there is no clause of the form $x = t \leftarrow B_1, ..., B_m$, where x is a variable. In Example 6.1, '*length*' and '+' are the defined symbols, and '0', '*s*', '*nil*', and '.' are the constructor symbols.

However, all that need be assumed for the ensuing development in this chapter is that the function symbols are partitioned into the two disjoint decidable sets of constructor symbols and defined symbols, and the set of constructor terms is a subset of the set of the irreducible terms.

The set of irreducible terms thus divides into two classes: constructor terms and non-constructor terms. We regard the constructor terms as denoting data objects and as constituting the intended domain of relations (including equality). To be consistent with the choice of the set of constructor terms as the intended domain, we propose a semantical model in which relations (including equality) can only hold among the terms equal to a constructor term. In this proposed semantical model, relations never hold among the terms not equal to any constructor term even if they are a logical implication of the program.

Example 6.2 Consider the logic program Q below defining the function *length* of lists and the relation *less than or equal to*.

$$
\begin{aligned}
Q = \ \{ \ & length(nil) = 0 \leftarrow \\
, \ & length(x.y) = s(length(y)) \leftarrow \\
, \ & le(0, x) \leftarrow \\
, \ & le(s(x), s(y)) \leftarrow le(x, y) \\
\}. &
\end{aligned}
$$

We assume that '0', '*s*', '*nil*', and '.' are the constructor symbols and '*length*' is the defined symbol. If one asks a question $\leftarrow length(0) = x$, the response is 'yes' according to logic and the answer will probably be $x = length(0)$; $length(0)$ is irreducible and $length(0) = length(0)$ is true. In the proposed semantical model, the answer should be 'no' because

$length(0)$ is not equal to any constructor term. If one asks a question:

$$\leftarrow le(length(0.nil), length(0.0)),$$

the response is 'yes' according to logic; $le(length(0.nil), length(0.0))$ is a logical implication of Q. In the proposed semantical model, the response should be 'no'; the term $length(0.0)$ is not equal to any constructor term. ■

This example may suggest the use of many-sorted logic, restricting, say, the sort of the argument of *length* to lists, thus making '$length(0)$' syntactically illegal. Notice however that our motivation for the proposed semantical model is the undecidability of irreducible terms and the plausibility of constructor terms as canonical terms, not the desire to exclude meaningless terms. The undecidability of irreducible terms arises from the presence of equations with conditions; it persists even if a many-sorted language is used. In fact, Kaplan [60] proved the undecidability in a many-sorted conditional term rewriting system. Thus, our semantical model can be applied to the many-sorted logic programs as well.

We now turn to the exact characterization of the proposed semantical model.

6.5 A Semantical Model Based on Constructor Terms

In the semantical model proposed in this section, all relations including equality can only hold among the terms equal to a constructor term. A positive Horn clause in a logic program:

$$A \leftarrow B_1, ..., B_m$$

is interpreted as saying: for all constructor terms $x_1, ..., x_n$, if $B_1, ..., B_m$ hold then A holds, where $x_1, ..., x_n$ are the variables in the clause. The variables are thought of as ranging over constructor terms only. The logic program itself is therefore a possibly recursive definition of some relations over the constructor terms. How to characterize these relations?

We recall the function F_P associated with a logic program P (Definition 4.6):

$$F_P(I) = cl(T_P(I)).$$

Given $I \subseteq B_P$, $F_P(I)$ is the congruence closure of the ground atoms obtained from I by a one-step application of modus ponens. Let $\cong_{F_P(I)}$ be the congruence relation associated with $F_P(I)$, namely the one defined by:

$$\forall s, t \in U_P, \quad s \cong_{F_P(I)} t \quad \longleftrightarrow \quad s = t \in F_P(I).$$

Within this congruence closure, a given ground term t is equal to a constructor term iff $[t]_{\cong_{F_P(I)}}$ contains a constructor term. If $[t]_{\cong_{F_P(I)}}$ contains no constructor terms, all the terms in it are not equal to any constructor term and, according to our new semantical model, incapable of being an argument of a relation. We therefore restrict the congruence classes induced by $\cong_{F_P(I)}$ to those containing a constructor term.

Formally, we define a new function G_P from Herbrand interpretations to Herbrand interpretations associated with a logic program P.

Definition 6.1 Let P be a logic program. The function G_P associated with P from Herbrand interpretations to Herbrand interpretations is defined as follows. Let I be a Herbrand interpretation.

$$G_P(I) = \{p(t_1, ..., t_n) \in B_P : p(t_1, ..., t_n) \in F_P(I) \text{ and for all } 1 \leq i \leq n, [t_i]_{\cong_{F_P(I)}} \text{ contains a constructor term } \}.$$ ■

As will be shown shortly, the function G_P is continuous. The least fixpoint of G_P therefore exists and

$$lfp(G_P) = \bigcup_{n=0}^{\infty} G_P(\emptyset).$$

It is this least fixpoint of G_P that we regard as the denotation of a logic program P in our new semantical model based on constructor terms.

Let g be the function from congruence closures I to Herbrand interpretations that discards the terms not equal to any constructor term within I:

$g(I) = \{p(t_1, ..., t_n) : p(t_1, ..., t_n) \in I$ and for all $1 \leq i \leq n$, $[t_i]_{\simeq_I}$ contains a constructor term $\}$.

Then $G_P(I) = g(F_P(I))$; G_P is the composition of g and F_P. The continuity of F_P was already proved (Theorem 4.12). Hence to show that G_P is continuous, it suffices to show

Lemma 6.1 *The function g, as defined above, is continuous.*

Proof We have to prove that for all increasing sequences of congruence closures

$$I_0 \subseteq I_1 \subseteq \cdots \subseteq I_n \subseteq I_{n+1} \cdots,$$

$$g(\bigcup_{n=0}^{\infty} I_n) = \bigcup_{n=0}^{\infty} g(I_n).$$

We note that $\bigcup_{n=0}^{\infty} I_n$ is a congruence closure. We also note that for all congruence closures I and for all constructor terms u, $[t]_{\simeq_I}$ contains u iff $t = u \in I$. Let $J = \bigcup_{n=0}^{\infty} I_n$.

($\subseteq$) Suppose that $p(t_1, ..., t_k) \in g(\bigcup_{n=0}^{\infty} I_n)$. Then $p(t_1, ..., t_k) \in \bigcup_{n=0}^{\infty} I_n$ and for all $1 \leq i \leq k$, $[t_i]_{\simeq_J}$ contains a constructor term. Hence there are constructor terms u_i $(1 \leq i \leq k)$ such that $t_i = u_i \in \bigcup_{n=0}^{\infty} I_n$. Let s_i $(1 \leq i \leq k)$ be such constructor terms. For every $1 \leq i \leq k$, there is a finite n_i such that $t_i = s_i \in I_{n_i}$. There is also a finite n_0 such that $p(t_1, ..., t_k) \in I_{n_0}$. Let $m = max_{0 \leq i \leq k} n_i$. Then $t_i = s_i \in I_m$ for all $1 \leq i \leq k$ and $p(t_1, ..., t_k) \in I_m$. Hence $p(t_1, ..., t_k) \in g(I_m)$. Hence $p(t_1, ..., t_k) \in \bigcup_{n=0}^{\infty} g(I_n)$.

($\supseteq$) Suppose that $p(t_1, ..., t_k) \in \bigcup_{n=0}^{\infty} g(I_n)$. Then there is a finite m such that $p(t_1, ..., t_k) \in g(I_m)$. Hence $p(t_1, ..., t_k) \in I_m$ and for all $1 \leq i \leq k$, there are constructor terms u_i such that $t_i = u_i \in I_m$. Hence $p(t_1, ..., t_k) \in \bigcup_{n=0}^{\infty} I_n$ and $t_i = u_i \in \bigcup_{n=0}^{\infty} I_n$ for all $1 \leq i \leq k$. So $p(t_1, ..., t_k) \in g(\bigcup_{n=0}^{\infty} I_n)$. ■

Theorem 6.1 *The function G_P is continuous.*

Proof By the argument preceding Lemma 6.1. ■

As remarked above, we take the least fixpoint of G_P as the denotation of P in the new semantical model.

Theorem 6.2 below shows that the least fixpoint of G_P is a subset of the least E-model of P which contains certain ground atoms whose arguments are all equal to a constructor term.

Lemma 6.2 *Let P be a logic program and $p(t_1, ..., t_n) \in B_P$. For all $m \geq 1$, if $p(t_1, ..., t_n) \in G_P^m(\emptyset)$ then there are constructor terms $u_i (1 \leq i \leq n)$ such that $t_i = u_i \in G_P^m(\emptyset)$.*

Proof Suppose that $p(t_1, ..., t_n) \in G_P^m(\emptyset), m \geq 1$. We note that

$$G_P^m(\emptyset) = G_P(G_P^{m-1}(\emptyset)) = g(F_P(G_P^{m-1}(\emptyset))).$$

So there are constructor terms u_i $(1 \leq i \leq n)$ such that for all $1 \leq i \leq n$, $t_i = u_i \in F_P(G_P^{m-1}(\emptyset))$, hence $t_i = u_i \in g(F_P(G_P^{m-1}(\emptyset))) = G_P^m(\emptyset)$. ■

Theorem 6.2 *Let P be a logic program and $p(t_1, ..., t_k) \in B_P$. If $p(t_1, ..., t_k) \in lfp(G_P)$ then $p(t_1, ..., t_k) \in \bigcap M_E(P)$ and for all $1 \leq i \leq k$, there is a constructor term u_i such that $t_i = u_i \in lfp(G_P)$.*

Proof We recall that $G_P(I) = g(F_P(I))$. Using this definition, it is easy to see that

$$\bigcup_{n=0}^{\infty} G_P^n(\emptyset) \subseteq \bigcup_{n=0}^{\infty} F_P^n(\{t = t : t \in U_P\}),$$

namely that

$$lfp(G_P) \subseteq lfp(F_P) = \bigcap M_E(P).$$

Suppose that $p(t_1, ..., t_k) \in lfp(G_P) = \bigcup_{n=0}^{\infty} G_P^n(\emptyset)$. Then there is a finite $m \geq 1$ such that $p(t_1, ..., t_k) \in G_P^m(\emptyset)$. By Lemma 6.2, there are constructor terms u_i $(1 \leq i \leq k)$ such that for all $1 \leq i \leq k$,

$$t_i = u_i \in G_P^m(\emptyset) \subseteq \bigcup_{n=0}^{\infty} G_P^n(\emptyset) = lfp(G_P). \quad ■$$

The converse of Theorem 6.2 is false, as can be seen by the example below.

Example 6.3 Let P be a logic program:

$$\begin{aligned} P = \ \{ \ & 0 + x = x \leftarrow \\ , \ & s(x) + y = s(x + y) \leftarrow \\ , \ & p(a) \leftarrow a + x = y \\ \}, & \end{aligned}$$

where '0', 's', and 'a' are the constructor symbols and '+' is the defined symbol. $P \models_E p(a)$, 'a' is a constructor term, and $a = a \in lfp(G_P)$. But $p(a) \notin lfp(G_P)$. This is because the premise of the third clause, $a + x = y$, does not have a solution over the constructor terms.

■

This example shows that $lfp(G_P)$ is in general a proper subset of $g(\bigcap M_E(P))$.

6.6 Computing the Least Fixpoint of G_P

In the most general case, computing the least fixpoint of G_P is as intractable as computing $\bigcap M_E(P)$. We therefore again use equations directionally and interpret them operationally as *confluence to a common constructor term*. Requiring the two sides of an equation to reduce to a common constructor term, rather than to just a term, has the effect of discarding the terms not equal to any constructor term.

Let Eq_1'' be the following set of equality axioms.

$$Eq_1'' = \ \{ \ eq_1(x, y) \leftarrow eq_2(x, z), eq_2(y, z) \tag{6.1}$$

$$, \ eq_2(x, x) \leftarrow constructor\_term(x) \tag{6.2}$$

$$, \ eq_2(x, z) \leftarrow eq_3(x, y), eq_2(y, z) \tag{6.3}$$

$$, \ eq_3(x, y) \leftarrow x = y \tag{6.4}$$

$$, \ eq_3(f(x_1, ..., x_i, ..., x_{n_f}), f(x_1, ..., y_i, ..., x_{n_f})) \leftarrow eq_3(x_i, y_i) \tag{6.5}$$

}.

(6.5) is included for all non-constant function symbols f in the language and the i-th argument, for all $1 \leq i \leq n_f$. It is assumed that Eq_1'' includes a possibly infinite number of ground unit clauses for the relation *constructor_term*, enumerating all the constructor terms. Just as the least E-model of a logic program P is computed by the transformed program P_T and the equality axioms Eq_1 (§§5.3 and 5.4), the least fixpoint of G_P is computed by the transformed program P_T and the equality axioms Eq_1''.

Compatibly with the adoption of Eq_1'', the reduction relation associated with a logic program is redefined.

Definition 6.2 For all ground terms M and N,

$$M \rightarrow_P N \overset{def}{\longleftrightarrow} P_T \cup Eq_1'' \models eq_3(M, N). \quad \blacksquare$$

Ground confluence, ground termination, and P-canonicality are defined in the same way as before. In particular, the Church-Rosser property (Proposition 5.1) holds for $\rightarrow_P$ defined above as well.

Theorem 6.3 *Let P be a logic program. Then for all ground terms M and N,*

$$M \overset{*}{\rightarrow}_P N \text{ and } N \text{ is a constructor term} \longleftrightarrow P_T \cup Eq_1'' \models eq_2(M, N).$$

Proof $\longrightarrow$ is proved by induction on the n of $\overset{n}{\rightarrow}$. $\longleftarrow$ is proved by induction on the n of $T_{P_T \cup Eq_1''} \uparrow n$. $\blacksquare$

Theorem 6.4 below is the correctness of $P_T \cup Eq_1''$ with respect to $lfp(G_P)$ for the non-equality relations and eq_1.

Theorem 6.4 *Let P be a logic program. For all ground atoms A other than equations and those of the form $eq_i(M, N)(1 \leq i \leq 3)$, or constructor_term$(M)$,*

$$P_T \cup Eq_1'' \models A \longrightarrow A \in lfp(G_P)$$

and for all ground terms M and N

$$P_T \cup Eq_1'' \models eq_1(M, N) \longrightarrow M = N \in lfp(G_P).$$

Proof It suffices to show that

$$A \in T_{P_T \cup Eq_1''} \uparrow \omega \longrightarrow A \in lfp(G_P)$$

and

$$eq_1(M, N) \in T_{P_T \cup Eq_1''} \uparrow \omega \longrightarrow M = N \in lfp(G_P).$$

We prove this by induction on the n of $T_{P_T \cup Eq_1''} \uparrow n$.

basis: Other than the equations and the atoms of the form *constructor_term*(M), only a 0-place predicate p appearing in a unit clause of P, and hence of P_T, can be a member of $T_{P_T \cup Eq_1''} \uparrow 1$. Clearly $p \in g(F_P(\emptyset)) = G_P(\emptyset) = G_P \uparrow 1$.

induction step: Consider first the case of the ground atoms other than the equations, $eq_i(M, N)$ $(1 \leq i \leq 3)$, and *constructor_term*(M). Suppose that $p(t_1, ..., t_n) \in T_{P_T \cup Eq_1''} \uparrow (m+1)$. There is a ground instance:

$$p(t_1, ..., t_n) \leftarrow eq_1(t_1, u_1), ..., eq_1(t_n, u_n), B_1^T, ..., B_k^T$$

of a clause in P_T such that $eq_1(t_i, u_i) \in T_{P_T \cup Eq_1''} \uparrow m$ and $B_j^T \in T_{P_T \cup Eq_1''} \uparrow m$ where B_j^T is the transformed atom of B_j. By the induction hypothesis,

$$t_i = u_i \in G_P \uparrow \omega \tag{6.6}$$

and $B_j \in G_P \uparrow \omega$. By Theorem 6.2, there are constructor terms s_i, $1 \leq i \leq n$, such that for all $1 \leq i \leq n$,

$$t_i = s_i \in G_P \uparrow \omega. \tag{6.7}$$

Since

$$p(u_1, ..., u_n) \leftarrow B_1, ..., B_k$$

is a ground instance of a clause in P,

$$p(u_1, ..., u_n) \in T_P(G_P \uparrow \omega). \tag{6.8}$$

We note that

$$G_P \uparrow \omega = G_P(G_P \uparrow \omega) = g(F_P(G_P \uparrow \omega)) \subseteq F_P(G_P \uparrow \omega) = cl(T_P(G_P \uparrow \omega)). \tag{6.9}$$

By (6.8),

$$p(u_1, ..., u_n) \in cl(T_P(G_P \uparrow \omega)). \tag{6.10}$$

By (6.6) and (6.9),

$$t_i = u_i \in cl(T_P(G_P \uparrow \omega)). \tag{6.11}$$

By (6.10) and (6.11),

$$p(t_1, ..., t_n) \in cl(T_P(G_P \uparrow \omega)). \tag{6.12}$$

By (6.7) and (6.9),

$$t_i = s_i \in cl(T_P(G_P \uparrow \omega)). \tag{6.13}$$

By (6.12) and (6.13),

$$p(t_1, ..., t_n) \in g(cl(T_P(G_P \uparrow \omega))) = G_P(G_P \uparrow \omega) = G_P \uparrow \omega. \tag{6.14}$$

Consider next the case of $eq_1(M, N)$. Suppose that $eq_1(M, N) \in T_{P_T \cup Eq_1''} \uparrow (m+1)$. Because of (6.1) and Theorem 6.3, there must be a constructor term L and two reduction sequences:

$$M \equiv M_0 \rightarrow M_1 \rightarrow \cdots \rightarrow M_n \equiv L,$$

$$N \equiv N_0 \rightarrow N_1 \rightarrow \cdots \rightarrow N_{n'} \equiv L$$

such that for all $0 \leq i \leq n-1$, there are terms s and t such that $M_{i+1} \equiv M_i[s \leftarrow t]$ and $s = t \in T_{P_T \cup Eq_1''} \uparrow k$ for some $k \leq m$, and for all $0 \leq i \leq n'-1$, there are terms u and v such that $N_{i+1} \equiv N_i[u \leftarrow v]$ and $u = v \in T_{P_T \cup Eq_1''} \uparrow l$ for some $l \leq m$. Moreover, for each such $s = t$, there is a ground instance:

$$s = t \leftarrow B_1^T, ..., B_q^T$$

of a clause in P_T such that $B_j^T \in T_{P_T \cup Eq_1''} \uparrow k$ for some $k \leq m$ where B_j^T is the transformed atom of B_j. By the induction hypothesis, $B_j \in G_P \uparrow \omega$. Hence $s = t \in T_P(G_P \uparrow \omega)$. Similarly $u = v \in T_P(G_P \uparrow \omega)$. So $M_i = M_{i+1} \in cl(T_P(G_P \uparrow \omega))$ for all $0 \leq i \leq n-1$ and $N_i = N_{i+1} \in cl(T_P(G_P \uparrow \omega))$ for all $0 \leq i \leq n'-1$. So

$$M = L \in cl(T_P(G_P \uparrow \omega)) = F_P(G_P \uparrow \omega)$$

and

$$N = L \in cl(T_P(G_P \uparrow \omega)) = F_P(G_P \uparrow \omega).$$

As L is a constructor term, $M = N \in g(F_P(G_P \uparrow \omega)) = G_P(G_P \uparrow \omega) = G_P \uparrow \omega$. This completes the induction step.

Thus the theorem holds. ■

Theorem 6.5 below is the completeness under ground confluence of $P_T \cup Eq_1''$ with respect to $lfp(G_P)$ for ground atoms.

Theorem 6.5 *Let P be a ground confluent logic program. For all non-equational ground atoms A,*

$$A \in lfp(G_P) \longrightarrow P_T \cup Eq_1'' \models A$$

and for all ground terms M and N,

$$M = N \in lfp(G_P) \longrightarrow P_T \cup Eq_1'' \models eq_1(M, N).$$

Proof It suffices to that

$$A \in G_P \uparrow \omega \longrightarrow P_T \cup Eq_1'' \models A$$

and

$$M = N \in G_P \uparrow \omega \longrightarrow P_T \cup Eq_1'' \models eq_1(M, N)$$

We prove this by induction on the n of $G_P \uparrow n$.

basis: The case $n = 0$ is vacuously true.

induction step: Consider first the equations. Suppose that $M = N \in G_P \uparrow (m+1)$. By Lemma 6.2, there is a constructor term u such that $M = u \in G_P \uparrow (m+1)$. Let u be such a constructor term. Clearly $N = u \in G_P \uparrow (m+1)$ as well. We note that

$$G_P \uparrow (m+1) = G_P(G_P \uparrow m) = g(F_P(G_P \uparrow m)) \subseteq F_P(G_P \uparrow m) = cl(T_P(G_P \uparrow m)). \quad (6.15)$$

So by Lemma 5.2, there are ground terms

$$M \equiv K_0, ..., K_k \equiv u, \quad k \geq 0,$$

such that for every pair of K_i and K_{i+1}, $0 \leq i \leq k-1$, $K_{i+1} \equiv K_i[s \leftarrow t]$ and either $s = t \in T_P(G_P \uparrow m)$ or $t = s \in T_P(G_P \uparrow m)$ for some ground terms s and t. Suppose that $s = t \in T_P(G_P \uparrow m)$. Then there is a ground instance:

$$s = t \leftarrow B_1, ..., B_l$$

of a clause in P such that $\{B_1, ..., B_l\} \subseteq G_P \uparrow m$. By the induction hypothesis, $P_T \cup Eq_1'' \models B_i^T$ for all $1 \leq i \leq l$ where B_i^T is the transformed atom of B_i. Hence $P_T \cup Eq_1'' \models s = t$. So by (6.4), $P_T \cup Eq_1'' \models eq_3(s,t)$. So by (6.5), $P_T \cup Eq_1'' \models eq_3(K_i, K_{i+1})$. Using a similar argument, we see that $t = s \in T_P(G_P \uparrow m)$ implies $P_T \cup Eq_1'' \models eq_3(K_{i+1}, K_i)$.

So the terms K_i's are such that $P_T \cup Eq_1'' \models eq_3(K_i, K_{i+1})$ or $P_T \cup Eq_1'' \models eq_3(K_{i+1}, K_i)$. Hence by Definition 6.2, the terms are such that $K_i \rightarrow K_{i+1}$ or $K_{i+1} \rightarrow K_i$. By the ground confluence of P and the Church-Rosser property, there is a ground term v such that $M \xrightarrow{*} v$ and $u \xrightarrow{*} v$. Using a similar argument, we see that $N = u \in G_P \uparrow (m+1)$ implies the existence of a ground term w such that $N \xrightarrow{*} w$ and $u \xrightarrow{*} w$. By the ground confluence of P, there is a ground term x such that $v \xrightarrow{*} x$ and $w \xrightarrow{*} x$. As u is a constructor term and hence is canonical, u and x must be the same. Hence $M \xrightarrow{*} u$ and $N \xrightarrow{*} u$. So by Theorem 6.3, $P_T \cup Eq_1'' \models eq_2(M, u)$ and $P_T \cup Eq_1'' \models eq_2(N, u)$. So by (6.1), $P_T \cup Eq_1'' \models eq_1(M, N)$.

Consider next non-equational atoms. Suppose that $p(t_1, ..., t_n) \in G_P \uparrow (m+1)$. By (6.15), there is a ground instance:

$$p(u_1, ..., u_n) \leftarrow B_1, ..., B_l \quad (6.16)$$

of a clause in P such that $t_i = u_i \in G_P \uparrow (m+1)$ and $\{B_1, ..., B_l\} \subseteq G_P \uparrow m$. By the proof above for the case of equations, $P_T \cup Eq_1'' \models eq_1(t_i, u_i)$. By the induction hypothesis, $P_T \cup Eq_1'' \models B_i^T$ for all $1 \leq i \leq l$ where B_i^T is the transformed atom of B_i. The ground clause:

$$p(t_1, ..., t_n) \leftarrow eq_1(t_1, u_1), ..., eq_1(t_n, u_n), B_1^T, ..., B_l^T$$

is a ground instance of a clause in P_T, since (6.16) is a ground instance of a clause in P. Hence $P_T \cup Eq_1'' \models p(t_1, ..., t_n)$. This completes the induction step.

Thus the theorem holds. ■

6.7 SLDCT-Resolution

In the previous section, we assumed that Eq_1'' includes a possibly infinite number of ground unit clauses for the relation *constructor_term*, enumerating all the constructor terms. This assumption is unsatisfactory in practice for two reasons:

1. it is unrealistic to use such an enumeration of ground unit clauses, even if the enumeration happens to be finite,
2. when a goal of the form $\leftarrow$ *constructor_term*(t), where t contains variables, has been resolved, all the variables in t are instantiated to some ground terms, hence failing to yield the most general answer for a question.

It is therefore necessary to use a refutation procedure that gives the effect of the predicate '*constructor_term*' and ensures the most general answer for any questions, instead of an enumeration of ground unit clauses for the relation *constructor_term*.

For this purpose, we propose *SLD-resolution with the constructor term test* (hereafter called *SLDCT-resolution*): SLD-resolution with the ability to test, given a term t, whether there is a ground instance of t that is a constructor term. The test will be done only when an atom of the form $eq_2(s, t)$ has been resolved.

The function of SLDCT-resolution is best defined by means of Eq_1''. In SLDCT-resolution the following function is performed when a goal of the form $\leftarrow$ *constructor_term*(t) is to be resolved: *constructor_term*(t) succeeds iff there is a ground instance of t that is a constructor term, without instantiating t, and when it has succeeded all the variables in t are *restricted*. An SLDCT-derivation (SLDCT-refutation) is defined as an SLD-derivation (SLD-refutation) using Eq_1'' in which

1. a goal of the form $\leftarrow$ *constructor_term*(M) is resolved according to the rule above,
2. only a unifier $\theta = \{x_1/t_1, ..., x_n/t_n\}$ may be used such that for all $1 \leq i \leq n$, if x_i is restricted then t_i must not contain any defined symbols,
3. if a unifier $\theta = \{x_1/t_1, ..., x_n/t_n\}$ is used and x_i is restricted, then all the variables in t_i are restricted, for all $1 \leq i \leq n$.

The restricting of variables and the restriction of unification are needed to ensure the correctness of computed answer substitutions with respect to $lfp(G_P)$. An SLDCT-refutation procedure is an SLD-refutation procedure that finds an SLDCT-refutation using Eq_1''. By an SLD-derivation (SLD-refutation) of $P_T \cup Eq_1'' \cup \{G\}$, we mean one for which Eq_1'' is used with a possibly infinite enumeration of ground unit clauses for the relation *constructor_term*, and distinguish it from an SLDCT-derivation (SLD-refutation) defined above.

One motivation for our new semantical model was the undecidability of canonical (i.e., irreducible) terms. Theorem 6.6 below shows that the problem that the predicate *constructor_term*(M) has to solve is decidable.

Theorem 6.6 *Suppose that there is at least one constructor constant symbol in the language. Let t be any term. Then there is a ground instance of t that is a constructor term iff t contains no defined symbols.*

Proof Straightforward. ∎

For any atom A, let $[A]$ be the set of all ground atoms obtained by instantiating the variables in A to some *constructor terms*. A ground substitution $\{x_1/t_1, ..., x_n/t_n\}$ will be called a *constructor substitution* if t_i is a constructor term for all $1 \leq i \leq n$. Let P be a logic program and G a goal clause $\leftarrow B_1, ..., B_m$. Let B_i^T be the transformed atom of B_i. Since in our new semantical model, all the variables in a clause range over constructor terms, the correctness of SLDCT-resolution with respect to $lfp(G_P)$ consists in the fact that $[B_i\theta] \subseteq lfp(G_P)$ for all $1 \leq i \leq m$, where θ is any computed answer substitution obtained from any SLDCT-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow B_1^T, ..., B_m^T\}$. For the same reason, the completeness of SLDCT-resolution with respect to $lfp(G_P)$ consists in the fact that for any substitutions θ such that $[B_i\theta] \subseteq lfp(G_P)$ for all $1 \leq i \leq m$, there is an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow B_1^T, ..., B_m^T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ. We now proceed to prove these.

Theorem 6.7 *Let P be a logic program. Let θ be a computed answer substitution obtained from an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{G\}$, where G is a goal clause $\leftarrow B_1, ..., B_m$ that may contain any predicates in $P_T \cup Eq_1''$. Then $P_T \cup Eq_1'' \models (B_1, ..., B_m)\theta\sigma$ for all constructor substitutions σ for the variables of $G\theta$.*

Proof By induction on the length of an SLDCT-refutation.

basis: Suppose that the length of an SLDCT-refutation is one. Then G is of the form $\leftarrow p$, $\leftarrow M = N$, or $\leftarrow constructor\_term(M)$, where p is a 0-place predicate, M and N are some terms. In the first two cases, the theorem clearly holds. In the last case, the theorem holds by Theorem 6.6.

induction step: Suppose that the theorem holds for SLDCT-refutations of length less than $m + 1$. Consider an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{G\}$ of length $m + 1$. Let G be $\leftarrow B_1, ..., B_m$ and B_i be the selected atom.

case 1: B_i is not of the form $constructor\_term(M)$. Let

$$A \leftarrow C_1, ..., C_k$$

be the first input clause and σ be a unifier of B_i and A. The first derived clause is:

$$\leftarrow (B_1, ..., B_{i-1}, C_1, ..., C_k, B_{i+1}, ..., B_m)\sigma.$$

Let γ be the computed answer substitution for this goal. By the induction hypothesis, for all constructor substitutions δ,

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, C_1, ..., C_k, B_{i+1}, ..., B_m)\sigma\gamma\delta.$$

Note that $\sigma\gamma$ is the computed answer substitution for the original goal G. Let δ_0 be any constructor substitution for $G\sigma\gamma$, namely for

$$\leftarrow (B_1, ..., B_{i-1}, B_i, B_{i+1}, ..., B_m)\sigma\gamma.$$

Clearly $B_i\sigma\gamma\delta_0 \equiv A\sigma\gamma\delta_0$ and the goal

$$\leftarrow (B_1, ..., B_{i-1}, B_i, B_{i+1}, ..., B_m)\sigma\gamma\delta_0$$

derives

$$\leftarrow (B_1, ..., B_{i-1}, C_1, ..., C_k, B_{i+1}, ..., B_m)\sigma\gamma\delta_0. \tag{6.17}$$

As seen above (by extending δ_0 so that (6.17) becomes ground if it is not ground),

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, C_1, ..., C_k, B_{i+1}, ..., B_m)\sigma\gamma\delta_0.$$

Hence

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, B_i, B_{i+1}, ..., B_m)\sigma\gamma\delta_0.$$

case 2: B_i is of the form $constructor\_term(M)$. Let $x_1, ..., x_n$ be all the variables in M. All x_i's are restricted. The first derived clause is:

$$\leftarrow B_1, ..., B_{i-1}, B_{i+1}, ..., B_m.$$

Let σ be the computed answer substitution for this goal, and hence for the original goal G. By the induction hypothesis, for all constructor substitutions γ,

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, B_{i+1}, ..., B_m)\sigma\gamma.$$

Since all the x_i's are restricted, σ can only instantiate x_i's to terms containing no defined symbols. Hence for any constructor substitution γ_0 for $G\sigma$, $\sigma\gamma_0$ can only instantiate x_i's to some constructor terms. So in the goal

$$\leftarrow (B_1, ..., B_{i-1}, constructor\_term(M), B_{i+1}, ..., B_m)\sigma\gamma_0,$$

$M\sigma\gamma_0$ is a constructor term and hence $constructor\_term(M\sigma\gamma_0)$ unifies with one of the enumerated unit clauses and derives

$$\leftarrow (B_1, ..., B_{i-1}, B_{i+1}, ..., B_m)\sigma\gamma_0.$$

As seen above,

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, B_{i+1}, ..., B_m)\sigma\gamma_0.$$

So

$$P_T \cup Eq_1'' \models (B_1, ..., B_{i-1}, constructor\_term(M), B_{i+1}, ..., B_m)\sigma\gamma_0. \quad \blacksquare$$

Theorem 6.8 (Correctness of SLDCT-resolution with respect to lfp(G_P)) *Let P be a logic program and G a goal clause $\leftarrow B_1, ..., B_m$. Let θ be a computed answer substitution obtained from an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow B_1^T, ..., B_m^T\}$ where B_i^T is the transformed atom of B_i. Then for all $1 \leq i \leq m$, $[B_i\theta] \subseteq lfp(G_P)$.*

Proof Let σ be any constructor substitution for $B_i\theta$. Clearly, σ is a constructor substitution for $B_i^T\theta$. We can extend σ to a constructor substitution σ' for $\leftarrow (B_1^T, ..., B_m^T)\theta$. By Theorem 6.7, $P_T \cup Eq_1'' \models (B_1^T, ..., B_m^T)\theta\sigma'$. So $P_T \cup Eq_1'' \models B_i^T\theta\sigma$. So by Theorem 6.4, $B_i\theta\sigma \in lfp(G_P)$. $\blacksquare$

Next we prove the completeness under ground confluence of SLDCT-resolution with respect to $lfp(G_P)$.

Theorem 6.9 *Let P be a ground confluent logic program. Suppose that $[A] \subseteq lfp(G_P)$ where A is an atom other than an equation. Then there is an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow A\}$ with the identity substitution as the computed answer substitution. Suppose*

that $[M = N] \subseteq lfp(G_P)$. *Then there is an SLDCT-refutation of* $P_T \cup Eq_1'' \cup \{\leftarrow eq_1(M, N)\}$ *with the identity answer substitution as the computed answer substitution.*

Proof Let $x_1, ..., x_n$ be all the variables occurring in A. Let $a_1, ..., a_n$ be some distinct constructor constant symbols not occurring in P or A. Let $\theta = \{x_1/a_1, ..., x_n/a_n\}$. Since $[A] \subseteq lfp(G_P)$, $A\theta \in lfp(G_P)$. Hence by Theorem 6.5, there is an SLD-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow A\theta\}$. This SLD-refutation can be modified into an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{\leftarrow A\}$ with the identity substitution as the computed answer substitution by replacing a_i's by x_i's, because a_i's do not occur in P_T or A and a goal $\leftarrow$ *constructor_term*(M) succeeds in SLDCT-resolution after the replacement if it succeeds in SLD-resolution before replacement. The case for equations is proved similarly. ■

In the proof of Theorem 6.10 below, we use the lifting lemma for SLDCT-refutations which is easily seen to be true: if there is an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{G_T\theta\}$, then there is one of $P_T \cup Eq_1'' \cup \{G_T\}$.

Theorem 6.10 (Completeness of SLDCT-resolution with respect to lfp(G_P)) *Let* P *be a ground confluent logic program and* G *a goal clause* $\leftarrow B_1, ..., B_m$. *Let* θ *be any substitution for the variables of* G *such that* $[B_i\theta] \subseteq lfp(G_P)$ *for all* $1 \leq i \leq m$. *Then there is an SLDCT-refutation of* $P_T \cup Eq_1'' \cup \{G_T\}$ *with the computed answer substitution* σ *such that* $\theta = \sigma\gamma$ *for some substitution* γ.

Proof By Theorem 6.9, there is an SLDCT-refutation for $P_T \cup Eq_1'' \cup \{\leftarrow B_i^T\theta\}$ with the identity substitution as the computed answer substitution for every $1 \leq i \leq m$. These refutations can be combined into an SLDCT-refutation of $P_T \cup Eq_1'' \cup \{G_T\theta\}$ with the identity substitution as the computed answer substitution. By the lifting lemma for SLDCT-refutations, there is an SLDCT-refutation for $P_T \cup Eq_1'' \cup \{\leftarrow G_T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$. ■

Chapter 7

Introduction of Higher-Order Functions

7.1 Introduction

In the last two chapters we have shown how functions can be incorporated into logic programming through the use of equality axioms. Specifically, the functions considered there were first-order functions. Can we introduce higher-order functions—functions that admit functions as arguments or values—within the framework of logic programming with equality? In this chapter we provide an affirmative answer by proposing *logic programming with the Curried Herbrand Universe*.

Higher-order functions are an important tool of most modern functional programming languages [3,10,78,94,97,99]. The designers of these languages recognized the power of abstraction engendered by being able to program with higher-order functions and embodied them in their languages. In the functional language community, the utility of higher-order functions is widely acknowledged, although Goguen and Meseguer [33,40,38] and O'Donnell [48,80] prefer to avoid them.

There are two approaches to incorporating higher-order functions into logic programming: radical and conservative. A radical approach abandons first-order logic altogether and adopts a form of higher-order logic [77]. In a conservative approach, one stays within first-order logic and defines higher-order functions using formulas of first-order logic [80,101].

The proposal put forward in this chapter takes an almost conservative approach (the sense of the qualification "almost" will be explained soon) because

1. unification in logic of the order greater than one is undecidable [43,50],
2. we want to use the theory developed in the previous chapters and Prolog implementation techniques as much as possible.

More specifically, we take the idea suggested by O'Donnell [80] (but already implicit in work by Church and Rosser [13]), viz., defining higher-order functions using first-order equations adopting the Schönfinkel-Curry function application operator as a binary functor.

A possible criticism of conservative approaches in favor of radical ones is that defining higher-order functions using first-order sentences is merely "encoding" or "simulating" higher-order functions, depriving them of semantic content. We will show that it is possible to incorporate equationally defined higher-order functions into logic programs with equality based on a formal semantics in terms of a *functional domain* [76]—a domain suitable for models of λ-calculus and combinatory logic—constructed from the domain of the least E-model of a logic program with equality. We depart from first-order logic in constructing the functional domain to interpret every ground term in a logic program as representing a unary function: this is why we call our approach almost conservative. Yet computation is accomplished by first-order deduction, avoiding the computational intractability incurred by the use of higher-order logic.

7.2 Logic Programming with the Curried Herbrand Universe

We propose *logic programming with the Curried Herbrand universe*, hereafter *LPCH*, introducing higher-order functions into logic programming with equality. In LPCH, we adopt the Schönfinkel-Curry function application operator [19,90] as a binary functor and define functions by first-order equations.

As far as syntax goes, a program in LPCH is a set of positive Horn clauses with equality. All that is needed is to redefine terms.

Definition 7.1 The terms in LPCH is defined as follows.

1. a variable is a term,
2. a constant symbol is a term,
3. if t and u are terms, so is $t : u$. ■

':' is an infix left-associative binary function symbol representing the function application operator; $t : u$ is the result of applying t to u. Parentheses may be used for grouping. The Herbrand universe of a program in LPCH therefore consists of the ground terms built up solely of the function application operator ':' and constant symbols. We shall assume that at least one constant symbol appears in every LPCH program.

Let P be an LPCH program and $M = \bigcap M_E(P)$. Let $\cong_M$ be the congruence relation over U_P associated with M:

$$\forall t, u \in U_P, \;\; t \cong_M u \longleftrightarrow P \models_E t = u.$$

Let $[t]$ be the congruence class of t under $\cong_M$. In the least E-model M, a constant symbol c is assigned the denotation $[c]$ and the function application operator ':' is assigned the binary function $:^M$ such that

$$\forall [t], [u] \in U_P/\cong_M, \;\; [t] :^M [u] = [t : u].$$

The least E-model assigns to ':' the denotation conforming to the intended interpretation of the function application operator. However, a ground term t receives the denotation $[t]$, which is just a congruence class. We see this as a drawback of the least E-model in giving meaning to the ground terms of an LPCH program, since each ground term in Curried notation is meant to represent a unary function that takes as argument a unary function and returns as value a unary function.

Consider, for example, the following LPCH program P.[1]

$$
\begin{array}{rll}
P = & \{ & +:0:X = X \leftarrow \\
 & , & +:(s:X):Y = s:(+:X:Y) \leftarrow \\
 & , & twice:F:X = F:(F:X) \leftarrow \\
 & \}. &
\end{array}
$$

'+' is supposed to represent a unary function which takes as argument a natural number (in fact a unary function) n and returns the unary function "*plus n*"; '*twice*' is supposed to represent a unary function which takes as argument a unary function f and returns the composition of f with itself. The least E-model $\bigcap M_E(P)$ fails to specify what unary function an object in the universe $U_P/\cong_M$ represents.

To remedy this unsatisfactory situation, we construct from $U_P/\cong_M$ a *functional domain* [76]—a domain suitable for interpretation of Curried terms—over which the ground terms of an LPCH program will be interpreted.

7.3 Functional Domain and Extensionality

This section gives a description of functional domains [76]. All the materials in this section are taken from [76] and Chapters 10 and 11 of [47].

A *functional domain* is a quadruple $\mathcal{U} = \langle D, D \rightarrow D, Fun, Rep \rangle$ where

[1]Henceforth, identifiers beginning with a capital letter will be used as variables.

1. D is a nonempty set,

2. $D \to D$ is a set of functions from D to D,

3. *Fun* is a surjective function from D onto $D \to D$,

4. *Rep* is a function from $D \to D$ to D such that $f = Fun(Rep(f))$ for all $f \in D \to D$.

Pictorially,

$$D \underset{Rep}{\overset{Fun}{\rightleftarrows}} D \to D.$$

Note that the existence of *Rep* as specified in (4) is guaranteed by the surjectivity of *Fun* and every such *Rep* is necessarily *injective*. *Fun* need not be injective.

Definition 7.2 Let $\mathcal{U} = \langle D, D \to D, Fun, Rep \rangle$ be a functional domain and $f \in D \to D$. A member $d \in D$ such that $Fun(d) = f$ is called a *representative* of f. The set of all representatives of a function $f \in D \to D$ is denoted $Reps(f)$. ■

Clearly, $Rep(f) \in Reps(f)$ for all $f \in D \to D$. A representative of a function $f \in D \to D$ is also called a *name, notation, index*, or *quotation* of f. In [47,76], the members of D are used as values of combinators and λ-expressions, each value v representing a function $Fun(v) \in D \to D$ and each function $f \in D \to D$ being represented by at least one member of D, namely $Rep(f)$, so that the standard laws of identity in combinatory logic and λ-calculus hold in $\mathcal{U}$.

Definition 7.3 Let $\mathcal{U} = \langle D, D \to D, Fun, Rep \rangle$ be a functional domain. For all $d_1, d_2 \in D$, d_1 *is extensionally equivalent to* d_2, written $d_1 \sim d_2$, iff $Fun(d_1) = Fun(d_2)$. ■

It is straightforward to see that $\sim$ is an equivalence relation over D. An equivalence class of $d \in D$ will be denoted by $[d]_\sim$.

Proposition 7.1 *Let $\mathcal{U} = \langle D, D \to D, Fun, Rep\rangle$ be a functional domain. For all $f \in D \to D$, $Reps(f) = [Rep(f)]_\sim$.* ■

Definition 7.4 A functional domain $\mathcal{U} = \langle D, D \to D, Fun, Rep\rangle$ is *extensional* iff for all $d_1, d_2 \in D, Fun(d_1) = Fun(d_2)$ implies $d_1 = d_2$, namely iff *Fun* is injective. ■

Proposition 7.2 *The extensionality of a functional domain $\mathcal{U} = \langle D, D \to D, Fun, Rep\rangle$ is equivalent to each of the following.*

1. $\forall d_1, d_2 \in D,\ d_1 \sim d_2 \longrightarrow d_1 = d_2$,
2. $\forall d \in D,\ [d]_\sim$ *is a singleton set,*
3. $\forall f \in D \to D,\ Reps(f)$ *is a singleton set,*
4. $\forall d \in D,\ Rep(Fun(d)) = d$. ■

Thus in an extensional functional domain, *Fun* is a bijection from D onto $D \to D$, *Rep* is a bijection from $D \to D$ onto D, and each is the inverse of the other. *Given* a functional domain $\mathcal{U} = \langle D, D \to D, Fun, Rep\rangle$, we can *define* the binary function application operation ":" on D by

$$\forall d_1, d_2 \in D,\ \ d_1 : d_2 = (Fun(d_1))(d_2).$$

Conversely, *given* a binary operation ":" on a nonempty set D, we can *construct* a functional domain $\mathcal{U} = \langle D, D \to D, Fun, Rep\rangle$ by defining, for every $d \in D$, the function f_d from D to D:

$$\forall d_1 \in D,\ \ f_d(d_1) = d : d_1$$

and

$$\begin{aligned}
&D \to D \ = \ \{f_d : d \in D\},\\
&Fun(d) = f_d, \quad \forall d \in D,\\
&Rep(f_d) = d_0 \text{ for some } d_0 \in D \text{ such that } f_{d_0} = f_d.
\end{aligned}$$

In either case, we have

$$\forall d_1, d_2 \in D, \ \ (Fun(d_1))(d_2) = d_1 : d_2,$$

showing that it makes no difference whether we apply two objects d_1 and d_2 $(d_1 : d_2)$ or apply the function represented by d_1 to d_2 $((Fun(d_1))(d_2))$. Since in a functional domain $\mathcal{U} = \langle D, D \rightarrow D, Fun, Rep \rangle$ the function Fun is surjective, the cardinality of $D \rightarrow D$ is at most that of D, hence $D \rightarrow D$ excludes many functions from D to D. For example, the following "paradoxical" diagonal function cannot exist in $D \rightarrow D$.[2]

Let d_0 be an arbitrary member of D. Let p be a function from D to D such that for all $d \in D$,

$p(d) = d_0$ if $(Fun(d))(d) \neq d_0$, and
$p(d) \neq d_0$ if $(Fun(d))(d) = d_0$.

Suppose that p exists in $D \rightarrow D$. Then $p = Fun(d_1)$ for some $d_1 \in D$. Taking in particular d to be d_1, the two clauses above lead to a contradiction. Hence p cannot exist in $D \rightarrow D$.

7.4 Models of LPCH Programs Based on Functional Domains

We construct a functional domain from the domain of the least E-model of an LPCH program. This functional domain will then allow us to interpret every ground term in an LPCH program as representing a unary function. Let P be an LPCH program, $M = \bigcap M_E(P)$, $\cong_M$ the congruence relation associated with M, and $:^M$ the binary function assigned to ':' in M. Since $:^M$ is a binary function on $U_P/\cong_M$, the desired construction can be done in the manner explained in the previous section.

[2] I thank Romas Aleliunas for calling my attention to this point.

For each congruence class $[t] \in U_P/\cong_M$, we define the function $f_{[t]}$ from $U_P/\cong_M$ to $U_P/\cong_M$ by $f_{[t]}([u]) = [t] :^M [u] = [t : u]$ for all $[u] \in U_P/\cong_M$.

For an LPCH program P, let $\mathcal{U}_P = \langle U_P/\cong_M, U_P/\cong_M \rightarrow U_P/\cong_M, Fun, Rep\rangle$ where

1. $U_P/\cong_M \rightarrow U_P/\cong_M \;=\; \{f_{[t]} : [t] \in U_P/\cong_M\}$,
2. $Fun([t]) = f_{[t]}$, for all $[t] \in U_P/\cong_M$,
3. $Rep(f_{[t]}) = [u_0]$ for some $[u_0]$ such that $f_{[t]} = f_{[u_0]}$.

We prove that $\mathcal{U}_P$ is a functional domain.

Theorem 7.1 *Let P be any LPCH program. $\mathcal{U}_P$ is a functional domain.*

Proof Clearly $U_P/\cong_M$ is nonempty, Fun is a surjective function from $U_P/\cong_M$ onto $U_P/\cong_M \rightarrow U_P/\cong_M$, and Rep is a function from $U_P/\cong_M \rightarrow U_P/\cong_M$ to $U_P/\cong_M$. To see that $Fun(Rep(f)) = f$ for all $f \in U_P/\cong_M \rightarrow U_P/\cong_M$, let $f_{[t]} \in U_P/\cong_M \rightarrow U_P/\cong_M$. Then $Rep(f_{[t]}) = [u_0]$ for some $[u_0]$ such that $f_{[t]} = f_{[u_0]}$. Thus,

$$Fun(Rep(f_{[t]}) = Fun([u_0]) = f_{[u_0]} = f_{[t]}. \quad \blacksquare$$

Example 7.1 Consider the following *LPCH* program.

$$\begin{aligned} P = \;\{ \;& +:0:X = X \leftarrow \\ ,\;& +:(s:X):Y = s:(+:X:Y) \leftarrow \\ ,\;& twice:F:X = F:(F:X) \leftarrow \\ \}. & \end{aligned}$$

The "*plus* 2" function $f_{[+:(s:(s:0))]}$ has three representatives:

$$Reps(f_{[+:(s:(s:0))]}) = \{[+:(s:(s:0))], [twice:s], [twice:(+:(s:0))]\}.$$

Any one of $Reps(f_{[+:(s:(s:0))]})$ can be assigned as $Rep(f_{[+:(s:(s:0))]})$. $\blacksquare$

By taking the functional domain $\mathcal{U}_P$ as the domain of an LPCH program P, it is now possible to interpret a ground term t in P as representing a unary function: t represents the function $Fun([t]) = f_{[t]}$. Thus, the denotation of a ground term is determined as follows:

1. a constant symbol c denotes $[c]$,

2. a ground term $t : u$ denotes $(Fun([t]))([u]) = f_{[t]}([u])$.

In the LPCH program P in Example 7.1, the term '*twice*', e.g., denotes $[twice]$, which represents the function $Fun([twice]) = f_{[twice]}$. Hence the denotation of '$twice : (+ : (s : 0))$' is $f_{[twice]}([+ : (s : 0)]) = [twice : (+ : (s : 0))]$, which in turn represents the function $Fun([twice : (+ : (s : 0))]) = f_{[twice:(+:(s:0))]}$. Hence the denotation of '$twice : (+ : (s : 0)) : 0$' is $f_{[twice:(+:(s:0))]}([0]) = [twice : (+ : (s : 0)) : 0] = [s : (s : 0)]$.

Since by definition,

$$(Fun([t]))([u]) = f_{[t]}([u]) = [t] :^M [u],$$

for all ground terms t and u, the interpretation of $t : u$ as the application of the function $Fun([t])$ to $[u]$ is consistent with the interpretation of $t : u$ as the application of two objects $[t]$ and $[u]$ by $:^M$.

Example 7.1 shows that the functional domain $\mathcal{U}_P$ of an LPCH program P is in general *intensional*: $Reps(f_{[+:(s:(s:0))]})$ has three members, which are all extensionally equivalent to one another representing the same function $f_{[+:(s:(s:0))]}$. The equality of the ground terms determined by $\bigcap M_E(P)$, namely $=$, is the equality of the representatives and not of the represented functions, namely $\sim$. Again in Example 7.1, $[twice : s]$ and $[+ : (s : (s : 0))]$ represent the same function, hence $[twice : s] \sim [+ : (s : (s : 0))]$, but $P \not\models_E (twice : s = + : (s : (s : 0)))$.

7.5 Computing with LPCH Programs

The intensional equality of the terms in an LPCH program, i.e., the equality of representatives rather than of represented functions, is merely the equality determined by logical implication of a program. Hence if '=' in a program is interpreted as the intensional equality, computing an LPCH program P is computing the least E-model $\bigcap M_E(P)$. Since the function application operator is represented by the binary function symbol ':', the standard equality axioms Eq with the substitutivity axioms for ':', when adjoined to an LPCH program P, will compute $\bigcap M_E(P)$. If in particular a program is ground confluence, Eq_1 and Eq_2 shown in Chapter 5 with the substitutivity axioms for ':' can be used:

$$
\begin{aligned}
Eq_{CH1} = \ \{ \ & eq_1(X,Y) \leftarrow eq_2(X,Z), eq_2(Y,Z) && (7.1)\\
, \ & eq_2(X,X) \leftarrow && (7.2)\\
, \ & eq_2(X,Z) \leftarrow eq_3(X,Y), eq_2(Y,Z) && (7.3)\\
, \ & eq_3(X,Y) \leftarrow X = Y && (7.4)\\
, \ & eq_3(X:Y_1, X:Y_2) \leftarrow eq_3(Y_1,Y_2) && (7.5)\\
, \ & eq_3(X_1:Y, X_2:Y) \leftarrow eq_3(X_1,X_2) && (7.6)\\
\}. &
\end{aligned}
$$

Eq_{CH2} is defined similarly. All the correctness and completeness results for Eq_1 and Eq_2 shown in Chapter 5 hold for Eq_{CH1} and Eq_{CH2} as well. Reduction of the search space for term evaluation using the canonicality predicate (§5.9) can be done for Eq_{CH1} and Eq_{CH2} in the same way.

In the functional domain of an LPCH program, every object represents a function (because of *Fun*) and every function defined by the program is represented by an object (because of *Rep*). All the relations defined by a program are therefore defined over the functions that the program defines. In LPCH, thus, defining relations over functions is a programming method supported by a formal semantics in terms of functional domains. For instance, the

following program P computes the relation $geq(F, G, L)$, where F and G are unary functions and L is a list of arguments $x_1, ..., x_n$, that holds if $F(x_i) \geq G(x_i)$ for all $1 \leq i \leq n$:

$$\begin{aligned} P = \ & \{ \ geq(F, G, nil) \leftarrow \\ & , \ geq(F, G, cons : X : L) \leftarrow F : X \geq G : X, \ geq(F, G, L) \\ & \}. \end{aligned}$$

This ability to define relations over functions has been found to bring about an interesting programming paradigm [12].

LPCH does not treat lambda expressions. A method of integrating lambda expressions into Horn clause logic programming is proposed by Cheng [11], where it is shown that a set of function definitions using lambda expressions is intertranslatable to a computationally equivalent set of equations by the processes called λ-lifting and λ-abstraction. Using this method, for example, the term '*twice* : *twice*', which is a normal form in the LPCH program in Example 7.1, is λ-abstracted to the lambda expression '$\lambda f.\lambda x.f(f(f(fx)))$'. Using Cheng's method, the equation defining Curry's Y combinator

$$Y = \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))$$

is translated into the following set of equations

$$\begin{aligned} Y = \ & \{ \ y : X = (g : X) : (g : X) \leftarrow \\ & , \ g : X : Y = X : (Y : Y) \leftarrow \\ & \}. \end{aligned}$$

With the following set of equations defining the "functional" f for the factorial function

$$\begin{aligned} P = \ & \{ \ + : 0 : X = X \leftarrow \\ & , \ + : (s : X) : Y = s : (+ : X : Y) \leftarrow \\ & , \ * : 0 : X = 0 \leftarrow \\ & , \ * : (s : X) : Y = + : (* : X : Y) : Y \leftarrow \end{aligned}$$

$$
\begin{aligned}
&, \quad f : F : 0 = s : 0 \leftarrow \\
&, \quad f : F : (s : N) = * : (s : N) : (F : N) \leftarrow \\
&\},
\end{aligned}
$$

the term '$y : f$' represents the factorial function, and the factorial of $n = s : (s : (\cdots(s : 0)\cdots))$ can be found by an SLD-refutation of $Y \cup P \cup Eq_{CH1} \cup \{\leftarrow eq_1(y : f : n, X)\}$.

7.6 Canonical Terms in LPCH Programs

In Chapter 6 we presented a semantical model based on constructor terms. Our motivation for this semantical model was twofold: the undecidability of irreducible terms and the plausibility of constructor terms as canonical terms. The undecidability of irreducible terms remains for LPCH programs, if the reduction for LPCH programs is defined using Eq_{CH1} (Definition 5.4). This fact again forces us to seek a decidable subset of irreducible terms that may be regarded as canonical.

Here as in Chapter 6, we may arbitrarily choose a decidable subset S of the set of irreducible terms and redefine "canonical" terms as the members of S, so far as a formal development is concerned. As an example, however, we focus on a particular such S in the following.

Definition 7.5 Let P be an LPCH program. The constant symbol f is said to be a *defined constant symbol of functional arity n* if there is in P a clause of the form:

$$f : t_1 : \cdots : t_n = t \leftarrow B_1, ..., B_m, \quad m, n \geq 0.$$

A *constructor constant symbol* is a constant symbol that is not a defined constant symbol.

■

In the LPCH program P in Example 7.1, '+' and '*twice*' are the defined constant symbols of functional arity two; '0' and 's' are the constructor constant symbols. We shall assume that

no defined constant symbol is used with different functional arities. We shall also assume that in every LPCH program, there is no clause of the form

$$x : t_1 : \cdots : t_n = t \leftarrow B_1, ..., B_m, \quad m, n \geq 0,$$

where x is a variable, t and t_i are terms, namely that every clause is of the form

$$f : t_1 : \cdots : t_n = t \leftarrow B_1, ..., B_m, \quad m, n \geq 0,$$

where f is a constant symbol.

A ground term whose constant symbols are all constructor constant symbols, is said to be a *constructor term*. Clearly all the constructor terms so defined are irreducible (because of the restriction of the form of the clauses in an LPCH program stated above) and should be regarded as canonical. It is desirable, however, to accept as canonical more than the constructor terms, namely the irreducible terms containing a subterm consisting of a defined constant symbol of functional arity n supplied with less than n arguments. Such a subterm may be shown to be equal to a constructor term when supplied with additional arguments making the number of the arguments equal to the functional arity. In the program in Example 7.1, '*twice* : *s*' is irreducible and contains a defined constant symbol '*twice*' of functional arity two. It represents a useful function "*plus* 2" and becomes equal to $s : (s : t)$ when supplied with one more term t. If on the other hand a defined constant symbol of functional arity n is supplied with more than or equal to n arguments and still is irreducible, then it can never be equal to a constructor term even after being supplied with additional arguments.

The consideration above suggests this definition of *strongly canonical* and *weakly canonical* terms.

Definition 7.6 A ground term in an LPCH program is said to be *strongly canonical* iff it is irreducible and does not contain a subterm of the form:

$$f_n : t_1 : \cdots : t_m, \tag{7.7}$$

where f_n is a defined constant symbol of functional arity n, such that $m \geq n$. An irreducible term that is not strongly canonical is said to be *weakly canonical*. ■

Clearly every constructor term is strongly canonical. The following theorem shows the redundancy of the irreducibility condition in the above definition.

Theorem 7.2 *A ground term t is irreducible and does not contain a subterm of the form (7.7) iff t does not contain a subterm of the form (7.7).*

Proof The only-if part is obvious. For the if part, suppose that (1) t is reducible or (2) t contains a subterm of the form (7.7).

case 1: If t is reducible then it contains a subterm of the form:

$$g_n : u_1 : \cdots : u_n, \quad n \geq 0$$

where g_n is a defined constant symbol of functional arity n, which is of the form (7.7).

case 2: If t contains a subterm of the form (7.7), then t contains a subterm of the form (7.7). ■

Theorem 7.3 *A ground term t in an LPCH program is strongly canonical iff it does not contain a subterm of the form:*

$$f_n : t_1 : \cdots : t_m,$$

where f_n is a defined constant symbol of functional arity n, such that $m \geq n$.

Proof By Definition 7.6 and Theorem 7.2. ■

Theorem 7.4 *The set of strongly canonical terms is decidable.*

Proof By Theorem 7.3. ■

Every ground term t in an LPCH program takes the form:

$$f : t_1 : \cdots : t_m, \quad m \geq 0$$

where f is a constant symbol. Suppose that f is a defined constant symbol of functional arity n. If each t_i does not contain a subterm of the form (7.7) and $m < n$, then t itself does not contain a subterm of the form (7.7). Suppose that f is a constructor constant symbol. If each t_i does not contain a subterm of the form (7.7), then t itself does not contain a subterm of the form (7.7). We have therefore arrived at the following recursive definition of strongly canonical terms.

Theorem 7.5 *The set of strongly canonical terms is equal to the following recursively defined set of terms:*

1. $f_n : t_1 : \cdots : t_m$, *where* f_n *is a defined constant symbol of functional arity* n, *is strongly canonical if each* t_i *is strongly canonical and* $0 \leq m < n$,

2. $c : t_1 : \cdots : t_m$, *where* c *is a constructor constant symbol and* $m \geq 0$, *is strongly canonical if each* t_i *is strongly canonical.*

Proof By the argument above. ■

Example 7.2 Let P be the following program.

$$
\begin{aligned}
P = \ \{ \ & + : 0 : X = X \leftarrow \\
, \ & + : (s : X) : Y = s : (+ : X : Y) \leftarrow \\
, \ & twice : F : X = F : (F : X) \leftarrow \\
, \ & map : F : nil = nil \leftarrow \\
, \ & map : F : (cons : X : Y) = cons : (F : X) : (map : F : Y) \leftarrow \\
\}. &
\end{aligned}
$$

Some examples of strongly canonical terms are '0', '$s : 0$', 'nil', '$cons : 0 : nil$', '$+ : (s : 0)$', '$twice : (+ : (s : 0))$', '$map : s$', and '$cons : twice : map$'. Examples of weakly canonical terms are '$+ : nil : nil$', '$+ : s : twice$', '$map : s : +$', and '$s : (map : s : 0)$'. ■

A semantical model based on strongly canonical terms is now readily constructed using the

method described in Chapter 6. The variables in a clause in an LPCH program are now to be thought of as ranging specifically over strongly canonical terms. The function g from congruence closures to Herbrand interpretations that discards the terms not equal to any strongly canonical terms is defined by:

$g(I) = \{p(t_1, ..., t_n) \in B_P : p(t_1, ..., t_n) \in I$ and for all $1 \leq i \leq n$, $[t_i]_{\simeq_I}$ contains a strongly canonical term $\}$.

The function G_P associated with an LPCH program P is defined by $G_P(I) = g(F_P(I))$. The function G_P can be shown to be continuous using the same argument in Chapter 6. The denotation of a program P is taken to be the least fixpoint of G_P, which is a subset of the least E-model of P that contains certain ground atoms whose arguments are all equal to a strongly canonical term. If P is ground confluent, the least fixpoint of G_P can be computed by the following equality axioms, with the same justification for the correctness and completeness under ground confluence contained in §6.6.

$$
\begin{aligned}
Eq'_{CH1} = \ \{ \ & eq_1(X,Y) \leftarrow eq_2(X,Z), eq_2(Y,Z) && (7.8)\\
, \ & eq_2(X,X) \leftarrow strongly\_canonical(X) && (7.9)\\
, \ & eq_2(X,Z) \leftarrow eq_3(X,Y), eq_2(Y,Z) && (7.10)\\
, \ & eq_3(X,Y) \leftarrow X = Y && (7.11)\\
, \ & eq_3(X:Y_1, X:Y_2) \leftarrow eq_3(Y_1,Y_2) && (7.12)\\
, \ & eq_3(X_1:Y, X_2:Y) \leftarrow eq_3(X_1,X_2) && (7.13)\\
\ & \}.
\end{aligned}
$$

It is assumed that Eq'_{CH1} includes a possibly infinite number of ground unit clauses for the relation *strongly_canonical*, enumerating all strongly canonical terms.

A specialized SLD-resolution with the strong canonicality test (*SLDSC-resolution*) can be defined analogously to SLDCT-resolution. A goal of the form $\leftarrow$ *strongly_canonical*(t) succeeds iff there is a ground instance of t that is a strongly canonical, without instantiating

t, and when it has succeeded all the variables in t are *restricted*. The restriction on and modification of unification are the following.

1. only a unifier $\theta = \{x_1/t_1, ..., x_n/t_n\}$ may be used such that for all $1 \leq i \leq n$, if x_i is restricted then t_i must have a ground instance that is strongly canonical,
2. if a unifier $\theta = \{x_1/t_1, ..., x_n/t_n\}$ is used and x_i is restricted, then all the variables in t_i are restricted, for all $1 \leq i \leq n$.

The restricting of variables is needed to ensure the correctness of computed answer substitutions with respect to $lfp(G_P)$.

We now prove that it is decidable whether a term has a ground instance that is strongly canonical.

Theorem 7.6 *A term t in an LPCH program has a ground instance that is strongly canonical iff it does not contain a subterm of the form:*

$$f_n : t_1 : \cdots : t_m, \tag{7.14}$$

where f_n is a defined constant symbol of functional arity n, such that $m \geq n$.

Proof *if*: Suppose that t does not have a ground instance that is strongly canonical. Let θ be any ground substitution for t. Then $t\theta$ is not strongly canonical. By Theorem 7.3, $t\theta$ contains a subterm of the form:

$$f_n : t_1\theta : \cdots : t_m\theta$$

such that $m \geq n$, where f_n is a defined constant symbol of functional arity n and $f_n : t_1 : \cdots : t_m$ is a subterm of t. Thus t contains a subterm of the form (7.14).

only if: Suppose that t contains a subterm $f_n : t_1 : \cdots : t_m$ of the form (7.14). Let θ be any ground substitution for t. Then $f_n : t_1\theta : \cdots : t_m\theta$ is a subterm of $t\theta$. By Theorem 7.3, $t\theta$ is not strongly canonical. Hence t does not have any ground instance that is strongly canonical. ■

Theorem 7.7 *It is decidable whether a term in an LPCH program has a ground instance that is strongly canonical.*

Proof By Theorem 7.6. ■

Every term t in an LPCH program takes the form:

$$u : t_1 : \cdots : t_m, \quad m \geq 0,$$

where u is a constant symbol or a variable. Suppose that u is a defined constant symbol of functional arity n. If each t_i does not contain a subterm of the form (7.14) and $m < n$, t itself does not contain a subterm of the form (7.14). Suppose that u is a constructor constant symbol or a variable. If each t_i does not contain a subterm of the form (7.14), t itself does not contain a subterm of the form (7.14). We have therefore arrived at the following recursive definition of the terms having a ground instance that is strongly canonical.

Theorem 7.8 *The set of terms having a ground instance that is strongly canonical is equal to the following recursively defined set of terms.*

1. $f_n : t_1 : \cdots : t_m$, *where* f_n *is a defined constant symbol of functional arity* n, *has a ground instance that is strongly canonical if each* t_i *has a ground instance that is strongly canonical and* $0 \leq m < n$,

2. $u : t_1 : \cdots : t_m$, *where* u *is a constructor constant symbol or a variable and* $m \geq 0$, *has a ground instance that is strongly canonical if each* t_i *has a ground instance that is strongly canonical.*

Proof By the argument above. ■

For any atom A, let $[A]$ be the set of all ground atoms obtained by instantiating the variables in A to some *strongly canonical terms*. Let P be an LPCH program and G a goal clause $\leftarrow B_1, ..., B_m$. Let B_i^T be the transformed atom of B_i. Since in our new semantical model,

all the variables in a clause range over strongly canonical terms, the correctness of SLDSC-resolution with respect to $lfp(G_P)$ consists in the fact that $[B_i\theta] \subseteq lfp(G_P)$ for all $1 \leq i \leq m$, where θ is any computed answer substitution obtained from any SLDSC-refutation of $P_T \cup Eq'_{CH1} \cup \{\leftarrow B_1^T, ..., B_m^T\}$. For the same reason, the completeness of SLDSC-resolution with respect to $lfp(G_P)$ consists in the fact that for any substitutions θ such that $[B_i\theta] \subseteq lfp(G_P)$ for all $1 \leq i \leq m$, there is an SLDSC-refutation of $P_T \cup Eq'_{CH1} \cup \{\leftarrow B_1^T, ..., B_m^T\}$ with the computed answer substitution σ such that $\theta = \sigma\gamma$ for some substitution γ. The correctness and completeness under ground confluence of SLDSC-resolution with respect to $lfp(G_P)$ can be proved analogously to those of SLDCT-resolution proved in §§6.6 and 6.7.

Chapter 8

Conclusions

8.1 Contributions to the Amalgamation

We have shown that in amalgamating functional and relational programming within the framework of Horn clauses with equality, one inference system of SLD-resolution suffices—that there is no need to use separate inference rules for equality. In particular, SLD-resolution of the equality axioms Eq_1 shown in Chapter 5 gives the same operational effect as that of narrowing—an inference rule considered to be crucial to the amalgamation by some researchers (e.g., [8,22,40]). In our opinion, it is advantageous to use a uniform inference system within a logic programming system, to exploit the well-developed Prolog implementation techniques, and not to impose on the implementors the burdensome task of implementing some separate inference rules and of interfacing them with SLD-resolution. It is thus satisfying to see that the amalgamation can be done by SLD-resolution of our equality axioms.

As shown in Chapter 6, the canonical terms as irreducible terms become undecidable in the presence of the equational clauses (i.e., equations with conditions). Our solution is to take a decidable set of constructor terms (or strongly canonical terms in the case of LPCH) as canonical terms and to construct a semantical model based on it, namely

the least fixpoint of the function G_P associated with a logic program P. Admittedly, this semantical model using the fixpoint technique is a disappointment from the viewpoint of logic programming, since it is not a logical semantics. We hope that this semantical model will be superseded by a logical system that deals with constructor terms properly.

We have shown that higher-order functions can be incorporated into Horn clauses with equality by adopting the Schönfinkel-Curry function application operator and constructing a simple but adequate semantical model by means of functional domains. It is satisfying to see, for the purpose of designing a logic programming language, that higher-order functions can be defined and computed within the framework of Horn clauses with equality without use of higher-order logic, which is believed to be computationally intractable because of the undecidability of unification and large search spaces.

To test the feasibility of the proposals in Chapters 5 and 7, a prototype system called AP has been built [12] running an optimized version of Eq_1 and Eq_2 as a Prolog program and incorporating higher-order functions using the method of LPCH. Our experience with AP suggests that it gives the programmer expressive power beyond Lisp and Prolog.

8.2 Contributions to the Semantics of Logic Programs with Equality

In Chapter 4 we generalized the van Emden-Kowalski semantics of logic programs to the case where a logic program may contain equations, using the E-interpretations due to J.A. Robinson and to G. Robinson and Wos. We hope that our semantics will be useful in discovering and proving the properties of the least E-models and of proof methods for logic programs with equality, independently of the particular applications of equality.

8.3 Future Research

Since AP [12] executes our equality axioms as a user program, term reduction is inefficient with respect both to time and space. Important future research will therefore be to investigate the efficient interpretation or compilation methods that give the effect of executing our equality axioms. In particular, the research on the methods of efficiently executing LPCH programs, where all ground terms can be regarded as combinators, could benefit from the recent research on the use of combinators for functional programming, especially on fast reduction of combinators (e.g., [53,29]).

Another possible research avenue would be to investigate, in the context of logic programs with equality, the role of "negation as failure"—an inference rule that deduces the negation of an atom A if there exists a finite SLD-tree of $P \cup \{\leftarrow A\}$ containing no SLD-refutations. In the case of logic programs without equality, the correctness and completeness of the negation as failure rule are stated in terms of the "completion" of a logic program (e.g., [66, Chapter 3]). Also, [2,66] studied various relationships among the finite failure sets, the negation as failure rule, and the greatest fixpoints of the function T_P associated with a logic program P. It would be especially interesting to see whether the parallel results can be obtained for logic programs with equality using our semantics proposed in Chapter 4.

Appendix A

Fundamentals of First-Order Logic

This appendix contains some fundamentals of first-order logic that are necessary for understanding this dissertation. They are taken from [6,28,40].

A.1 First-Order Languages

The symbols (or alphabet) of a first-order language is divided into the following categories:

1. relation symbols (or predicates),
2. function symbols (or functors),
3. the universal quantifier $\forall$ and the existential quantifier $\exists$,
4. propositional (or sentential) connectives: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (conditional), $\leftrightarrow$ (biconditional),
5. variables,
6. the equality symbol: $=$ (optional),
7. punctuation symbols: '(', ')', ','.

The symbols in the categories (1) and (2) are called the *parameters* of a first-order language, while those in (3)–(7) are called the *logical symbols*. A first-order language is specified by the parameters and indicating whether the equality symbol is present as a logical symbol. When the equality symbol is present as a logical symbol, it receives the fixed interpretation of the identity relation (see below). Each function or relation symbol has associated with it the *arity* n which is some natural number. A function (relation) symbol with arity n is called an n-place function (relation) symbol. A zero-place function symbol is also called a *constant symbol*. Even when the equality symbol is a logical symbol, it is a two-place relation symbol.

The *formulas* of a first-order language are built up of the *terms* and *atomic formulas*. The set of terms is defined recursively as follows:

1. a variable is a term,
2. a constant symbol is a term,
3. if f is an n-place function symbol and $t_1, ..., t_n$ are terms, $f(t_1, ..., t_n)$ is a term.

An atomic formula is an expression of the form $p(t_1, ..., t_n)$ where p is an n-place relation symbol and $t_1, ..., t_n$ are terms. The set of formulas is defined recursively as follows:

1. an atomic formula is a formula,
2. if φ and ψ are formulas, so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$,
3. if x is a variable and φ is a formula, $\forall x\varphi$ and $\exists x\varphi$ are formulas.

Free occurrence of variables is defined recursively as follows. Let x be a variable.

1. For an atomic formula α, x occurs free in α iff x occurs in α.
2. x occurs free in $(\neg\varphi)$ iff x occurs free in φ.

3. x occurs free in $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi)$, or $(\varphi \leftrightarrow \psi)$ iff x occurs free in φ or ψ.

4. x occurs free in $\forall y\varphi$ or $\exists y\varphi$ iff $x \not\equiv y$ and x occurs free in φ.

A formula in which no variables occur free is said to be a *sentence*.

A.2 Interpretations, Truth, and Models

Formal semantics of a first-order language is given by means of interpretations which determine the truth and falsity of sentences.

An *interpretation* (or *structure*) of a first-order language L is a function whose domain is the set of parameters of L and the universal quantifier $\forall$ such that

1. I assigns to the universal quantifier $\forall$ a nonempty set $|I|$ called the *universe* (or *domain*) of I,

2. I assigns to each n-place relation symbol p an n-ary relation $p^I \subseteq |I|^n$,

3. I assigns to each constant symbol c a member c^I of $|I|$,

4. I assigns to each n-place function symbol f an n-ary function f^I from $|I|^n$ to $|I|$.

Truth of formulas in an interpretation is defined by means of *satisfaction*. Let L be a first-order language, φ a formula of L, I an interpretation of L, and s a function from the set of the variables of L to $|I|$. The function s can be extended to the function $\bar{s}$ from the set of the terms of L to $|I|$ as follows.

1. For each variable x, $\bar{s}(x) = s(x)$.

2. For each constant symbol c, $\bar{s}(c) = c^I$.

3. If $t_1, ..., t_n$ are terms and f is an n-place function symbol, then

$$\bar{s}(f(t_1, ..., t_n)) = f^I(\bar{s}(t_1), ..., \bar{s}(t_n)).$$

The formal definition of satisfaction is given by defining recursively the ternary relation "*I satisfies φ with s*", written $\models_I \varphi[s]$.

1. $\models_I (t_1 = t_2)[s]$ iff $\bar{s}(t_1) = \bar{s}(t_2)$, when '=' is a logical symbol,

2. for each n-place relation symbol p,

$$\models_I p(t_1, ..., t_n)[s] \text{ iff } \langle \bar{s}(t_1), ..., \bar{s}(t_n) \rangle \in p^I,$$

3. $\models_I \neg\varphi[s]$ iff $\not\models_I \varphi[s]$,

4. $\models_I (\varphi \wedge \psi)[s]$ iff $\models_I \varphi[s]$ and $\models_I \psi[s]$,

5. $\models_I (\varphi \vee \psi)[s]$ iff $\models_I \varphi[s]$ or $\models_I \psi[s]$,

6. $\models_I (\varphi \rightarrow \psi)[s]$ iff $\not\models_I \varphi[s]$ or $\models_I \psi[s]$,

7. $\models_I (\varphi \leftrightarrow \psi)[s]$ iff $\models_I (\varphi \rightarrow \psi)[s]$ and $\models_I (\psi \rightarrow \varphi)[s]$,

8. $\models_I \forall x\varphi[s]$ iff for all $d \in |I|$, $\models_I \varphi[s(x|d)]$, where $s(x|d)$ is the function from the variables to $|I|$ such that

$$s(x|d)(y) = \begin{cases} s(y) & \text{if } y \not\equiv x \\ d & \text{if } y \equiv x, \end{cases}$$

9. $\models_I \exists x\varphi[s]$ iff for some $d \in |I|$, $\models_I \varphi[s(x|d)]$, where $s(x|d)$ is the function above.

Let Γ be a set of formulas and φ a formula. Γ *logically implies* φ, written $\Gamma \models \varphi$, iff for all interpretations I and for all functions s from the variables to $|I|$ such that I satisfies every member of Γ with s, I also satisfies φ with s. For a sentence φ, either I satisfies φ with every s or I satisfies φ with no s. In the former case, we say that *φ is true in I* or *I is a model of φ*, written $\models_I \varphi$. In the latter case, we say that *φ is false in I*. An interpretation I is said to be a model of a set Γ of sentences iff I is a model of every member of Γ.

Proposition A.1 *For all sets of sentences* Γ *and for all sentences* φ, $\Gamma \models \varphi$ *iff* φ *is true in every model of* Γ. ■

A set Γ of formulas is said to be *satisfiable* iff there exist an interpretation I and a function s such that I satisfies every member of Γ with s.

Proposition A.2 *Let* Γ *be a set of formulas and* φ *a formula.* $\Gamma \models \varphi$ *iff* $\Gamma \cup \{\neg\varphi\}$ *is unsatisfiable.* ■

A.3 Homomorphisms and Initiality

Let I and J be interpretations. A homomorphism h from I to J is a function from $|I|$ to $|J|$ such that

1. for each n-place relation symbol p and each $\langle d_1, ..., d_n\rangle \in |I|^n$,

$$\text{if } \langle d_1, ..., d_n\rangle \in p^I \text{ then } \langle h(d_1), ..., h(d_n)\rangle \in p^J,$$

2. for each n-place function symbol f and each $\langle d_1, ..., d_n\rangle \in |I|^n$,

$$h(f^I(d_1, ..., d_n)) = f^J(h(d_1), ..., h(d_n)).$$

   In particular, $h(c^I) = c^J$ for all constant symbols c.

If h is bijective, it is called an *isomorphism* from I to J. If there is an isomorphism from I to J, I and J are said to be *isomorphic*.

An *initial model* of a set of sentences Γ is a model of Γ from which there is a *unique* homomorphism to *any* model of Γ.

Proposition A.3 *Any two initial models of a set of sentences are isomorphic.* ■

References

[1] H. Abramson. A prological definition of HASL: a purely functional language with unification-based conditional biding expressions. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 73–129, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[2] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[3] J. Backus. Can programming be liberated from the von Nuemann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. LEAF: a language which integrates logic, equations and functions. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 201–238, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 160–166, IEEE, February 1984. Atlantic City, New Jersey.

[6] J. Barwise, editor. *Handbook of Mathematical Logic*. North-Holland, New York, 1977.

[7] L.D. Baxter. *The Complexity of Unification*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1976.

[8] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, pages 119–132, Lecture Notes in Computer Science, vol. 213, Springer-Verlag, 1986.

[9] S. Burris and H.P. Sankappanavar. *A Course in Universal Algebra.* Springer-Verlag, New York, 1981.

[10] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. HOPE: an experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, 1980. Stanford.

[11] M.H.M. Cheng. *λ-Equational Logic Programming.* PhD thesis, University of Waterloo, Waterloo, Ontario, 1987.

[12] M.H.M. Cheng and K. Yukawa. *AP: An Assertional Programming System.* Technical Report CS-86-11, Department of Computer Science, University of Waterloo, Waterloo, Ontario, April 1986.

[13] A. Church and J.B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39:472–482, 1936.

[14] K.L. Clark. *Predicate Logic as a Computational Formalism.* Research Monograph, Department of Computing, Imperial College, London, 1979.

[15] S. Cohen. The APPLOG language. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 239–276, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[16] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, November 1984. Tokyo.

[17] A. Colmerauer. Prolog and infinite trees. In K.L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 231–251, Academic Press, London, 1982.

[18] G. Cousineau, P.-L. Curien, and B. Robinet, editors. *Combinators and Functional Programming Languages.* Volume 242 of *Lecture Notes in Computer Science*, Springer-Verlag, 1986.

[19] H.B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North-Holland, Amsterdam, 1958.

[20] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 37–70, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[21] N. Dershowitz and D.A. Plaisted. Logic programming cum applicative programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 54–66, IEEE, July 1985. Boston, Massachusetts.

[22] M. Dincbas and P. Vanhentenryck. *Extended Unification Algorithms for the Integration of Functional and Logic Languages.* manuscript, European Computer-Industry Research Centre, Munich, West Germany, 1986.

[23] E.W. Elcock and P. Hoddinott. Classical equality and Prolog. In *Proceedings of the Sixth Canadian Conference on Artificial Intelligence*, pages 103–108, May 1986. Montreal.

[24] E.W. Elcock and P. Hoddinott. Comments on Kornfeld's "Equality for Prolog": E-unification as a mechanism for augmenting the Prolog search strategy. In *Proceedings of AAAI-86*, pages 766–775, August 1986. Philadelphia, Pennsylvania.

[25] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[26] M.H. van Emden and J.W. Lloyd. A logical reconstruction of Prolog II. *The Journal of Logic Programming*, 1(2):143–149, 1984.

[27] M.H. van Emden and K. Yukawa. *Logic Programming with Equations.* Technical Report CS-86-05, Department of Computer Science, University of Waterloo, Waterloo, Ontario, October 1986. Also, to appear in the Journal of Logic Programming.

[28] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[29] J. Fairbairn and S.C. Wray. Code generation techniques for functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 94–104, ACM, August 1986.

[30] L. Fribourg. Oriented equational clauses as a programming language. *The Journal of Logic Programming*, 1(2):165–177, 1984.

[31] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 172–184, IEEE, July 1985. Boston, Massachusetts.

[32] U. Furbach and S. Hölldobler. Modelling the combination of functional and logic programming languages. *Journal of Symbolic Computation*, 2(2):123–138, 1986.

[33] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, ACM, January 1985. New Orleans, Louisiana.

[34] J.H. Gallier and S. Raatz. *Refutation Methods for Horn Clauses with Equality Based on E-Unification.* Technical Report MS-CIS-86-44, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, June 1986.

[35] J.H. Gallier and S. Raatz. SLD-resolution methods for Horn clauses with equality based on E-unification. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 168–179, IEEE, September 1986. Salt Lake City, Utah.

[36] J.A. Goguen. Abstract errors for abstract data types. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 491–522, North-Holland, Amsterdam, 1978.

[37] J.A. Goguen. How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation. In W. Bibel and R.A. Kowalski, editors, *5th Conference on Automated Deduction*, pages 356–373, Lecture Notes in Computer Science, vol. 87, Springer-Verlag, 1980.

[38] J.A. Goguen. One, none, a hundred thousand specification languages. In H.-J. Kugler, editor, *Information Processing 86*, pages 995–1003, Elsevier Science Publishers, New York, 1986.

[39] J.A. Goguen and R.M. Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs*, pages 221–256, Lecture Notes in Computer Science, vol. 164, Springer-Verlag, 1984.

[40] J.A. Goguen and J. Meseguer. EQLOG: equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[41] J.A. Goguen and J.J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In *Specifications of Reliable Software*, pages 170–189, IEEE, 1979.

[42] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Data Structuring, vol. 4*, pages 80–149, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[43] W.D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

[44] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF*. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[45] P. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall International, London, 1980.

[46] J. Herbrand. Investigations in proof theory: the properties of true propositions. In J. van Heijenoort, editor, *From Frege to Gödel, A Source Book in Mathematical Logic, 1879-1931*, pages 525–581, Harvard University Press, Cambridge, Massachusetts, 1967.

[47] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge, 1986.

[48] C.M. Hoffmann and M.J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[49] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[50] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[51] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, 1982.

[52] G. Huet and D.C. Oppen. Equations and rewrite rules, a survey. In R.V. Book, editor, *Formal Language Theory, Perspectives and Open Problems*, pages 349–405, Academic Press, New York, 1980.

[53] R.J.M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, ACM, August 1982.

[54] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R.A. Kowalski, editors, *5th Conference on Automated Deduction*, pages 318–334, Lecture Notes in Computer Science, vol. 87, Springer-Verlag, 1980.

[55] J. Jaffar and J.-L. Lassez. *Constraint Logic Programming.* manuscript, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1986.

[56] J. Jaffar, J.-L. Lassez, and M.J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 441–467, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[57] J. Jaffar, J.-L. Lassez, and M.J. Maher. *PROLOG-II as an Instance of the Logic Programming Language Scheme.* Technical Report RC 11707, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December 1985.

[58] J. Jaffar, J.-L. Lassez, and M.J. Maher. A theory of complete logic programs with equality. *The Journal of Logic Programming*, 1(3):211–223, 1984.

[59] K.M. Kahn. UNIFORM: a language based upon unification which unifies (much of) Lisp, Prolog, and ACT 1. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 411–438, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[60] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2,3):175–193, 1984.

[61] W.A. Kornfeld. Equality for Prolog. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 279–293, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[62] R.A. Kowalski. *Logic for Problem Solving.* Elsevier North Holland, New York, 1979.

[63] R.A. Kowalski. Predicate logic as programming language. In J.L. Rosenfeld, editor, *Information Processing 74*, pages 569–574, North-Holland Publishing Company, New York, 1974.

[64] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[65] G. Lindstrom. Functional programming and the logical variable. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 266–280, ACM, January 1985. New Orleans, Louisiana.

[66] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[67] J.A. Makowsky. *Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples.* manuscript, Department of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel, 1985.

[68] Y. Malachi, Z. Manna, and R. Waldinger. TABLOG: a new approach to logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 365–394, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[69] A.I. Mal'cev. *Algebraic Systems.* Springer-Verlag, 1973.

[70] A.I. Mal'cev. *The Metamathematics of Algebraic Systems.* North-Holland, Amsterdam, 1971.

[71] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[72] Per Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen et al., editors, *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland, 1982.

[73] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[74] J. McCarthy et al. *LISP 1.5 Programmer's Manual.* The M.I.T. Press, Cambridge, Massachusetts, 1962.

[75] G.F. McNulty. Fragments of first order logic, I: universal Horn logic. *The Journal of Symbolic Logic*, 42(2):221–237, 1977.

[76] A.R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1982.

[77] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 448–462, Lecture Notes in Computer Science, vol. 225, Springer-Verlag, 1986.

[78] R. Milner. A proposal for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197, ACM, August 1984.

[79] M.O. Newton. *A Combined Logical and Functional Programming Language.* Technical Report 5172:TR:85, Department of Computer Science, California Institute of Technology, Pasadena, California, 1985.

[80] M.J. O'Donnell. *Equational Logic as a Programming Language.* The MIT Press, Cambridge, Massachusetts, 1985.

[81] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[82] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 138–151, IEEE, July 1985. Boston, Massachusetts.

[83] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence, vol. 4*, pages 135–150, American Elsevier, New York, 1969.

[84] J.A. Robinson. The generalized resolution principle. In D. Michie, editor, *Machine Intelligence, vol. 3*, pages 77–93, American Elsevier, New York, 1968.

[85] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[86] J.A. Robinson and E.E. Sibert. LOGLISP: an alternative to PROLOG. In J.E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence, vol. 10*, pages 399–419, Halsted Press, John Wiley & Sons, New York, 1982.

[87] J.A. Robinson and E.E. Sibert. LOGLISP: motivation, design and implementation. In K.L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 299–313, Academic Press, London, 1982.

[88] P. Roussel. *PROLOG: Manuel de Reference et d'Utilisation.* Rapport interne, Groupe d'Intelligence Artificielle, UER de LUMINY, Universite d'Aix-Marseille, 1975.

[89] M. Sato and T. Sakurai. QUTE: a functional language based on unification. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 131–155, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[90] M. Schönfinkel. On the building blocks of mathematical logic. In J. van Heijenoort, editor, *From Frege to Gödel, A Source Book in Mathematical Logic, 1879-1931*, pages 355–366, Harvard University Press, Cambridge, Massachusetts, 1967.

[91] A. Selman. Completeness of calculi for axiomatically defined classes of algebras. *Algebra Universalis*, 2:20–32, 1972.

[92] G. Smolka. FRESH: a higher-order language with unification and multiple results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 469–524, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[93] A. Srivastava, D. Oxley, and A. Srivastava. An(other) integration of logic and functional programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 254–260, IEEE, July 1985. Boston, Massachusetts.

[94] G.L. Steele and G.J. Sussman. *Scheme: An interpreter for the extended lambda calculus.* AI memo 349, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1975.

[95] P.A. Subrahmanyam and J.-H. You. FUNLOG: a computational model integrating logic programming and functional programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 157–198, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[96] H. Tamaki. Semantics of a logic programming language with a reducibility predicate. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 259–264, IEEE, February 1984. Atlantic City, New Jersey.

[97] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16, Lecture Notes in Computer Science, vol. 201, Springer-Verlag, 1985.

[98] D.A. Turner. A new implementation technique for applicative languages. *Software–Practice and Experience*, 9(1):31–49, 1979.

[99] D.A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and Its Applications*, pages 1–28, Cambridge University Press, Cambridge, 1982.

[100] D.A. Turner. *SASL Language Manual.* Technical Report, St. Andrews University, December 1976.

[101] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In J.E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence, vol. 10*, pages 441–454, Halsted Press, John Wiley & Sons, New York, 1982.

[102] D.H.D. Warren. *Implementing PROLOG—Compiling Predicate Logic Programs.* D.A.I. Research Report No. 39, Department of Artificial Intelligence, University of Edinburgh, 1977.

[103] J.-H. You and P.A. Subrahmanyam. Equational logic programming: an extension to equational programming. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 209–218, ACM, January 1986. St. Petersburg Beach, Florida.

Index