«LE MATEMATICHE»
Vol. XLIII (1988) - Fasc. I, pp. 247-294

LOGIC PROGRAMMING EXTENSIONS OF HORN CLAUSE LOGIC

RON SIGAL (Catania) (*)

1. Introduction.

Logic programming is now firmly established as an alternative programming paradigm, distinct and arguably superior to the still dominant imperative style of, for instance, the Algol family of languages. The concept of a logic programming language is not precisely defined, but it is generally understood to be characterized by

1. a declarative nature, i.e. programs describe the desired results of a computation rather than prescribe the steps of a computation; and
2. foundation in some well understood logical system, e.g., first-order logic, whose model theory provides a meaning

(*) This paper was written while the author was a Visiting Professor supported by the Italian National Research Council (CNR). Current address: Department of Mathematics, Yale University, New Haven, CT 06520, USA.

for programs and whose proof theory characterizes their computations.

By usage and historical development logic programming has come to have more specific connotations.

3. Historical roots in clausal theorem proving based on resolution have brought forth the relational style of programming, in which the objects of computation are sets and relations, with functions playing a distinctly secondary role.
4. The need for computational efficiency has led to syntactical restrictions, in particular, to the language of Horn clauses, where the loss of expressive power is compensated by a breakthrough in interpreter efficiency.
5. This same restriction to Horn clauses has led, perhaps serendipitously, to the elegant model theoretic property that the meanings of programs can be characterized, up to a point, by a single canonical model, the minimal Herbrand model over ground terms.

Recent years have seen an evolution away from the restriction of logic programming to the domain of relations, an evolution toward the recognition of the first-class role of functions. This suggests that the classical notion of logic programming is more aptly named «relational programming», freeing the term «logic programming» for the more expansive terrain of functional programming, relational programming, and various integrations of the two.

On the other hand, the practical and theoretical charms of Horn clause logic are quite alluring, being a major source of the success of logic programming. There is a broad (though perhaps not universal) sense that they should be preserved as much as possible in the experimental definition of new languages and families of languages, and the extent to which they are is often taken, or given, as a criterion of the success of a design.

In this article we will survey some proposals for logic programming languages and language features, taking the point of view that these may be considered (as, indeed, they are frequently presented) as extensions of Horn clause logic. Loosely speaking we can identify two kinds of extensions: logical and algebraic. These are neither precise nor mutually exclusive categories, but by logical extensions we mean the inclusion of language features found in richer logical systems, such as negation and sorts. By algebraic extensions we mean equipping a language with features that support reasoning in a particular model or class of models with a richer structure than the term models.

The article is organized as follows. In section 2 we present the basic definitions and basic results for first-order and Horn clause logic. In the following four sections we discuss extensions of Horn clause logic by means of negation, function definitions, equality, and constraints. Most of the material in section 2 and 3 can be found in [17]. The material on functions and equality comes largely from [2] and [12] respectively, both of which appear in [9]. An alternative source for the material in section 2 and 5 is [18].

2. First-order logic and Horn clause logic.

Two of the main branches of mathematical logic are model theory and proof theory. Model theory is concerned with giving a mathematical characterization of what it means for a statement to be true, and proof theory is about formal, syntactic methods for discovering which statements are true. In the study of programming languages we have a similar distinction between *declarative* or *denotational semantics*, which assigns meanings to programs independent of any particular computational system, and *operational* or *procedural semantics*, which associates programs with computations. Logic programming languages, as the name implies, have roots in both logic and programming languages, and their declarative and operational semantics are (often) based on model theory and proof theory.

In the first three parts of this section we introduce first-order and Horn clause languages, and discuss the standard treatments of (section 2) declarative and (section 3) operational semantics. In the fourth part we look at an alternative approach to declarative semantics.

2.1. First-order and Horn clause languages.

Each first-order language is characterized by an alphabet, or *first-order signature*, $\Sigma = \langle \mathcal{F}, \mathcal{P}\ \mathrm{ar}\rangle$, where

- $\mathcal{F}$ is a countable (possibly infinite) set of function symbols;
- $\mathcal{P}$ is a countable (possibly infinite) set of predicate symbols;
- ar: $\mathcal{F} \cup \mathcal{P} \rightarrow$ Nat, where Nat is the set of natural numbers.

We also assume the availability of a countably infinite set $\mathcal{V}$ of variable symbols, independent of Σ.

We define the following sets of syntactic objects relative to signature Σ. The set of *Σ-terms over* $X \subseteq \mathcal{V}$, denoted $TM_\Sigma(X)$, consists of

- all $v \in X$
- all $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$, and $t_1, \ldots, t_n \in TM_\Sigma(X)$.

$TM_\Sigma(\mathcal{V})$ is the set of *Σ-terms* and will be denoted TM_Σ. $TM_\Sigma(\emptyset)$ is the set of *ground Σ-terms* and will be denoted GT_Σ. Σ-terms $f \in \mathcal{F}$, where $\mathrm{ar}(f) = 0$, are *Σ-constants. Atomic Σ-formulas*, or *Σ-atoms*, over $X \subseteq \mathcal{V}$ are

- all $p(t_1, \ldots, t_n)$, where $p \in \mathcal{F}$, $\mathrm{ar}(p) = n$, and $t_1, \ldots, t_n \in TM_\Sigma(X)$.

The set of *Σ-formulas* over $X \subseteq \mathcal{V}$, denoted $FM_\Sigma(X)$, consists of all

- Σ-atoms over X;
- $(\sim F)$, where $F \in FM_\Sigma(X)$;
- $(F \vee G)$, where $F, G \in FM_\Sigma(X)$;
- $(F \wedge G)$, where $F, G \in FM_\Sigma(X)$;
- $(\forall x)F$, where $x \in \mathcal{V}$ and $F \in FM_\Sigma(X)$;

- $(\exists x)F$, where $x \in \mathcal{V}$ and $F \in FM_{\Sigma}(X)$.

$FM_{\Sigma}(\mathcal{V})$ is the set of *Σ-formulas*, $FM_{\Sigma}(\emptyset)$ is the set of *ground Σ-formulas*, and they will be denoted FM_{Σ} and GF_{Σ} respectively. The pair $\langle TM_{\Sigma}, FM_{\Sigma}\rangle$ is the *first-order language* over Σ. Any subset of FM_{Σ} is a *first-order Σ-theory.*

We let $(F \leftarrow G)$ abbreviate $(F \vee \sim G)$ and let $(F \leftrightarrow G)$ abbreviate $((F \leftarrow G) \wedge (G \leftarrow F))$, where F, G are Σ-formulas. $\vee$ and $\wedge$ are associative, so parentheses will be omitted when no confusion results. *Σ-expressions* are either Σ-terms or Σ-formulas. If E is a Σ-expression, then $\text{var}(E) \subseteq \mathcal{V}$ denotes the set of all variables that occur in E.

Negated Σ-atoms are of the form $\sim A$, where A is a Σ-atom. *Σ-literals* are Σ-atoms or negated Σ-atoms. The set of *Σ-clauses*, denoted CL_{Σ}, is the subset of FM_{Σ} consisting of all Σ-formulas of the form $(L_1 \vee \ldots \vee L_n)$, where $L_1, \ldots, L_n$ are Σ-literals. We adopt the following special representation for Σ-clauses. Let a Σ-clause C be partitioned into

$$\{A_i | A_i \text{ is a } \Sigma\text{-atom in } C,\ i = 1, \ldots, n_1\} \cup$$

$$\{\sim B_i | \sim B_i \text{ is a negated } \Sigma\text{-atom in } C,\ i = 1, \ldots, n_2\}.$$

Then C can be represented

$$A_1 \vee \ldots \vee A_{n_1} \leftarrow B_1 \wedge \ldots \wedge B_{n_2},$$

or, more succinctly,

$$A_1, \ldots, A_{n_1} \leftarrow B_1, \ldots, B_{n_2}.$$

$A_1, \ldots, A_{n_1}$ is the *head* of C, and $B_1, \ldots, B_{n_2}$ is the *tail* of C. The set of *Horn Σ-clauses*, denoted HC_{Σ}, is the subset of CL_{Σ} consisting of all Σ-clauses with one of the following forms, where $n > 0$:

- $A \leftarrow$
- $A \leftarrow B_1, \ldots, B_n$
- $\leftarrow B_1, \ldots, B_n$
- $\leftarrow$

which are called, respectively, Σ-*facts,* Σ-*rules,* Σ-*goals*, and the *empty clause*. The pair $\langle TM_\Sigma, HC_\Sigma\rangle$ is the *Horn clause language* over Σ, and any subset of HC_Σ is a *Horn clause* Σ-*theory*.

Σ-*program statements* are Σ-facts and Σ-goals, and *Horn clause* Σ-*programs* are finite sets $\{C_1, \ldots, C_n\}$ of Σ-program statements, frequently written

$$\begin{matrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{matrix}$$

Thus, Horn clause Σ-programs are a particular kind of finite Σ-theory.

When a particular signature Σ is assumed we will sometimes refer simply to atoms, formulas, theories, literals, clauses, facts, rules, goals, and Horn clause programs. Also, throughout this section, program means Horn clause program.

2.2. *Declarative semantics.*

Intuitively programs represent assertions held to be true (in some intended domain), and goals are questions about the nature of that domain. Programming is the art of accurately describing the intended domain, and computation is the process of answering questions about the domain.

Horn clause programs inherit immediately a model theoretic semantics from first-order logic. Let Σ be some fixed signature. A Σ-*pre-interpretation* is a pair $\langle \mathcal{D}, \mu\rangle$, where

- the *domain* $\mathcal{D}$ is a set of objects, and
- μ is a map defined on $\mathcal{F}$ such that $\mu(f) : \mathcal{D}^n \to \mathcal{D}$, for $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$,

where $\mathcal{D}^n$ denotes the n-ary cartesian product of $\mathcal{D}$. A Σ-*interpretation* based on Σ-pre-interpretation $\langle \mathcal{D}, \mu\rangle$ is a pair $\langle \mathcal{D}, \mu \cup \mu'\rangle$, where

- μ' is a map defined on $\mathcal{P}$ such that $\mu'(p) \subseteq \mathcal{D}^n$, for $p \in \mathcal{P}$, $\mathrm{ar}(p) = n$.

Let $I = \langle \mathcal{D}, \mu \rangle$ be a Σ-pre-interpretation. A *variable assignment* into I is any map $\alpha : \mathcal{V} \rightarrow \mathcal{D}$, and the *term assignment* $\alpha_\Sigma : TM_\Sigma \rightarrow \mathcal{D}$ is the extension of α defined

$$\alpha_\Sigma(t) = \begin{cases} \alpha(t) & \text{if } t \text{ is } x \in \mathcal{V} \\ \mu(f)(\alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n)) & \text{if } t \text{ is } f(t_1, \ldots, t_n). \end{cases}$$

Since α_Σ is the unique extension of α with respect to Σ (and I) we will often simply refer to α when Σ is understood. If $X \subseteq \mathcal{V}$ is such that $\alpha(x) = x$ for all $x \in \mathcal{V} \backslash X$, we will sometimes write $\alpha : X \rightarrow \mathcal{D}$ to indicate that fact, not excluding thereby the possibility that $\alpha(x) = x$ for any $x \in X$. If $\alpha : \mathcal{V} \rightarrow \mathcal{D}$ is an assignment, and $x \in \mathcal{V}$ and $d \in \mathcal{D}$, then $\alpha^x_d : \mathcal{V} \rightarrow \mathcal{D}$ is the assignment

$$\alpha^x_d(y) = \begin{cases} d & \text{if } y \text{ is } x \\ \alpha(y) & \text{otherwise,} \end{cases}$$

and if $X \subseteq \mathcal{V}$ then $\alpha_{|X}$ is the assignment

$$\alpha_{|X}(x) = \begin{cases} \alpha(x) & \text{if } x \in X \\ x & \text{otherwise.} \end{cases}$$

The 3-ary *satisfaction* relation $\models$ on Σ-interpretations $I = \langle \mathcal{D}, \mu \rangle$, assignments $\alpha : \mathcal{V} \rightarrow \mathcal{D}$, and Σ-formulas (or Σ-theories) is defined as follows. Let $p(t_1, \ldots, t_n)$ be a Σ-atom, F, G be Σ-formulas, and $\mathbf{F}$ be a Σ-theory. Then

$$\begin{array}{lll} I, \alpha \models p(t_1, \ldots, t_n) & \text{iff} & \langle \alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n) \rangle \in \mu(p) \\ I, \alpha \models \sim p(t_1, \ldots, t_n) & \text{iff} & \langle \alpha_\Sigma(t_1), \ldots, \alpha_\Sigma(t_n) \rangle \notin \mu(p) \\ I, \alpha \models F \vee G & \text{iff} & I, \alpha \models F \text{ or } I, \alpha \models G \\ I, \alpha \models F \wedge G & \text{iff} & I, \alpha \models F \text{ and } I, \alpha \models G \\ I, \alpha \models (\exists x) F & \text{iff} & \text{there exists } d \in \mathcal{D} \text{ such that } I, \alpha^x_d \models F \\ I, \alpha \models (\forall x) F & \text{iff} & \text{for all } d \in \mathcal{D}\ I, \alpha^x_d \models F \\ I, \alpha \models \mathbf{F} & \text{iff} & I, \alpha \models F \text{ for all } F \in \mathbf{F} \end{array}$$

The binary *satisfaction* relation $\models$ on Σ-interpretations I and Σ-formulas F (or Σ-theories $\mathbf{F}$) is defined

$$\begin{array}{lll} I \models F & \text{iff} & I, \alpha \models F \text{ for all assignments } \alpha : \mathcal{V} \rightarrow \mathcal{D} \\ I \models \mathbf{F} & \text{iff} & I \models F \text{ for all } F \in \mathbf{F} \end{array}$$

If $I \models F$ we say I *satisfies* F, or F is *true* in I. For Σ-interpretation I and Σ-theory $\mathbf{F}$, if $I \models \mathbf{F}$ then I is a *model* of $\mathbf{F}$. If $\mathbf{F}$ has a model then it is *satisfiable*; otherwise it is *unsatisfiable*. For Σ-formula G, if $I \models \mathbf{F}$ implies $I \models G$ for all Σ-interpretations I, then G is a *logical consequence* of $\mathbf{F}$, denoted $\mathbf{F} \models G$.

Clauses are formulas, so the definition of satisfaction of clauses is given a fortiori by the above definition, but we write it out anyway. Let $A_1, \ldots, A_n$, $B_1, \ldots, B_m$ be atoms, let C be a clause, and let $\mathbf{C}$ be a set of clauses. Then

$$\begin{array}{lll}
I, \alpha \models A_1, \ldots, A_n \leftarrow & \text{iff} & I, \alpha \models A_i \text{ for some } i = 1, \ldots, n \\
I, \alpha \models \leftarrow B_1, \ldots, B_m & \text{iff} & I, \alpha \models \sim B_i \text{ for some } i = 1, \ldots, m \\
I, \alpha \models A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m & \text{iff} & I, \alpha \models A_1, \ldots, A_n \leftarrow \text{ or} \\
 & & \quad I, \alpha \models \leftarrow B_1, \ldots, B_m \\
I, \alpha \models \mathbf{C} & \text{iff} & I, \alpha \models C \text{ for all } C \in \mathbf{C} \\
I \models C & \text{iff} & I, \alpha \models C \text{ for all assignments} \\
 & & \quad \alpha : \mathcal{V} \to \mathcal{D} \\
I \models \mathbf{C} & \text{iff} & I \models C \text{ for all } C \in \mathbf{C}.
\end{array}$$

The first important result to note is that for the language of clauses over signature Σ we can restrict our attention to interpretations over a single canonical domain, namely, GT_Σ. The *$TM_\Sigma(X)$-pre-interpretation* is the Σ-pre-interpretation $\langle TM_\Sigma(X), \mu \rangle$ such that μ satisfies

$$\mu(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \text{ for all}$$

$$f \in \mathcal{F}, ar(f) = n, \text{ and } t_i \in TM_\Sigma(X), i = 1, \ldots, n,$$

and a *$TM_\Sigma(X)$-interpretation* is any Σ-interpretation based on $\langle TM_\Sigma(X), \mu \rangle$. An *Herbrand Σ-interpretation* is any GT_Σ-interpretation. In this context GT_Σ is often called the *Herbrand universe* of Σ and denoted U_Σ. The *Herbrand base* of Σ, denoted B_Σ, is defined

$$B_\Sigma = \{p(t_1, \ldots, t_n) | p \in \mathcal{P}, ar(p) = n, \text{ and } t_i \in GT_\Sigma,\ i = 1, \ldots, n\},$$

and since the behavior of functions in Herbrand Σ-interpretations is completely determined by Σ, each particular Herbrand Σ-interpretation can be identified with a subset of B_Σ, and vice versa. If I is an

Herbrand Σ-interpretation, $\mathbf{F}$ is a Σ-theory, and $I \models \mathbf{F}$, then I is an *Herbrand model* of $\mathbf{F}$.

THEOREM 1. *Let* $\mathbf{C}$ *be a set of* Σ*-clauses. Then* $\mathbf{C}$ *has a model if and only if it has an Herbrand model.*

For programs we can go further and restrict our attention to a single canonical model. Since each program P implicitly determines a unique signature, according to the function and predicate symbols that appear in P, we will sometimes refer to the Herbrand universe U_P and base B_P of program P, and to Herbrand interpretations for and models of P.

THEOREM 2. (Model intersection property) *Let* P *be a program and* $\{M_i\}$ *be a non-empty set of Herbrand models of* P*. Then* $\cap\{M_i\}$ *is also an Herbrand model of* P*.*

Since, as it is easy to verify, a program P always has B_P as a model, the set of Herbrand models of P is always non-empty, and so the *minimal Herbrand model* of P, defined as

$$M_P = \cap\{M | M \text{ is an Herbrand model of } P\},$$

always exists, and we have

THEOREM 3. *Let* P *be a program and* A *an atom in* B_P*. Then*

$$P \models A \textit{ if and only if } A \in M_P.$$

In Theorem 3 we have, for ground atoms, a model theoretic characterization of the meaning of a program, namely, its minimal Herbrand model. Although the minimal Herbrand model is defined nonconstructively, it can also be given a constructive characterization. First we need some more definitions.

Let $\langle TM_\Sigma, \mu\rangle$ be the TM_Σ-pre-interpretation. A Σ*-substitution* is a variable assignment $\theta : \mathcal{V} \to TM_\Sigma$ such that for some finite subset $X \subseteq \mathcal{V}$, $\theta(x) = x$ for all $x \in \mathcal{V} \backslash X$. Because of this restriction a

Σ-substitution is often represented explicitly as a set $\{x_1/t_1, \ldots, x_n/t_n\}$, where $\theta(x_i) = t_i$, $i = 1, \ldots, n$, and $\{x_1, \ldots, x_n\}$ is exactly the set of variables on which θ does not act as the identity map. The *identity substitution* acts as the identity map on all $x \in \mathcal{V}$. A Σ-substitution θ can be extended to the term assignment $\theta_\Sigma : TM_\Sigma \to TM_\Sigma$ in the usual way, and again we will generally refer simply to θ. We will write the application of θ to term t in postfix form, i.e., $t\theta$ means $\theta(t)$. We will also use the shorthand notation

$p(t_1, \ldots, t_n)\theta$	abbreviates	$p(t_1\theta, \ldots, t_n\theta)$
$\sim p(t_1, \ldots, t_n)\theta$	abbreviates	$\sim p(t_1\theta, \ldots, t_n\theta)$
$(A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m)\theta$	abbreviates	$A_1\theta, \ldots, A_n\theta \leftarrow B_1\theta, \ldots, B_m\theta$
$\mathbf{C}\theta$	abbreviates	$\{C\theta \mid C \in \mathbf{C}\}$

where $p(t_1, \ldots, t_n)$ is an atom, $A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$ is a clause, and $\mathbf{C}$ is a set of clauses. If E is a term, a literal, or a clause, then $E\theta$ is an *instance* of E, and if no variables occur in $E\theta$ then it is a *ground instance* of E. A Σ-substitution $\theta : \mathcal{V} \to GT_\Sigma$ is a *ground substitution*, and a Σ-substitution $\theta : \mathcal{V} \to \mathcal{V}$ is a *renaming substitution.* If E is a term, a literal, or a clause, and θ is a renaming substitution $\{x_1/y_1, \ldots, x_n/y_n\}$ such that $\{y_1, \ldots, y_n\} \cap X = \emptyset$ for some $X \subseteq \mathcal{V}$, then $E\theta$ is a *variant of E away from X*. The *composition* of two Σ-substitutions θ, σ, denoted $\theta \circ \sigma$ or simply $\theta\sigma$, is defined

$$x(\theta \circ \sigma) = \sigma_\Sigma(x\theta) \quad \text{for all } x \in \mathcal{V}.$$

Now, for program P the transformation function $T_P : 2^{B_P} \to 2^{B_P}$, where 2^{B_P} is the power set of B_P, i.e., the set of all Herbrand interpretations of P, is defined

$$T_P(I) = \{A \in B_P \mid A \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause in } P, \text{ and } B_i \in I, i = 1, \ldots, n\}.$$

T_P is monotonic and continuous on 2^{B_P}, and if we define

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow \beta + 1 &= T_P(T_P \uparrow \beta) \\ T_P \uparrow \lambda &= \bigcup_{\beta < \lambda} T_P \uparrow \beta, \ \lambda \text{ limit ordinal,} \end{aligned}$$

then we have (by results in [24]) that $T_P \uparrow \omega$ is the least fixpoint of T_P, denoted $lfp(T_P)$, and in particular we have

THEOREM 4. *Let P be a program. Then $M_P = lfp(T_P) = T_P \uparrow \omega$.*

Theorem 4 gives us a constructive characterization of the minimal model semantics of program P, but it is inadequate for the implementation of a programming language since it gives us only a naive enumeration of M_P. For a practical operational semantics we turn to the resolution method.

2.3. *Operational semantics.*

As we noted in the introduction to this section, we want to supplement the non-constructive definition of the meaning of theories (including programs) with a mechanical system, or *calculus*, for discovering the logical consequences of those theories. In logic we work with *proof systems*, which allow the construction, either by hand or automatically by computer, of *proofs*, syntactic objects intended to certify that a formula is a logical consequence of a theory. The important thing is that this certification be correct. We usually formalize two aspects of correctness: *soundness*, meaning that certified formulas are indeed logical consequences, and *completeness*, meaning that all logical consequences can be certified. For example, let $\mathcal{S}$ be a proof system for first-order languages, and let $\mathbf{F} \vdash_{\mathcal{S}} F$ denote that a proof of formula F from theory $\mathbf{F}$ can be constructed in $\mathcal{S}$. Then $\mathcal{S}$ is sound if

$$\mathbf{F} \vdash_{\mathcal{S}} F \text{ implies } \mathbf{F} \models F,$$

and it is complete if

$$\mathbf{F} \models F \text{ implies } \mathbf{F} \vdash_{\mathcal{S}} F.$$

Resolution, introduced in [19], is a family of proof systems for first-order logic, all of which are sound (when defined correctly) and some of which are complete. Naturally they work for Horn clause logic as well. However, we want more than a proof system: we will show that resolution can be adapted to give an operational characterization of Horn clause logic *as a programming language*. In this context our notions of soundness and completeness must take account of the desired characteristics of a programming language. These ideas will be made precise in this subsection.

Resolution is a milestone in a line of research into general syntactic automated theorem proving methods based on ideas introduced by J. Herbrand in 1930. These methods have in common the characteristic of reasoning over Herbrand interpretations, as justified by Theorem 1 and

THEOREM 5. (Herbrand's Theorem) *A set* $\mathbf{C}$ *of clauses is unsatisfiable if and only if there is a finite unsatisfiable set* $\mathbf{C}'$ *of ground instances of clauses in* $\mathbf{C}$.

The main ideas in this branch of automated theorem proving are the following.

1. These are generally refutation methods. That is, to show that a first-order formula F is a logical consequence of a set $\mathbf{C}$ of clauses, we take the clause form of the negation of F, call it $\sim F'$, and try to show that $\mathbf{C} \cup \{\sim F'\}$ is unsatisfiable. This accounts for the statement of Herbrand's Theorem in a negative form.
2. By Theorem 1 unsatisfiability is equivalent to unsatifiability in Herbrand interpretations, and the right hand side of Herbrand's Theorem gives us the necessary leverage in making this test by purely syntactic means. Given a finite subset $\mathbf{C}'$ of Σ-clauses $\mathbf{C}$, any assignment $\alpha :$ var$(\mathbf{C}') \to GT_\Sigma$ is by definition a ground substitution, and vice versa. Model theoretically, Herbrand's Theorem tells us to look for an assignment α such that for any Herbrand interpretation $I = \langle GT_\Sigma, \mu \rangle$ we have $I, \alpha \not\models \mathbf{C}'$. In other words, the constraints placed by $\mathbf{C}'$ on the relations assigned by μ to the predicate symbols in $\mathbf{C}'$ are set-theoretically contradictory.

Syntactically, an assignment into GT_Σ is a ground substitution θ, and contradictory constraints on the relations are reflected in the propositional unsatisfiability of $\mathbf{C}'\theta$.

The trick, then, is to find the right finite subset $\mathbf{C}'$ of clauses and the right ground substitution θ, which (if they exist) can be done, at great expense, by enumeration. The significance of resolution is the directness with which it hunts down $\mathbf{C}'$ and θ. There are two key ideas.

3. Contradictions depend on contradictory literals. The suggested heuristic, then, is to look for clauses with literals $p(t_1, \ldots, t_n)$ and $\sim p(u_1, \ldots, u_n)$, and try to find a substitution that will make them contradictory. This idea can already be found in [8].

4. It is not necessary to actually present a ground substitution that yields a contradiction: it suffices to show that one exists. For example, given literals $p(x)$ and $\sim p(f(y))$, the substitution $\{x/f(y)\}$ applied to $p(x)$ is such that *any* further instantiation to ground literals will produce a ground contradiction. The advantage of working with non-ground substitutions can be seen in the example of clauses $(p(x) \vee q(x))$ and $(\sim p(f(y)) \vee \sim q(f(b)))$. If we start out with the ground substitution $\{x/f(a), y/a\}$, for example, we are on the wrong track, but if we start with $\{x/f(y)\}$ we can go on to construct $\{x/f(y), y/b\}$, which yields a contradiction.

The tool used in resolution to implement idea (4) is *unification*. Until further notice, equality between pairs of Σ-terms s, t, denoted $s = t$, means that they are identical. A *unifier* of two Σ-terms s, t is a Σ-substitution θ such that $s\theta = t\theta$. Σ-terms s, t, are *unifiable* if they have a unifier. We now define an ordering $\leq$ on Σ-substitutions. Two Σ-substitutions θ, σ are *equal on* $W \subseteq \mathcal{V}$, denoted $\theta = \sigma[W]$, if $x\theta = x\sigma$ for all $x \in W$, and θ is *more general than* σ *on* $W \subseteq \mathcal{V}$, denoted $\sigma \leq \theta[W]$, if there is a Σ-substitution δ such that $\sigma = \theta\delta[W]$. A unifier θ of s, t is a *most general unifier (mgu)* if $\sigma \leq \theta[\text{var}(s) \cup \text{var}(t)]$ for all unifiers σ of s, t.

THEOREM 6. *Let s, t be two terms, and θ, σ two mgu's of s, t. Then $\theta = \sigma\rho[var(s) \cup var(t)]$ for some renaming substitution ρ.*

By Theorem 6 the mgu of two unifiable terms s, t is unique (with respect to its action on var(s)$\cup$var(t)) up to renaming. An algorithm which finds mgu's of terms is a *unification* algorithm, of which many have been given. See, for example, [15].

For atoms $p(t_1, \ldots, t_n)$ and $q(u_1, \ldots, u_m)$, we will say «$p(t_1, \ldots, t_n)$ and $q(u_1, \ldots, u_m)$ are unifiable» to abbreviate the conditions

1. p and q are the same predicate symbol;
2. $m = n$;
3. there is a substitution θ which unifies t_i, u_i, $i = 1, \ldots, n$,

and we will write $p(t_1, \ldots, t_n)\theta = q(u_1, \ldots, u_m)\theta$ to indicate that conditions (1) - (3) are satisfied and that θ satisfies (3).

Many versions of resolution have been proposed, varying in efficiency and completeness. We will present SLD-resolution, the standard version for Horn clause logic, which is incomplete in general but complete for Horn clauses. SLD-resolution begins with a program and a goal, and constructs a sequence of goals, each derived from the previous one, attempting to generate the empty goal $\leftarrow$, often written $\square$.

A *computation rule* R is a function which selects an atom from a goal. Let P be a program, G a goal, and R a computation rule. Then an *SLD-derivation* from $P \cup \{G\}$ via R is a (possibly infinite) sequence of triples $[G_0, *, *]$, $[G_1, C_1, \theta_1]$, $[G_2, C_2, \theta_2]$,... such that

1. G_0 is G and $*$ is an arbitrary symbol;
2. C_i, $i = 1, 2, \ldots$ is a variant away from $\bigcup_{j=0}^{i-1}$ var(G_j) of a clause in P;
3. θ_i, $i = 1, 2, \ldots$ is an mgu of the head of C_i and the atom in

G_{i-1} selected by R;

4. for $i = 1, 2, \ldots,$ if G_{i-1} is $\leftarrow A_1, \ldots, A_m,$ and C_i is $A \leftarrow B_1, \ldots, B_n,$ and A_j is the atom in G_{i-1} selected by R, then G_i is $(\leftarrow A_1, \ldots, A_{j-1}, B_1, \ldots, B_n, A_{j+1}, \ldots, A_m)\theta_i$.

An *SLD-refutation* of $P \cup \{G\}$ via R is a finite SLD-derivation from $P \cup \{G\}$ via R in which the last goal is $\square$. The *SLD success set* of a program P, denoted $SS(P)$, is

$$\{A \in B_P \mid \text{there exists an } SLD\text{-refutation of } P \cup \{\leftarrow A\} \text{ via } R$$
$$\text{for some computation rule } R\}.$$

The SLD success set of program P characterizes, for ground atoms, the operational semantics of P, and we have the following equivalence.

THEOREM 7. *Let P be a program. Then $SS(P) = M_P$.*

Theorem 7 reassures us that SLD-resolution behaves well with respect to ground atoms. However, it does not tell us very much about SLD-resolution as the basis of a computational system. The reason is that in a relational programming language, unlike a functional language, the only useful information we can extract is in the arguments of atoms, and if we were limited to asking about the truth of ground atoms, we would be forced to supply all of the information in advance, guessing an answer, in effect, and asking if it is correct.

In fact, SLD-resolution, in constructing a sequence of substitutions in a derivation, constructs answers as it seeks a refutation. We have two formal notions to capture this idea. Let P be a Σ-program and G a Σ-goal $\leftarrow A_1, \ldots, A_n$. A substitution $\theta : \mathrm{var}(G) \to TM_\Sigma$ is a *correct answer substitution* for $P \cup \{G\}$ if $P \models A_1\theta \wedge \ldots \wedge A_n\theta$. Let $[G_0, *, *]$, $[G_1, C_1, \theta_1], \ldots, [G_n, C_n, \theta_n]$ be a refutation of $P \cup \{G\}$ via R, for some computation rule R. Then $\theta_1 \circ \ldots \circ \theta_{n|\mathrm{var}(G)}$. is an *$R$-computed answer substitution* for $P \cup \{G\}$, and we have

THEOREM 8. (Soundness and completeness of SLD-resolution) *Let P be a program, G a goal, and R a computation rule.*

1. *If θ is an R-computed answer substitution for $P \cup \{G\}$, then*

it is a correct answer substitution for $P \cup \{G\}$.

2. *If* θ *is a correct answer substitution for* $P \cup \{G\}$*, then there is an* R*-computed answer substitution* σ *for* $P \cup \{G\}$ *and a substitution* γ *such that* $\theta = \sigma\gamma[var(G)]$.

In other words, we can pose queries in the form of non-ground goals, and SLD-resolution will return the most general correct substitution. Theorem 8 characterizes the role of SLD-resolution in supplying the operational semantics of Horn clause logic as a programming language.

2.4. *S-interpretations.*

We should note that in extending the semantics to account for answer substitutions, we have left behind the elegance we found in the minimal Herbrand model semantics. In both cases the philosophical position is that the semantics of a program is determined by truth in all of its models. It happens that, for ground atoms, truth in all models coincides with truth in the minimal Herbrand model; for non-ground atoms it turns out otherwise. For example, $int(x)$ is true in the minimal Herbrand model of the program P

$$\mathrm{int}(0) \leftarrow$$

$$\mathrm{int}(s(x)) \leftarrow \mathrm{int}(x)$$

but the identity substitution is not a correct answer substitution, and therefore not a R-computed answer substitution, because there are models of P in which $int(x)$ is not true.

If we hold that Horn clause programs should be seen as computing in the domain of the minimal Herbrand model, then SLD-resolution is certainly incomplete. To retain completeness we must consider more than the minimal Herbrand model, but, in fact, we need not surrender the existence of a natural canonical model. In [10] there is suggested the expedient of replacing the Herbrand universe with a domain based on non-ground terms. We define the binary relation $\approx_r$ on Σ-terms as follows. $s \approx_r t$, for Σ-terms s, t, if there exists a renaming substitution θ such that $s\theta = t$. $\approx_r$ is an equivalence relation, and our new «Herbrand» domain, denoted U^v_Σ, is the quotient structure

$$U^v_\Sigma = TM_\Sigma/_{\approx_r} = \{[t]_r | t \in TM_\Sigma\},$$

where $[t]_r$ is the equivalence class of t in TM_Σ with respect to $\approx_r$. An *Herbrand S-interpretation* for $\Sigma = \langle \mathcal{F}, \mathcal{P}, \text{ar}\rangle$ is a pair $\langle U^v_\Sigma, \mu\rangle$ such that μ satisfies

$$\mu(f)([t_1]_r, \ldots, [t_n]_r) = [f(t_1, \ldots, t_n)]_r, \text{ for}$$

$$f \in \mathcal{F},\ \text{ar}(f) = n, t_i \in TM_\Sigma,\ i = 1, \ldots, n.$$

As with ordinary Herbrand interpretations, Herbrand S-interpretations for Σ can be identified with subsets of the *Herbrand S-base* for Σ

$$B^v_\Sigma = \{p([t_1]_r, \ldots, [t_n]_r) | p \in \mathcal{P}, \text{ar}(p) = n, t_i \in TM_\Sigma, i = 1, \ldots, n\}.$$

However, for a non-ground atom, say $p(x)$, it is useful to let $p([x]_r) \in I$ imply that I also satisfies $p([x\theta]_r)$ for any substitution θ, so that $I_1 = \{p([x]_r)\}$ and $I_2 = \{p([x]_r), p([a]_r)\}$ both satisfy the same atoms. In this case an interpretation $\langle U^v_\Sigma, \mu\rangle$ that satisfies some non-ground atom can be identified with more that one subset of B^v_Σ. Therefore, we *define* an S-interpretation for Σ to be any subset of B^v_Σ.

Now, *S-satisfaction* of clauses in an Herbrand S-interpretation $I \subseteq B^v_\Sigma$ is defined

$I \models_S p(t_1, \ldots, t_n)$	iff	there exist $p([u_1]_r, \ldots, [u_n]_r) \in I$ and Σ-substitution σ such that $u_i\sigma = t_i,\ i = 1, \ldots, n$
$I, \theta \models_S A \leftarrow B_1, \ldots, B_n$	iff	$I \models_S B_i\theta$ for all $i = 1, \ldots, n$ implies $I \models_S A\theta$
$I \models_S A \leftarrow B_1, \ldots, B_n$	iff	$I, \sigma \models_S A \leftarrow B_1, \ldots, B_n$ for all Σ-substitutions σ
$I \models_S P$	iff	$I \models_S C$ for each clause $C \in P$

where $p(t_1, \ldots, t_n)$, A, $B_1, \ldots, B_n$ are Σ-atoms, P is a Σ-program, and θ is a Σ-substitution. If $I \models_S P$ then I is an *S-model* of P.

Let P be a Σ-program and I an arbitrary model of P. Then

$$I_S = \{p([t_1]_r, \dots, [t_n]_r) \in U_\Sigma^v \mid I \models p(t_1, \dots, t_n)\}$$

is clearly an S-model of P such that for any atom A

$$I_S \models_S A \text{ if and only if } I \models A,$$

and so

$$\begin{aligned} I \models_S A \text{ for all } S\text{-models } I \text{ of } P \text{ implies } & I_S \models_S A \text{ for all models } I \text{ of } P \\ \text{implies } & I \models A \text{ for all models } I \text{ of } P \\ \text{implies } & P \models A. \end{aligned}$$

On the other hand, let I be an S-model of P, and A an atom. It follows from the definition of $\models_S$ that

$$I \models_S A \text{ implies } I \models_S A\theta \text{ for every substitution } \theta,$$

and in fact, given the existence of the identity substitution,

(1) $I \models_S A$ if and only if $I \models_S A\theta$ for every substitution θ.

For substitution $\theta : \mathcal{V} \to TM_\Sigma$ we define the assignment $\theta_{\approx_r} : \mathcal{V} \to U_\Sigma^v$ as $[t]_r\theta_{\approx_r} = [t\theta]_r$. Now, a set $I \subseteq B_\Sigma^v$, together with the relation $\models_S$, determines a unique interpretation $I' = \langle U_\Sigma^v, \mu \rangle$ such that, for any substitution $\theta : \mathcal{V} \to TM_\Sigma$,

$$I \models_S A\theta \text{ if and only if } I', \theta_{\approx_r} \models A,$$

so that, by (1),

$$I \models_S A \text{ if and only if } I', \theta_{\approx_r} \models A, \text{ for every substitution } \theta : \mathcal{V} \to TM_\Sigma.$$

But any assignment $\alpha : \mathcal{V} \to U_\Sigma^v$ is $\theta_{\approx_r}$ for some $\theta : \mathcal{V} \to TM_\Sigma$, and so we have

$$I \models_S A \text{ if and only if } I' \models A.$$

Therefore,

$$\begin{aligned} P \models A \quad \text{implies} \quad & I \models A && \text{for all models } I \text{ of } P \\ \text{implies} \quad & I' \models A && \text{for all } S\text{-models } I \text{ of } P \\ \text{implies} \quad & I \models_S A && \text{for all } S\text{-models } I \text{ of } P, \end{aligned}$$

and putting the two chains of implications together we get

(2) $P \models A$ if and only if $I \models_S A$ for all S-models I of P.

It is shown in [10] that every program P has a minimal S-model M_P^S, defined, however, by a construction more complex than intersection. M_P^S has the property that, for any atom A,

$$M_P^S \models_S A \text{ if and only if } I \models_S A \text{ for all } S\text{-models } I \text{ of } P,$$

so that by (2),

$$M_P^S \models_S A \text{ if and only if } P \models A$$

and we have

THEOREM 9. *Let P be a program and G a goal $\leftarrow A_1, \ldots, A_n$. Then substitution θ is a correct answer substitution for $P \cup \{G\}$ if and only if $M_P^S \models_S \{A_1\theta, \ldots, A_n\theta\}$.*

Theorem 9 tells us that minimal S-models supply canonical models that also characterize correct answer substitutions. More specifically, it is shown in [10] that M_P^S exactly characterizes R-computed answer substitutions.

THEOREM 10. *Let P be a program and G a goal $\leftarrow A_1, \ldots, A_n$. Then substitution θ is an R-computed answer substitution for $P \cup \{G\}$ if and only if there exist $A'_1, \ldots, A'_n \in M_P^S$ such that θ' is an mgu of $\langle A_1, \ldots, A_n \rangle$ and $\langle A'_1, \ldots, A'_n \rangle$, and $\theta = \theta'[var(G)]$.*

At this point the philosophical question remains: Does the minimal S-model semantics sensibly reflect how the programmer thinks about programs? Indeed, it seems that it does, for two reasons.

First, the behavior one typically expects from queries put to an interpreter is to return ground substitutions. For example, when one writes the program P

$$\begin{array}{ll} \text{int}(0) \leftarrow & \text{plus}(0, y, y) \leftarrow \text{int}(y) \\ \text{int}(s(x)) \leftarrow \text{int}(x) & \text{plus}(s(x), y, s(z)) \leftarrow \text{int}(x, y, z), \end{array}$$

one uses the predicate plus to do arithmetic. Proving $(\forall x)\text{plus}(0, x, x)$, which holds in the minimal Herbrand model but not in all models of P, lies more in the realm of theorem proving (in the theory of arithmetic) than in programming. For P the minimal S-model coincides with the minimal Herbrand model, and they both satisfy exactly the desired ground atoms. Furthermore, the minimal S-model, though not the minimal Herbrand model, indicates that all of the correct answer substitutions are ground.

On the other hand, writing programs which generate partially determined, i.e., non-ground, substitutions in the course of computation is an intrinsic aspect of logic programming. When one write the one-line program Q

$\text{head}(\text{cons}(x, l), x) \leftarrow$

one is probably not so much interested in

(3) $$Q \models \text{head}(\text{cons}(x,\text{nil}),x)$$

as in

$$Q \models \text{head}(\text{cons}(a, \text{nil}),a)$$

where «a» is a constant, but (3) is necessary for head to work with arbitrary lists. Now consider the program R:

$\text{int}(0) \leftarrow$

$\text{int}(s(x)) \leftarrow \text{int}(x)$

$\text{head}(\text{cons}(x, l), x) \leftarrow$

$\text{tail}(\text{cons}(x, l), l) \leftarrow$

$\text{gen}(0, x, \text{nil}) \leftarrow$

$\text{gen}(s(n), x, \text{cons}(x, l)) \leftarrow \text{gen}(n, x, l)$

$\text{gen\_n\_heads}(l, n, h) \leftarrow \text{gen}(n, x, h), \text{head}(l, x)$

Again one is probably not so much interested in

(4) $$R \models \text{gen\_n\_heads}(\text{cons}(x, \text{nil}), s(s(0))), \text{cons}(x, \text{cons}(x,\text{nil}))$$

as in

$$R \models \text{gen\_n\_heads}(\text{cons}(a, \text{nil}), s(s(0))), \text{cons}(a, \text{cons}(a,\text{nil}))$$

but (4) is important operationally, at least with a left-to-right computation rule, because if gen were constrained to generate ground lists the computation of gen_n_heads would require searching through the ground lists in the gen relation to find one whose elements satisfy head.

The point is that there are times when one wants non-ground substitutions, and one programs them explicitly. In this case they show up in the minimal S-model, but not in the minimal Herbrand model.

3. Negation.

One of the most serious deficiencies, and most widely pursued extensions, of Horn clause logic is the ability to express negation. The problem is that if we add a negative literal to the body of a Horn clause, we immediately leave the realm of Horn clause logic and risk the loss of its many advantages. For example, if we add the negative literal $\sim C$ to the body of $A \leftarrow B$, then the result $A \leftarrow B, \sim C$ is equivalent to $A \vee C \leftarrow B$. Of course we could fall back on ordinary first-order semantics and on forms of resolution complete for first-order logic, but we will have forsaken a sensible programming language for general purpose theorem proving. The usual approach, rather, is to try to add negation while preserving as much as possible the benefits of Horn clause logic.

The first thing to notice is that a negative Σ-literal $\sim A$ can never be a logical consequence of a Σ-program P because B_Σ is always a model of $P \cup \{A\}$. One alternative, originally proposed in the field of databases, is to use the *Closed World Assumption*: if $P \not\models A$ then infer $\sim A$. The Closed World Assumption is a non-monotonic inference rule in the sense that it is possible to have P imply $\sim A$ and $P \cup Q$ not imply $\sim A$, in particular if joining Q to P allows the refutation of $P \cup Q \cup \{\leftarrow A\}$. The problem with the Closed World Assumption is that it is in general not computable because of the undecidability of the $P \models A$ relation for Horn Clause logic. The best we can do in practice is to approximate it, and the most straightforward approximation is simply to run SLD-resolution on $P \cup \{\leftarrow A\}$ and hope that it fails in a finite amount of time, in which case $\sim A$ may be inferred. This inference rule is called *negation as failure.*

Historically, then, the approach to negation began with an operational concept, which was later given a fixpoint and model theoretic characterization. We will proceed similarly in this exposition.

Let R be a computation rule, P a program, and G a goal. A derivation from $P \cup \{G\}$ via R is *failed* if it is finite and if the atom selected by R from the last goal does not unify with the head of any clause in P. $P \cup \{G\}$ has a *finitely failed* SLD *R-search space* if

1. there are a finite number of SLD-derivations from $P \cup \{G\}$ via R, and
2. each of them is failed.

$A \in B_P$ is in the SLD *finite failure set* of P if $P \cup \{\leftarrow A\}$ has a finitely failed SLD R-search space for some computation rule R.

The fixpoint characterization of the SLD finite failure set of P is straightforward. With T_P as defined in section 2.2, we define

$$\begin{aligned} T_P \downarrow 0 &= B_P \\ T_P \downarrow \beta + 1 &= T_p(T_p \downarrow \beta) \\ T_P \downarrow \lambda &= \bigcap_{\beta<\lambda} T_P \downarrow \beta,\ \lambda \text{ a limit ordinal,} \end{aligned}$$

and we have

THEOREM 11. *Let P be a program and let $A \in B_p$. Then A is in the SLD finite failure set of P if and only if $A \in B_P \backslash T_P \downarrow \omega$.*

The definition of SLD finite failure set requires only the existence of some computation rule R, which could vary according to program and goal. However, there is a class of computation rules which always work. A computation rule R is *fair* if for every SLD-derivation D via R, either 1) D is failed, or 2) for every goal G_i in D, for every atom A in G_i, A (or some instantiation of A) is eventually selected by R at some step $j > i$.

THEOREM 12. *Let R be a fair computation rule, P be a program, and $A \in B_P$. Then A is in the SLD finite failure set of P if and only if $P \cup \{\leftarrow A\}$ has a finitely failed R-search space.*

The model theoretic characterization of the SLD finite failure set of a program is less obvious since, as noted above, it never happens that $P \models \sim A$. We need to leave the confines of Horn clause logic, transforming P into a first-order statement which says explicitly that the heads of clauses are true if *and only if* the bodies are true. Since the heads of more than one clause defining the same predicate may unify with a given atom, we need to exercise a bit of care in doing the transformation. We illustrate with an example. If P is

$$p(x, f(y)) \leftarrow$$

$$p(g(x), y) \leftarrow q(x, y, z)$$

$$q(f(x), y, z) \leftarrow q(x, y, z), r(z)$$

then we transform it first to

$$p(x_1, x_2) \leftarrow (\exists x, y) x_1 = x \wedge x_2 = f(y)$$

$$p(x_1, x_2) \leftarrow (\exists x, y, z) x_1 = g(x) \wedge x_2 = y \wedge q(x, y, z)$$

$$q(x_3, x_4, x_5) \leftarrow (\exists x, y, z) x_3 = f(x) \wedge x_4 = y \wedge x_5 = z \wedge q(x, y, z) \wedge r(z)$$

and then to

$$\begin{aligned}
&[(\forall x_1, x_2) p(x_1, x_2) \leftrightarrow ((\exists x, y) x_1 = x \wedge x_2 = f(y)) \\
&\qquad \vee ((\exists x, y, z) x_1 = g(x) \wedge x_2 = y \wedge q(x, y, z))] \\
&\wedge [(\forall x_3, x_4, x_5) q(x_3, x_4, x_5) \leftrightarrow (\exists x, y, z) x_3 = f(x) \wedge x_4 = y \wedge x_5 = z \\
&\wedge q(x, y, z) \wedge r(z)] \\
&\wedge (\forall x_6) \sim r(x_6).
\end{aligned}$$

The last conjunct $(\forall x_6) \sim r(x_6)$ is added because r is not defined in P. Now we have introduced the « = » predicate symbol, which is intended to be interpreted as equality, so we need to add a set of equality axioms, which vary according to the signature of the program. For $\Sigma = \langle \mathcal{F}, \mathcal{P}, \mathrm{ar}\rangle$ the equality axioms are

1) $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ for each pair of distinct $f, g \in \mathcal{F}$, with $ar(f) = n \geq 0$, $\mathrm{ar}(g) = m \geq 0$,
2) $t \neq x$ for each $t \in TM_\Sigma$ with an occurrence of variable x
3) $f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n) \leftarrow x_1 \neq y_1 \vee \ldots \vee x_n \neq y_n$ for each $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$
4) $x = x$
5) $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \leftarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$ for each $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$
6) $(p(x_1, \ldots, x_n) \leftarrow p(y_1, \ldots, y_n)) \leftarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$ for each $p \in \mathcal{P}$, $\mathrm{ar}(p) = n$, and also for $=$.

The *completion* of Σ-program P, denoted comp(P), is the transformed version of P together with the equality axioms for Σ.

THEOREM 13. (Soundness and completeness of negation as failure) *Let P be a program, G a goal, and R a fair computation rule. Then $P \cup \{G\}$ has a finitely failed SLD R-search space if and only if comp(P) $\models G$.*

The model theoretic characterization of the finite failure set is an immediate corollary.

COROLLARY 14. *Let P be a program and $A \in B_P$. Then A is in the SLD finite failure set of P if and only if comp(P) $\models \sim A$.*

At this point we have a satisfactory characterization of negation, and we are ready to add negative literals to clauses and try to compute with the extended class of programs.

Let Σ be a signature. A *general Σ-program clause* is a clause of the form $A \leftarrow L_1, \ldots, L_n$, where A is a Σ-atom and $L_1, \ldots, L_n$, $n \geq 0$, are Σ-literals. A *general Σ-program* is a finite set of general Σ-program clauses. A *general Σ-goal* is a clause of the form $\leftarrow L_1, \ldots, L_n$, where $L_1, \ldots, L_n$, $n \geq 0$, are Σ-literals. The *completion* of a general program is given by the same transformation as for Horn clause programs.

Theorem 13 tells us that if a literal $\sim A$, possibly non-ground, is a logical consequence of program P, then we can use SLD-resolution to determine that fact. However, it tells us nothing about the answer substitutions generated by SLD-resolution, and it is precisely here that we find a serious source of trouble in computing with negation. For example, if P consists of the single clause $p(a) \leftarrow$, then for any ground or non-ground goal $\leftarrow p(t)$ such that t does not unify with a, SLD-resolution fails finitely and we can derive $\sim p(t)$. On the other hand, if t is a variable, say x, when we submit the goal $\leftarrow p(x)$, SLD-resolution succeeds with the answer substitution $\{x/a\}$, leaving us to conclude, correctly, that $\sim p(x)$ is not a logical consequence of comp(P). However, we do not get back the information that for any non-variable term t other than a, comp(P) $\models \sim p(t)$. Now, if we add $q(b) \leftarrow$ to the program and consider the goal $\leftarrow \sim p(x), q(x)$, and if we select the literal $\sim p(x)$ first, we find that $p(x)$ succeeds, $\sim p(x)$ fails, and the entire goal fails, even though $\{x/b\}$ is a correct answer substitution. In other words, we have lost completeness with respect to answer substitutions. More seriously, the loss of completeness even jeopardizes soundness, because if we add the clauses

$$r(x) \leftarrow \sim p(x), q(x)$$
$$s(x) \leftarrow \sim r(x)$$

we can now conclude $\sim r(x)$ and therefore $s(x)$, which is incorrect because comp(P) $\models \sim s(b)$. The moral is that we must be cautious concerning substitutions and negation.

We now define $SLDNF$-resolution, that is, SLD-resolution together with the negation as failure rule. Let P be a general program, G a general goal, and R a computation rule. Then an *$SLDNF$-derivation* from $P \cup \{G\}$ via R is a sequence of triples $[G_0, *, *]$, $[G_1, C_1, \theta_1]$,... such that G_0 is G, and for all i, if G_i is $\leftarrow L_1, \ldots, L_n$, and the literal L_j in G_i selected by R is

1. positive, then $[G_{i+1}, C_{i+1}, \theta_{i+1}]$ satisfies the definition of SLD-derivation;

2. negative, say $\sim A$, and if $P \cup \{\leftarrow A\}$ has a finitely failed

$SLDNF$ R-search space, then

- G_{i+1} is $\leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n$,
- C_{i+1} is *,
- θ_{i+1} is the identity substitution.

An *$SLDNF$-refutation* of $P \cup \{G\}$ via R is a finite $SLDNF$-derivation from $P \cup \{G\}$ via R in which the last goal is $\square$. An $SLDNF$-derivation via R is *failed* if it finite and, for the literal L selected by R in the last goal G_n, either

1. L is positive and does not unify with the head of any clause in P, or
2. L is negative, say $\sim A$, and there is an $SLDNF$-refutation of $P \cup \{\leftarrow A\}$ via R.

$P \cup \{G\}$ has a *finitely failed $SLDNF$ R-search space* if there are only a finite number of $SLDNF$-derivations from $P \cup \{G\}$ via R, and they are all failed.

Note that we impose the policy that $SLDNF$-resolution applied to negative literals returns only the identity substitution. We can be particularly conservative by restricting our attention to *safe* computation rules, which select negative literals only when they are ground. For safe computation rules the soundness part of Theorems 13 and 8 can be extended to general programs and goals.

THEOREM 15. *Let P be a general program, G a general goal $\leftarrow L_1, \ldots, L_n$, and R a safe computation rule.*

1. *If $P \cup \{G\}$ has a finitely failed $SLDNF$ R-search then $comp(P) \models G$.*
2. *If θ is an R-computed answer substitution for $P \cup \{G\}$ then θ is a correct answer substitution for $comp(P) \cup \{G\}$; i.e., $comp(P) \models L_1\theta \wedge \ldots \wedge L_n\theta$.*

Unfortunately, the corresponding completeness results do not hold in general. One line of research has been to find subclasses of general programs and goals for which $SLDNF$-resolution is complete. For example, in [3] a program P is defined to be *ground-categorical* if for every $A \in B_P$, either $P \models A$ or comp(P) $\models \sim A$. For a ground-categorical program P the decision problem for ground atoms, i.e., determining if $P \models A$ for $A \in B_P$, is clearly decidable by enumerating all SLD-derivations from $P \cup \{\leftarrow A\}$ via some fair computation rule R, since either there is a refutation of $P \cup \{\leftarrow A\}$ or $P \cup \{\leftarrow A\}$ has a finitely failed R-search space. Building on the concept of ground-categoricity, [3] defines a class of general programs, called *structured programs*, for which $SLDNF$-resolution is complete for ground correct answer substitutions. However, being structured is an undecidable semantic property which depends on the nature of the relations defined by a program.

Another approach has been to find decidable syntactic properties of programs which guarantee completeness of $SLDNF$-resolution. We will describe one such class of programs for which a completeness proof is given in [1]

Let P be a general program. The *dependency graph* of P is a labelled directed graph $DG(P) = \langle V, E\rangle$ where

- V is the set of all predicate symbols appearing in P, and
- $\langle p, q\rangle \in E$ if and only if there is a clause $p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m$ in P such that q is the predicate symbol in some $L_i, i = 1, \ldots, m$. $\langle p, q\rangle$ is labelled *positively (negatively)* if L_i is a positive (negative) atom.

Note that an edge can be labelled both positively and negatively. For two predicate symbols p, q which appear in P, p *depends positively (negatively)* on q in P if and only if there is a path in $DG(P)$ from p to q with an even (odd) number of edges labelled negatively. P is *stratified* if $DG(P)$ contains no cycles with at least one negative edge. P is *strict* if there is no pair p, q of predicate symbols in P such that p depends both positively and negatively on q.

P is *allowed* if for every clause C in P, each variable in C occurs at least once in a positive literal in the body of C. P satisfies the *closed derivation condition* if it is allowed and if for every clause C in P, each variable occurring in a positive literal in the body of C also occurs in the head of C.

THEOREM 16. *Let P be a strict stratified general program satisfying the closed derivation condition, and let G be a general goal and R a computation rule. If θ is a correct answer substitution for $P \cup \{G\}$ then θ is an R-computed answer substitution for $P \cup \{G\}$.*

Note that the conditions imposed on P are sufficient to force θ to be a ground substitution. Related work can be found in [6].

4. Functions.

In addition to relational programming, a declarative paradigm which has been studied deeply is functional programming, and a great deal of discussion has been generated concerning their relative strengths and weaknesses. Abstractly it is certainly true that functions andthose based on relations are interchangeable, since functions are a particular kind of relation, and relations can be represented as characteristic functions. However, divergent classical approaches to the definition and implementation of programming languages based on functions and relations result in important practical differences. For example, computation in functional languages is usually designed to determine canonical values for ground functional expressions, and so lacks the ability to find answer substitutions. One consequence is that in a typical functional language there is no analog to the possibility of using a single relation like plus, defined in section 2, to determine both that $\{z/s(s(s(0)))\}$ is a correct answer substitution for $\leftarrow\text{plus}(s(0), s(s(0)), z)$ *and* that $\{x/s(0)\}$ is a correct answer substitution for $\leftarrow\text{plus}(x, s(s(0)), s(s(s(0))))$, even though plus is normally thought of as a function.

On the other hand it is often the case that the solution to a given problem is, semantically, a function, in which case a relational syntax is an obstacle to the natural expression of the solution. For example, if we need to compute factorial, and do not need the ability to treat the input and the output symmetrically as in the plus relation, then

$$\mathrm{fact}_f(0) = 1$$
$$\mathrm{fact}_f(s(x)) = \mathrm{times}_f(s(x),\ \mathrm{fact}_f(x))$$

is clearer than

$$\mathrm{fact}_r(0, 1) \leftarrow$$
$$\mathrm{fact}_r(s(x), y) \leftarrow \mathrm{fact}_r(x, z),\ \mathrm{times}_r(s(x), z, y)$$

because it avoids, by nested function application, the unstructured device of shared variables, in this case z.

In recognition of the complementary strengths of the functional and relational styles, a growing body of work aims at finding a suitable model in which the two can usefully co-exist. A number of approaches have appeared. An early one was to embed an interpreter for one language in the interpreter of another. In the language LOGLISP ([20]), for example, an interpreter for Horn clause logic is implemented in Lisp, and when passed a goal from a Lisp function, it will return a list of one or more answer substitutions. Also, when the Horn clause interpreter is evaluating a goal, it can call the Lisp interpreter to evaluate function symbols to which Lisp functions have been bound.

More recently attempts have been made to try to find a single unified framework for expressing aspects of functional and relational programming. One approach is to stay as close as possible to the programming language foundations of Horn clause logic, namely, minimal model semantics and SLD-resolution. We will discuss one language in this category, LEAF (Logic, Equations, and Functions), described in [2].

A key to the way in which LEAF extends Horn clause logic is the ability to associate function definitions with function symbols.

LEAF partitions the usual set of function symbols into two parts: undefined *data constructor* symbols, and defined *function* symbols, so that a signature for a LEAF language is a 4-tuple $\langle \mathcal{D}, \mathcal{F}, \mathcal{P}, \mathrm{ar}\rangle$, where $\mathcal{D}$ is the set of data constructor symbols and $\mathcal{F}$ is the set of function symbols. In addition to $\mathcal{P}$ the predicate symbol =, with arity 2, is available. There are two classes of terms, *data terms* with no (defined) function symbols, and *terms*, which may have occurrences of function symbols. An atom is either an atom in the usual sense, or a *functional atom* of the form $f(t_1, \ldots, t_n) = t$, where f is a function symbol and $t, t_1, \ldots, t_n$ are terms. Function definitions are given by way of a new class of program clauses of the form

$$f(t_1, \ldots, t_n) = t \leftarrow B_1, \ldots, B_m$$

where f is a function symbol, $t_1, \ldots, t_n$ are data terms, t is a term, $B_1, \ldots, B_m$ are atoms, $m \geq 0$, and each variable in t is *functionally derived* in the clause. A variable x is functionally derived in a clause C

$$f(t_1, \ldots, t_n) = t \leftarrow B_1, \ldots, B_m$$

if either 1) x occurs in some t_i, $i = 1, \ldots, n$, or 2) x occurs in some B_j of the form $t' = x$ and all of the variables in t' are functionally derived in C. Clauses defining relations are of the form

$$p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$$

where $p \in \mathcal{P}$, where $t_1, \ldots, t_n$ are data terms, and where $B_1, \ldots, B_m$ are atoms, $m \geq 0$. A *LEAF program* is a set of clauses such that for each pair of clauses with heads $f(t_1, \ldots, t_n) = t$ and $f(u_1, \ldots, u_n) = u$, the terms $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$ are not unifiable. The restrictions imposed on function definitions are sufficient to guarantee that the relations they define are indeed functions.

A feature of LEAF which it shares with lazy evaluation functional languages (see, e.g., [25]) is the ability to define and partially construct infinite data structures. For example, in the program

```
sum_1_n(n)=nth(n,sum())←
```

nth(0,Cons(x, y)) = $x \leftarrow$ /* Cons is a data constructor */

nth($S(n)$, Cons(x, y)) =nth(n, y) $\leftarrow$ /* S is a data constructor */

sum() = accum(0, nats (0)) $\leftarrow$

accum(x, Cons(y, z)) =Cons(+(x, y), accum(+(x, y), z)) $\leftarrow$

nats(n)=Cons(n,nats($S(n)$)) $\leftarrow$

+(0, y) = y $\leftarrow$nat(y)

+($S(x)$, y) = S(+(x, y)) $\leftarrow$nat(y)

nat(0) $\leftarrow$

nat($S(x)$) $\leftarrow$nat(x)

nats(x) defines an infinite list of numbers beginning with x, and the 0-ary function sum() defines an infinite sequence $n_0, n_1, \ldots$, where $n_i = \sum_{j=1}^{i} j$. Calculating $\sum_{i=1}^{n} i$, which is the value returned by sum_1_n(n), requires only that the first $n+1$ elements of the infinite structure sum() be generated.

The model theoretic meaning of LEAF progams is given by a version of Herbrand interpretations carefully expanded to account for the call-by-name semantics of function applications. Let $\Sigma = \langle \mathcal{D}, \mathcal{F}, \mathcal{P}, \mathrm{ar} \rangle$ be a signature for a LEAF language. U_Σ, the Herbrand universe for Σ, is the set of all terms $d(t_1, \ldots, t_n)$ over $\mathcal{D} \cup \{\omega\}$, where $d \in \mathcal{D}$, ar(d) = n, such that either $n = 0$ or $t_i \neq \omega$ for some $i = 1, \ldots, n$. $\omega \notin \mathcal{D}$ is a special symbol intended to denote that an argument of a data term has not yet been evaluated. U_Σ is ordered by the reflexive relation $\leq$ defined

$$\begin{array}{ll} \omega \leq t & \text{for all } t \in U_\Sigma \\ d(t_1, \ldots, t_n) \leq d(u_1, \ldots, u_n) & \text{iff } t_i \leq u_i,\ i = 1, \ldots, n. \end{array}$$

$t \leq u$ expresses the intuitive notion that t is a possibly less completely computed approximation to u. B_Σ, the Herbrand base for Σ, is the set of all atoms of the form

- $f(t_1, \ldots, t_n) = t$, where $f \in \mathcal{F}$, ar(f) = n, and $t_1, \ldots, t_n \in U_\Sigma$

- $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$, $\mathrm{ar}(p) = n$, and $t_1, \ldots, t_n \in U_\Sigma$.

A LEAF-interpretation for Σ is a subset I of B_Σ such that

- $f(t_1, \ldots, t_n) = \omega$ is in I, for all $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$, and all $t_1, \ldots, t_n \in U_\Sigma$, and
- if $f(t_1, \ldots, t_n) = t$ and $f(u_1, \ldots, u_n) = u$ are in I, then either $t \leq u$ or $u \leq t$.

Thus in any LEAF-interpretation the meaning of terms $f(t_1, \ldots, t_n)$ is given by a consistent set of approximations.

The notions of satisfaction and model are the usual ones, and the model intersection property holds. The model theoretic semantics of a LEAF program, then, is its minimal LEAF-model. An equivalent fixpoint semantics is also defined in [2].

The operational semantics of LEAF programs is based on SLD-resolution, but there are two additional ideas. First, to put programs and goals in a suitable form so that SLD-resolution can cope with the semantics of function application, clauses go through a preprocessing «flattening» transformation to remove nested applications. A *canonical clause* is a clause in which all atoms are *canonical atoms*, where a canonical atom is of the form

- $f(t_1, \ldots, t_n) = t$, where $f \in \mathcal{F}$, $\mathrm{ar}(f) = n$, and $t_1, \ldots, t_n$ are data terms, or
- $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$, $\mathrm{ar}(p) = n$, and $t_1, \ldots, t_n$ are data terms.

We illustrate the transformation with an example. The clause

$$\mathrm{accum}(x,\ \mathrm{Cons}(y, z)) = \mathrm{Cons}(+(x, y),\ \mathrm{accum}(+(x, y), z)) \leftarrow$$

becomes

$$\mathrm{accum}(x,\ \mathrm{Cons}(y, z)) = \mathrm{Cons}(v_1, v_2) \leftarrow +(x, y) = v_1,$$

$$\mathrm{accum}(v_3, z) = v_2, +(x, y) = v_3$$

which can be simplified to

$$\text{accum}(x,\ \text{Cons}(y,z)) = \text{Cons}(v_1,v_2) \leftarrow +(x,y) = v_1,\ \text{accum}(v_1,z) = v_2.$$

Once program and goal have been put into canonical form, SLD-resolution can begin, treating the equality symbol as just another predicate, but one more inference rule is needed to account for the lazy evaluation of function application. In particular, the data term which is the value of a function application is generated only to the extent necessary to perform resolution on the atoms in which it occurs. Given an atom $f(t_1,\ldots,t_n) = t$, if the computation reaches a point where t is not a subterm of an argument to any other atom, and if the variables of t are not needed to determine the answer substitution, then the *atom elimination* inference rule allows the atom $f(t_1,\ldots,t_n) = t$ to be deleted from the current goal. A precise definition is given in [2].

Given a canonical program and goal, computation proceeds essentially by applying SLD-resolution and the atom elimination rule, with preference given to the latter when both are applicable, until the empty clause is derived. See [2] for details, as well as aspects of the language which we have not discussed here.

5. Equality.

Equality is such a fundamental relation that it is sometimes considered part of first-order logic, and so it is natural to consider how to incorporate it into Horn clause logic. In fact we have already seen two uses of equality. However, the equality theory in program completions is quite limited, since its intention is to prevent some terms from being equal as much as it is to require others to be equal. The use of equality in LEAF is motivated by the introduction of functions, and consequently the interpretation of equality is no stronger than necessary to achieve that goal. Consider for example the program repr(0) = 0 $\leftarrow$

repr($S(x)$) = +(1, repr(x)) $\leftarrow$ /*+ is a data constructor here */

which flattens to

repr(0) = 0 $\leftarrow$

repr($S(x)$) = +(1, v) $\leftarrow$ repr(x) = v.

If = were interpreted as true equality then the goal $\leftarrow$repr(x) = repr(x) should yield the identity substitution. In LEAF, however, this goal flattens to

$$\leftarrow \text{repr}(x) = v,\ \text{repr}(x) = v$$

which simplifies to $\leftarrow$repr(x) = v. Now SLD-resolution will return the sequence of answer substitutions $\{x/0\}$, $\{x/s(0)\}$, $\{x/s(s(0))\}, \ldots$.

In this section we will look at languages in which equality is used to impose additional algebraic structure on the domain of computation, so that the programmer can reason in an algebraic model rather than the unstructured minimal Herbrand model (or minimal S-model).

When SLD-resolution tries to unify (the arguments of) an atom in a goal with (the arguments of) the head of a clause, say $p(t_1, \ldots, t_n)$ and $p(u_1, \ldots, u_n)$, this process may be thought of as an attempt to find a most general solution for the equations $t_i = u_i$, $i = 1, \ldots, n$. In ordinary Horn clause logic two terms are considered equal only if they are syntactically identical, but if we require the equality relation to obey additional axioms, then non-identical terms may denote identical objects. Furthermore, more than one most general solution may exist. For example, consider the equation $g(f(x, a)) = g(f(a, x))$ in the presence of the axiom

$$f(x, f(y, z)) = f(f(x, y), z) \qquad \text{(associativity for } f\text{)}.$$

Here $\theta_1 = \{x/a\}$, $\theta_2 = \{x/f(a, a)\}$, $\theta_3 = \{x/f(f(a, a), a)\}, \ldots$ are all most general solutions.

In the presence of additional equality axioms we need to generalize the notion of most general unifier. Let T be a first-order theory over signature Σ. The relations $=$ and $\leq$ on pairs of Σ-substitutions θ, σ generalize to

$$\begin{array}{ll} \sigma =_T \theta[W] & \text{if } T \models x\sigma = x\theta \text{ for all } x \in W, \text{ where } W \subseteq \mathcal{V} \\ \sigma \leq_T \theta[W] & \text{if } \sigma =_T \theta \circ \gamma[W] \text{ for some } \Sigma\text{-substitution } \gamma. \end{array}$$

A T-*unifier* for $t, u \in TM_\Sigma$ is a Σ-substitution θ such that

$$T \models t\theta = u\theta,$$

and a *most general set* of T-unifiers for $t, u \in TM_\Sigma$ is a set U of Σ-substitutions satisfying

1. (Correctness) every $\theta \in U$ is a T-unifier for t, u;
2. (Completeness) for every T-unifier σ for t, u, there is a $\theta \in U$ such that $\sigma \leq_T \theta[\text{var}(t) \cup \text{var}(u)]$;
3. (Minimality) for all θ, $\sigma \in U$, if $\sigma \leq_T \theta[\text{var}(t) \cup \text{var}(u)]$ then $\sigma = \theta$ (i.e., σ and θ are the same element of U).

Note that if t, u are not unifiable then the empty set satisfies (1) - (3). A set of substitutions that satisfies (1) and (2) is a *complete set* of T-unifiers for t, u.

T-unification has been studied intensively in the case where T is a set of equations, in which case it is also called E-unification or universal unification, and theories are known to exist in each of the following classes.

1. *unitary*: a most general set of T-unifiers for t, $u \in TM_\Sigma$ always exists and has at most one element
2. *finitary*: a most general set of T-unifiers for t, $u \in TM_\Sigma$ always exists and has more than one but at most a finite number of elements
3. *infinitary*: a most general set of T-unifiers for t, $u \in TM_\Sigma$ always exists and is infinite for some t_0, $u_0 \in TM_\Sigma$
4. *type* 0: there are t, $u \in TM_\Sigma$ for which there is no most general set of T-unifiers

In an infinitary theory most general sets of $\mathcal{T}$-unifiers are not necessarily r.e. See [21] for more on universal unification.

Two general strategies can be identified for extending a Horn clause interpreter to handle equality. A fixed equality theory can be assumed and a specialized unification algorithm built into the interpreter, or axiomatization of equality can be made available at the program level, in which case the interpreter needs a general purpose unification algorithm. Prolog-XT ([4]), for example, is an experimental implementation of Prolog with a built-in unification algorithm ([5]) for the unitary theory of Boolean rings. In the remainder of this section we will discuss the language Eqlog ([12]), an instance of the second strategy.

Eqlog is rooted as much in the theory of abstract data structures as in logic programming, and it demonstrates the advantages of bringing together these two areas. From the former Eqlog brings the idea of axiomatizing data structures by sets of equations. The abstract behavior of a stack, for example, can be described (ignoring error conditions) by the equations

$$\text{pop}(\text{push}(x, y)) = y$$
$$\text{top}(\text{push}(x, y)) = x$$

The intended model theoretic interpretations of such equations is, as we will see, closely related to minimal Herbrand model semantics.

Abstract data type equations are typically expressed in order-sorted languages, a generalization of many-sorted languages, which are in turn a generalization of first-order languages. A *manysorted signature* is a tuple $\Sigma = \langle \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$, where $\mathcal{S}$ is a set of *sort symbols*, $\mathcal{F}$ and $\mathcal{P}$ are sets of function and predicate symbols, and the arity function ar is defined on $\mathcal{F} \cup \mathcal{P}$ and takes values in the set of finite sequences of sort symbols. We assume the availability of a countable set V_ξ of variable symbols for each sort $\xi \in \mathcal{S}$, and we denote $\bigcup_{\xi \in \mathcal{S}} V_\xi$ as $\mathcal{V}$. We also assume that $\mathcal{P}$ contains a symbol $=_\xi$ for each $\xi \in \mathcal{S}$, with $\text{ar}(=_\xi) = \langle \xi, \xi \rangle$. Many-sorted Σ-terms, Σ-formulas, Σ-theories, Σ-atoms,

Σ-literals, Σ-clauses, Horn Σ-clauses, and Horn clause Σ-programs are defined as for ordinary signatures, with the addition of the following *well-sortedness conditions.*

1. $v \in V_\xi$ has sort ξ.
2. For $f \in \mathcal{F}$, if $\text{ar}(f) = \langle \xi_1, \ldots, \xi_{n+1} \rangle$, then term $f(t_1, \ldots, t_n)$ has sort ξ_{n+1}.
3. For $f \in \mathcal{F}$, if $\text{ar}(f) = \langle \xi_1, \ldots, \xi_{n+1} \rangle$, then in term $f(t_1, \ldots, t_n)$, each t_i must have sort ξ_i, $i = 1, \ldots, n$.
4. For $p \in \mathcal{P}$, if $\text{ar}(p) = \langle \xi_1, \ldots, \xi_n \rangle$, then in atom $p(t_1, \ldots, t_n)$, each t_i must have sort ξ_i, $i = 1, \ldots, n$.

Σ-formulas of the form $t =_\xi u$ are *Σ-equations.*

An *order-sorted signature* is a tuple $\Sigma = \langle \mathcal{O}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$, where $\langle \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$ is a many-sorted signature and $\mathcal{O}$ is a set of *subsort declarations* of the form $\xi < \eta$, where $\xi, \eta, \in \mathcal{S}$. $\mathcal{O}$ determines a *subsort relation* $<_\mathcal{O}$, which is the least quasi-ordering on $\mathcal{S}$ satisfying $\xi <_\mathcal{O} \eta$ if $(\xi < \eta) \in \mathcal{O}$. Order-sorted Σ-terms, Σ-formulas, Σ-theories, Σ-atoms, Σ-literals, Σ-clauses, Horn Σ-clauses, and Horn clause Σ-programs are defined as for many-sorted signatures, except that the well-sortedness conditions are relaxed according to the following stipulation.

5. If term t has sort ξ and $\xi <_\mathcal{O} \eta$, then t also has sort η.

Note that a first-order signature can be transformed easily into a many-sorted signature with a single sort, and that a many-sorted signature $\langle \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$ can be identified with the order-sorted signature $\langle \emptyset, \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$.

The following example is a definition, in Eqlog syntax, of the data type List_of_Integers. Int is a built-in Eqlog data type. The subsort Ne_list is used to restrict the domains of head and tail to non-empty lists, that is, lists constructed by the cons function.

```
module List_of_Integers using Int is

          sorts       Int, Ne_list,List
          subsorts    Ne_list<List
          fns         nil: → List
                      head: Ne_list → Int
                      tail: Ne_list → List
                      cons: Int,List → Ne_list
          vars        I:Int, L:List
          axioms      head(cons(I, L)) = I
                      tail(cons(I, L)) = L
          endmod      List_of_Integers
```

Note that the signature can be derived from the information given.

A Σ-interpretation for order-sorted signature $\Sigma = \langle \mathcal{O}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar}\rangle$ is a pair $I = \langle \mathcal{D}, \mu\rangle$ such that

- $\mathcal{D} = \{\xi^I | \xi \in \mathcal{S}\}$, where ξ^I is a set, for all $\xi \in \mathcal{S}$, and $\xi <_\mathcal{O} \eta$ implies $\xi^I \subseteq \eta^I$
- μ is a map defined on $\mathcal{F} \cup \mathcal{P}$ such that
  - $\mu(f) : \xi_1^I \times \ldots \times \xi_n^I \to \xi_{n+1}^I$ for $f \in \mathcal{F}$, $\text{ar}(f) = \langle \xi_1, \ldots, \xi_{n+1}\rangle$
  - $\mu(p) \subseteq \xi_1^I \times \ldots \times \xi_n^I$ for $p \in \mathcal{P}$, $\text{ar}(p) = \langle \xi_1, \ldots, \xi_n\rangle$
  - $\mu(=_\xi) = \{\langle d, d\rangle | d \in \xi^I\}$.

The last condition forces the equality symbols to be interpreted as identity. An *order-sorted variable assignment* into I is a map $\alpha : \mathcal{V} \to \bigcup_{\xi \in \mathcal{S}} \xi^I$ satisfying the condition that $x \in V_\xi$ implies $\alpha(x) \in \xi^I$. Satisfaction in an order-sorted interpretation is defined as for ordinary first-order interpretations, and similarly for models and logical consequence.

Let $I = \langle \mathcal{D}_I, \mu_I\rangle$, $J = \langle \mathcal{D}_J, \mu_J\rangle$ be two interpretations for $\Sigma = \langle \mathcal{O}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar}\rangle$, and let $C_I = \cup\mathcal{D}_I$ and $C_J = \cup\mathcal{D}_J$. A Σ-*homomorphism* from I to J is a map $\phi : C_I \to C_J$ such that

- if $a \in \xi^I$ then $\phi(a) \in \xi^J$;
- if $f \in \mathcal{F}$, $\text{ar}(f) = \langle \xi_1, \ldots, \xi_{n+1}\rangle$, and $\langle a_1, \ldots, a_n\rangle \in \xi_1^I \times \ldots \times \xi_n^I$,

then

$$\phi(\mu_I(f)(a_1,\ldots,a_n)) = \mu_J(f)(\phi(a_1),\ldots,\phi(a_n));$$

- if $p \in \mathcal{P}$, $\mathrm{ar}(p) = \langle \xi_1,\ldots,\xi_n \rangle$, and $\langle a_1,\ldots,a_n \rangle \in \xi_1^I \times \ldots \times \xi_n^I$, then

$$\langle a_1,\ldots,a_n \rangle \in \mu_I(p) \text{ implies } \langle \phi(a_1),\ldots,\phi(a_n) \rangle \in \mu_J(p).$$

A bijective Σ-homomorphism in which the implication in the last condition holds in both directions is a Σ-*isomorphism*, and two Σ-interpretations I, J are *isomorphic* if there is a Σ-isomorphism from I to J.

Let P be a program over an order-sorted signature Σ. A model M of P which satisfies the *initiality* condition

for every model M' of P there exists a unique

Σ-homomorphism $\phi : M \to M'$

is an *initial model* of P. In the field of abstract data types initial models are taken as the intended model theoretic semantics of programs, based on the following

THEOREM 17. *Let Σ be an order-sorted signature, and let $\mathcal{P}$ be a Σ-program.*

1. *Initial models of P exist.*
2. *All initial models of P are isomorphic.*
3. *Let M be an initial model of P, and let A be a ground Σ-atom. Then $M \models A$ if and only if $P \models A$.*

Thus initial models are an abstraction of the concept of minimal Herbrand models. Moreover, a specific initial model directly comparable to minimal Herbrand models of ordinary Horn clause programs can be constructed from the terms over the signature associated with a program.

Let $\Sigma = \langle \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar} \rangle$ be a many-sorted signature, and let P be a Σ-program. Let $\vdash$ denote derivability in some sound and complete calculus for many-sorted Horn logic with equality (as given in [18], for example), let $\sim_P$ be an equivalence relation on ground terms of Σ defined

$$t \sim_P u \text{ if and only if } P \vdash t =_\xi u,$$

where t, u have sort ξ, and let $[t]_P$ be the equivalence class of t with respect to $\sim_P$. The *initial term model* of P is $I_P = \langle \mathcal{D}_P, \mu_P \rangle$, where

- $\mathcal{D}_P = \{\xi^P | \xi \in \mathcal{S}\}$, where $\xi^P = \{[t]_P | t$ is a ground Σ-term with sort $\xi\}$;
- $\mu_P(f)([t_1]_P, \ldots, [t_n]_P) = [f(t_1, \ldots, t_n)]_P$, where $f \in \mathcal{F}$, $\text{ar}(f) = \langle \xi_1, \ldots, \xi_{n+1} \rangle$, and t_i has sort ξ_i, $i = 1, \ldots, n$;
- $\mu_P(p) = \{\langle [t_1]_P, \ldots, [t_n]_P \rangle | P \vdash p(t_1, \ldots, t_n)\}$, for $p \in \mathcal{P}$.

A semantic construction (independent of $\vdash$) of initial term models for programs over order-sorted signatures is given in [22].

The operational semantics of Eqlog programs is given in part by SLD-resolution, up to the point of solving the equations that appear explicitly in the bodies of clauses and that are implicitly created in the process of unifying goals with clause heads. Equation solving can be handled in various ways, depending on the role that equality plays in a particular program. The simplest case is when a program can be partitioned into two sets of clauses P and E such that P is a Horn clause program with no clause heads which are equations, and E is a set of equations. In this case the solution of equations with respect to the theory E can be carried out, independently of the SLD-resolution process, by *narrowing* (see, e.g., [13]), a technique that comes from the field of term rewriting.

A *term rewriting system* R is a set of *rewrite rules* of the form $t \rightarrow u$, where t, u are terms such that $\text{var}(u) \subseteq \text{var}(t)$. The *one-step*

R-reduction relation on terms, denoted $\rightarrow_R$, is defined

$t \rightarrow_R u$ if 1). $t_1 \rightarrow t_2$ is in R;
2). σ is a substitution and s is a subterm of t such that $s = t_1\sigma$;
3). u is t with s replaced by $t_2\sigma$;

and the *R-reduction* relation, denoted $\rightarrow^*_R$, is the reflexive, transitive closure of $\rightarrow_R$. R is *confluent* if $t_1 \rightarrow^*_R t_2$ and $t_1 \rightarrow^*_R t_3$ implies there is a term t_4 such that $t_2 \rightarrow^*_R t_4$ and $t_3 \rightarrow^*_R t_4$. R is *noetherian* if there are no infinite sequences $t_1 \rightarrow_R t_2 \rightarrow_R \ldots$, and it is *canonical* if it is confluent and noetherian.

A term t is in *R-normal form* if there is no term u such that $t \rightarrow_R u$. It can be shown that in canonical systems R every term t has a unique canonical form, denoted $\text{can}_R(t)$. If E is a set of equations, then canonical term rewriting system R is a *complete set of reductions* for E if

$$E \models t = u \text{ if and only if } \text{can}_R(t) \text{ is identical to } \text{can}_R(u).$$

Since R is noetherian, the R-normal form of any two terms can be found in a finite amount of time, and so we have a decision procedure for E.

However, we need more than a decision procedure: we need a way of enumerating solutions to equations. We now generalize reduction to *narrowing* by relaxing (2) in the definition of $\rightarrow_R$. The *one-step R-narrowing* relation, denoted $\xrightarrow{\theta}_R$, is defined

$t \xrightarrow{\theta}_R u$ if 1). $t_1 \rightarrow t_2$ is a variant away from var(t) of a rule in R;
2). s is a subterm of t and θ is a most general unifier of s, t_1;
3). t' is t with s replaced by t_2 and u is $t'\theta$;

and the *R-narrowing* relation, denoted $\xrightarrow{\theta}{}^*_R$, is defined

$$t \xrightarrow{\theta}{}^*_R u \text{ if } t = t_1 \xrightarrow{\theta_1}_R t_2 \xrightarrow{\theta_2}_R \ldots \xrightarrow{\theta_{n-1}}_R t_n = u \text{ and } \theta = \theta_1 \circ \ldots \circ \theta_{n-1}.$$

THEOREM 18. *Let E be a set of equations, and let canonical term rewriting system R be a complete set of reductions for E. Let h be a symbol that does not appear in E or R, and let $R' = R \cup \{h(x, x) \rightarrow true\}$. Then $\{\theta | h(t, u) \xrightarrow{\theta}{}^{*}_{R'} true\}$ is a complete set of E-unifiers for t, u.*

Thus, if P is an Eqlog program which can be partitioned into $P' \cup E$, where P' is a Horn clause program without equations in the heads of clauses, and E is a set of equations $\{t_i = u_i | i = 1, \dots, n\}$ such that $\{t_i \rightarrow u_i | i = 1, \dots, n\}$ is canonical and is a complete set of reductions for E, then SLD-resolution plus narrowing can be used to find a complete set of correct answer substitutions for $P \cup \{G\}$ for any goal G.

Now let P be an Eqlog program which can be partitioned into $P' \cup E$ where P' is as before and E is a set of *conditional equations* of the form $t = u \leftarrow t_1 = u_1, \dots, t_n = u_n$. [11] defines a class of sets of conditional equations for which a suitable generalization of narrowing, called *conditional narrowing*, enumerates complete sets of E-unifiers.

In the general case, of course, Eqlog programs cannot be partitioned this way. In this case [12] proposes that SLD-resolution and conditional narrowing work together in a co-routining relationship, though no completeness results are given.

6. Constraints.

An answer substitution for a Horn clause program and a goal can be thought of as a set of constraints on the form of certain terms. For example, for the goal $\leftarrow p(f(x))$ the substitution $\{x/g(y)\}$ says that p is true of any term of the form $f(g(y))$. Since we take the Herbrand universe of a program as its canonical domain of computation, we are guaranteed that every object in the domain has a name, and so a (possibly infinite) set of substitutions can describe all possible solutions to a goal. Given, for example, the program

$\text{int}(0) \leftarrow$

$\text{int}(s(x)) \leftarrow \text{int}(x)$

$lt(0, s(y)) \leftarrow \text{int}(y)$

$lt(s(x), s(y)) \leftarrow lt(x, y)$

$\text{between}(x, y, z) \leftarrow lt(x, y), lt(y, z)$

the complete set of answer substitutions for the goal $\leftarrow \text{between}(0, y, s(s(s)0))))$ is $\{y/s(0)\}$, $\{y/s(s(0))\}$.

The same situation holds for Horn clause programs with equality, where all initial models are isomorphic to the initial term model. It does not hold, however, if the intended domain is uncountable. Consider the clause

$\text{between\_real}(x, y, z) \leftarrow x < y, y < z$

where the domain is the real numbers and $<$ is interpreted as usual. Then every real in the open interval (0,3) is a solution for the goal $\leftarrow \text{between\_real}(0, y, 3)$, and no countable set of substitutions to y can represent that set. On the other hand, the pair of inequalities $0 < y < 3$ does the job nicely.

The point is that by generalizing from substitution to set of constraints, where a constraint is simply an atom, we get a more expressive notion of an answer. Moreover, if the constraints are interpreted in a particular algebraic structure, then we can augment the operational semantics of a programming language with any of the specialized algorithms associated with that structure. For the goal $\leftarrow \text{between\_real}(3, y, 0)$, for example, we would prefer the answer «No» to the answer $3 < y < 0$, and, indeed, there are fast linear inequality solvers that make finding the first answer feasible.

Constraint programming languages offer yet another declarative programming paradigm, and a number of such languages have been proposed and implemented. (See [16] for a survey.) Indeed, the literature on constraint languages predates that of logic programming languages, e.g., [23]. In this section we will discuss a class of languages which combines the features of both. This class ([14]) is called $CLP(X)$, where CLP is Constraint Logic Programming and X

can be instantiated to any of a large class of structures.

To describe $CLP(\mathbf{A})$, we begin with a fixed structure $\mathbf{A}$ and an order-sorted signature $\Sigma_{\mathbf{A}} = \langle \mathcal{O}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \mathrm{ar}\rangle$ for which $\mathbf{A}$ is an interpretation. Intuitively, $\mathbf{A}$ is the structure in which constraints are to be interpreted. Now, just as each Horn clause program induces a particular signature, each $CLP(\mathbf{A})$ program induces a particular extension of $\Sigma_{\mathbf{A}}$ with new uninterpreted function and predicate symbols. Also, we can assume that associated with $\mathbf{A}$ is some uniform policy of 1) relating symbols in $\mathcal{P}$ to terms with function symbols not in $\mathcal{F}$ (for example, by considering such atoms ill-sorted), and 2) extending $\mathbf{A}$, when necessary, with new objects to interpret new terms. Therefore we extend $\Sigma_{\mathbf{A}}$ in two stages, where the first stage implements this policy.

First, let $\mathcal{F}_P$, where $\mathcal{F}_P \cap \mathcal{F} = \emptyset$, be a finite set of function symbols to appear in $CLP(\mathbf{A})$ program P. The arity of each symbol in $\mathcal{F}_P$ is fixed, except for its length, by policy, so let $\mathbf{A}_{P^-}$ be $\mathbf{A}$ augmented as necessary with new objects, and let $\Sigma_{A_{P^-}}$ be the new signature

$$\langle \mathcal{O} \cup \mathcal{O}^-, \mathcal{S} \cup \mathcal{S}^-, \mathcal{F} \cup \mathcal{F}_P, \mathcal{P}, \mathrm{ar} \cup \mathrm{ar}_P^1\rangle,$$

where $\mathcal{O}^-$ and $\mathcal{S}^-$ are determined by policy. Next, each $CLP(\mathbf{A})$ program P contributes a set of predicate symbols $\mathcal{P}_P$, where $\mathcal{P}_P \cap \mathcal{P} = \emptyset$, sorted according to ar_P^2, so let Σ_{A_P} be

$$\langle \mathcal{O} \cup \mathcal{O}^-, \mathcal{S} \cup \mathcal{S}^-, \mathcal{F} \cup \mathcal{F}_P, \mathcal{P} \cup \mathcal{P}_P, \mathrm{ar} \cup \mathrm{ar}_P^1 \cup \mathrm{ar}_P^2\rangle.$$

For example, let $\mathbf{A}$ be the real numbers with $+, ^*, =$, and $<$. There is a single sort Real, and $\Sigma_{\mathbf{A}}$ is $\langle \emptyset, \{\mathrm{Real}\}, \{+, *, 0, 1, \ldots\}, \{=, <\}, \mathrm{ar}\rangle$ where

$$\begin{array}{lll}
\mathrm{ar}(+) = \langle \mathrm{Real,Real,Real}\rangle & \mathrm{ar}(0) = \langle \mathrm{Real}\rangle \\
\mathrm{ar}(*) = \langle \mathrm{Real,Real,Real}\rangle & \mathrm{ar}(1) = \langle \mathrm{Real}\rangle \\
\mathrm{ar}(<) = \langle \mathrm{Real,Real}\rangle & \cdot \quad \cdot \\
\mathrm{ar}(=) = \langle \mathrm{Real,Real}\rangle & \cdot \quad \cdot \\
 & \cdot \quad \cdot
\end{array}$$

Let $\mathcal{F}_P$ be any set of new function symbols, say $\{f, g\}$, to appear in program P. The policy for interpreting these symbols is to create a new sort RealTerm, make Real a subsort of RealTerm (i.e., $O^- = \{\text{Real}<\text{RealTerm}\}$), and to give each symbol in $\mathcal{F}_P$ an arity of $\langle \text{RealTerm},..., \text{RealTerm}\rangle$. We leave the arity of $<$ as it is in $\Sigma_{\mathbf{A}}$, so that $f(0) < f(1)$ is not well-sorted. The domain of $\mathbf{A}_{P^-}$ is $\{\mathcal{D}_{\text{Real}}, \mathcal{D}_{\text{RealTerm}}\}$, where $\mathcal{D}_{\text{Real}}$ is the set of the reals, $\mathcal{D}_{\text{Real}} \subseteq \mathcal{D}_{\text{RealTerm}}$, and $\mathcal{D}_{\text{RealTerm}} \backslash \mathcal{D}_{\text{Real}}$ consists of objects, say finite trees, which interpret terms with symbols in $\mathcal{F}_P$.

Terms, formulas, theories, atoms, literals, clauses, Horn clauses, and programs over $\Sigma_{\mathbf{A}_P}$ are defined as for any order-sorted signature, except that the predicate symbols in the heads of clauses must come from $\mathcal{P}_P$. Atoms whose predicate symbol is in $\mathcal{P}$ are distinguished as *atomic constraints*, and *constraints* are (possibly infinite) sets of atomic constraints. For notational convenience we write clauses as $A \leftarrow C \,\square\, B_1, \ldots, B_n$, where C is a finite constraint, $\square$ is a special notation for $\wedge$, and $B_1, \ldots, B_n$ are atoms with predicate symbols not in $\mathcal{P}$.

Given an interpretation $\mathbf{A}$ for signature $\Sigma_{\mathbf{A}}$, a finite set of new function symbols $\mathcal{F}_P$, and a policy for extending $\mathbf{A}$ to $\mathbf{A}_{P^-}$, then the interpretation $\mathbf{A}_{P^-} = \langle \mathcal{D}_{\mathbf{A}_{P^-}}, \mu_{\mathbf{A}_{P^-}} \rangle$ is fixed. A *solution* in $\mathbf{A}_{P^-}$ to a constraint C is an order-sorted assignment $\alpha : \bigcup_{c \in C} \text{var}(c) \rightarrow \cup \mathcal{D}_{\mathbf{A}_{P^-}}$ such that $\mathbf{A}_{P^-}, \alpha \models c$ for all $c \in C$. A constraint is *solvable* in $\mathbf{A}_{P^-}$ if it has a solution in $\mathbf{A}_{P^-}$. Element $d \in \cup \mathcal{D}_{\mathbf{A}_{P^-}}$ is *(finitely) definable* if there is a (finite) constraint C, a distinguished variable $x \in \bigcup_{c \in C} \text{var}(c)$, and a solution α of C in $\mathbf{A}_{P^-}$ such that $\alpha(x) = d$. If $\alpha(x) = d$ for all solutions α of C in $\mathbf{A}_{P^-}$ then d is *uniquely definable*, and if it uniquely definable but only by infinite constraints then it is a *limit element*. $\mathbf{A}_{P^-}$ is *solution compact* if it satisfies

1. every $d \in \cup \mathcal{D}_{\mathbf{A}_{P^-}}$ is uniquely definable
2. if $d \in \cup \mathcal{D}_{\mathbf{A}_{P^-}}$ is a limit element uniquely defined by constraint C_d, then for any constraint C, the constraint $C \cup C_d$ is not solvable in $\mathbf{A}_{P^-}$ if and only if there is a finite constraint C'_d defining d such that $C \cup C'_d$ is not solvable in $\mathbf{A}_{P^-}$.

The results in [14] apply to all solution compact structures $\mathbf{A}_{P-}$, so we restrict our attention to such structures. Henceforth in this section let $\mathbf{A}$ be an interpretation of $\Sigma = \langle \mathcal{O}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \text{ar}\rangle$, let $\mathcal{F}_P$ be a finite set of function symbols such that $\mathbf{A}_{P-} = \langle \mathcal{D}_{\mathbf{A}_{P-}}, \mu_{\mathbf{A}_{P-}}\rangle$ is solution compact, and let P be a program over

$$\Sigma_{\mathbf{A}_P} = \langle \mathcal{O} \cup \mathcal{O}^-, \mathcal{S} \cup \mathcal{S}^-, \mathcal{F} \cup \mathcal{F}_P, \mathcal{P} \cup \mathcal{P}_P, \text{ar} \cup \text{ar}_P\rangle.$$

A $CLP(\mathbf{A})$*-interpretation* of $\Sigma_{\mathbf{A}_P}$ is any order-sorted interpretation of $\Sigma_{\mathbf{A}_P}$ which extends $\mathbf{A}_{P-}$. Since the only information left unspecified in $\mathbf{A}_{P-}$ is the interpretation of symbols in $\mathcal{P}_P$, we can identify each $CLP(\mathbf{A})$-interpretation of $\Sigma_{\mathbf{A}_P}$ with a subset of the $\mathbf{A}_P$*-base*

$$\begin{aligned} B_{\mathbf{A}_P} = \{p(\alpha(x_1), \ldots, \alpha(x_n)) | p \in \mathcal{P}_P, \text{ar}_P(p) = \langle \xi_1, \ldots, \xi_n\rangle, &\text{ and} \\ \alpha : \{x_1, \ldots, x_n\} \to \cup \mathcal{D}_{\mathbf{A}_{P-}}& \\ \text{is an order-sorted assignment}&\}. \end{aligned}$$

The definition of satisfaction in a $CLP(\mathbf{A})$-interpretation is given by the definition for order-sorted interpretations, and similarly for $CLP(\mathbf{A})$ models.

The $CLP(\mathbf{A})$ models of program P enjoy the model intersection property, so we can give the model theoretic semantics of P as the least $CLP(\mathbf{A})$ model of P, denoted $M_{\mathbf{A}_P}$. We can also give a fixpoint characterization of $M_{\mathbf{A}_P}$ by way of the function $T_{\mathbf{A},P} : 2^{B_{\mathbf{A}_P}} \to 2^{B_{\mathbf{A}_P}}$, defined

$$\begin{aligned} T_{\mathbf{A},P}(I) = \{p(d_1, \ldots, d_n) \in B_{\mathbf{A}_P} | \\ &1).\ p(t_1, \ldots, t_n) \leftarrow C \,\square\, B_1, \ldots, B_m \text{ is a clause in } P; \\ &2).\ \alpha : \bigcup_{i=1}^{n} \text{var}(t_i) \to \cup \mathcal{D}_{\mathbf{A}_{P-}} \text{ is an order-sorted assignment;} \\ &3).\ \alpha(t_i) = d_i, i = 1, \ldots, n; \\ &4).\ \mathbf{A}_{P-}, \alpha \models c, \text{ for all } c \in C; \\ &5).\ B_i \text{ is } p_i(u_1, \ldots, u_{m_i}) \text{ and} \\ &\qquad p_i(\alpha(u_1), \ldots, \alpha(u_{m_i})) \in I, i = 1, \ldots, m\}. \end{aligned}$$

Defining

$$\begin{aligned} T_{\mathbf{A},P} \uparrow 0 &= \emptyset \\ T_{\mathbf{A},P} \uparrow \beta + 1 &= T_{\mathbf{A},P}(T_{\mathbf{A},P} \uparrow \beta) \\ T_{\mathbf{A},P} \uparrow \lambda &= \bigcup_{\beta<\lambda} T_{\mathbf{A},P} \uparrow \beta, \ \lambda \text{ a limit ordinal,} \end{aligned}$$

we have

THEOREM 19. $M_{\mathbf{A}_P} = lfp(T_{\mathbf{A},P}) = T_{\mathbf{A},P} \uparrow \omega$.

The operational semantics of $CLP(\mathbf{A})$ programs can be given based on SLD-resolution, where the result of a computation is a constraint which exactly characterizes a set of solutions to a goal. See [14] for details.

REFERENCES

[1] Baratella S., Filè G., *A completeness result for SLDNF resolution,* Proc. GULP-89, 85-98.

[2] Barbuti R., Bellia M., Levi G., Martelli M., *LEAF: A language which integrates logic, equations, and functions,* in [9].

[3] Barbuti R., Martelli M., *Completeness of the SLDNF-resolution for a class of logic programs,* Proc. 3rd Int. Conf. on Logic Programming, London, (1986), 600-614.

[4] Büttner W., *personal communication.*

[5] Büttner W., Simonis H., *Embedding boolean expressions into logic programming,* Journal of Symbolic Computation, **4** (1987), 191-206.

[6] Cavedon L., Lloyd J.W., *A completeness theorem for SLDNF-resolution,* Comp. Sci. Dept. technical report CS-87-06, Univ. of Bristol, 1987. (To appear in Journal of Logic Programming.)

[7] Clark K.L., Tärnlund S.A., *Logic Programming,* Academic Press, eds., 1982.

[8] Davis M., *Eliminating the irrelevant from mechanical proofs,* Proc. Symp. Appl. Math., **15** (1963), 15-30.

[9] DeGroot D., Lindstrom G., *Logic Programming: Functions, Relations, and Equations,* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[10] Falaschi M., Levi G., Martelli M., Palamidessi C., *Declarative modelling of the operational behavior of logic languages,* Theoretical Computer Science, **70** (1989).

[11] Giovannetti E., Moiso C., *A completeness result for E-unification algorithms based on conditional narrowing,* Proc. Foundations of Logic and Functional Programming, LNCS 306, Springer-Verlag, Berlin, 1988.

[12] Goguen J.A., Meseguer J., *EQLOG: Equality, types, and generic modules for logic programming,* in [9].

[13] Hullot J.M., *Canonical forms and unification,* 5th Conf. on Automated Deduction, LNCS, Springer-Verlag, Berlin, 1980.

[14] Jaffar J., Lassez J.L., *Constraint logic programming,* Conf. Proc. Principles of Programming Languages, Munich, 1987.

[15] Knight K., *Unification: a multidisciplinary survey,* Computing Surveys, **21** (1989), 93-124.

[16] Leler W., *Constraint Programming Languages,* Addison-Wesley, Reading, MA, 1988.

[17] Lloyd J.W., *Foundations of Logic Programming,* 2nd ed., Springer-Verlag, Berlin, 1987.

[18] Padawitz P., *Computing in Horn Clause Theories,* Springer-Verlag, Berlin, 1988.

[19] Robinson J.A., *A machine-oriented logic based on the resolution principle,* JACM, **12** (1965), 23-41.

[20] Robinson J.A., Sibert E.E, *LOGLISP: Motivation, design, and implementation,* in [7].

[21] Siekmann J.H., *Unification Theory,* Journal of Symbolic Computation, **7** (1989), 207-274.

[22] Smolka G., *Order-sorted Horn logic: semantics and deduction,* SEKI report SR-86-17, University of Kaiserslautern, 1986.

[23] Sutherland I., *SKETCHPAD: A man-machine graphical communication system,* Proc. IFIPS Spring Joint Comp. Conf., 1965.

[24] Tarski A., *A lattice-theoretical fixpoint theorem and its applications,* Pacific J. Math., **5** (1955), 285-309.

[25] Turner D.A., *Miranda - a non-strict functional language with polymorphic types,* Proc. Conf. on Functional Programming Languages and Computer Languages, LNCS 201, Springer-Verlag, Berlin 1985.

Dipartimento di Matematica
Università di Catania
Catania (Italy)