Logic Programming with Bounded Quantifiers

Andrei Voronkov*
ECRC
Arabellastr.17
8000 Munich 81
Germany
voronkov@ecrc.de

Abstract

This paper describes an extension of Horn clause logic programs by bounded quantifiers. Bounded quantifiers had been extensively used in a part of mathematical logic called theory of admissible sets [2]. Later some variants of bounded quantifiers had been introduced in logic programming languages [12, 19, 21, 9, 6, 7]. We show that an extension of logic programs by bounded quantifiers has several equivalent logical semantics and is efficiently implementable using a variant of SLD-resolution, which we call SLDB-resolution. We give examples showing that introduction of bounded quantifiers results in a high level logical specification language. An expressive power of subsets of Horn clauses and subsets of logic programs with bounded quantifiers is compared. We also show that the use of bounded quantifiers sheds new light on classical negation in logic programming.

1 Introduction

Although Horn clause programs are sufficiently rich to express all computable predicates, they are not sufficiently expressive to naturally represent some relations which are easily expressed in richer languages, for example in full first order logic. Since the publication of early papers on logic programming and on Prolog there has been many attempts to extend Horn clauses programs in various ways. Examples of practical extensions are numerous in different implementations of Prolog. These are usually control primitives like cut or built-in primitives. Among more theoretical extensions are programming with full first order logic, higher order logic programming, etc.

There are many foundational problems with extensions of Horn clauses. The practical solutions are mostly non-logical which means that they have no natural logical semantics. As a consequence, such programs are more difficult to understand and to verify. The problem with more theoretical extensions is that most of them cannot be efficiently implemented. In most cases the inefficiency is inherent – for example negation combined

*On leave from the International Laboratory of Intelligent Systems (SINTEL), 630090, Universitetski Prospect 4, Novosibirsk 90, Russia.

with recursion leads to non-computable predicates. Similar problem arise when using universal quantifiers.

Here we present an extension of logic programming with *bounded quantifiers* – i.e. quantifiers over finite domains. We prove that this extension can be efficiently implemented. Moreover we show on some practical examples that bounded quantifiers can be used in practical cases to express iterative algorithms and to specify the exhaustive search over finite domains. Our extension is logical in the sense that it enjoys a complete and sound model-theoretical semantics while still being efficiently implementable.

In our presentation we also handle built-in domains, so our semantics can in principle be easily generalized to a semantics of constraint logic programming as well. The motivation to consider built-in domains with built-in predicates defined on these domains was to handle correctly non-toy examples (and in particular not to define natural numbers via the constant 0 and the function symbol s).

In fact we extend the usual model-theoretical semantics of logic programming in three ways. The first extension consists of considering built-in predicates as well. The second extension is the introduction of sorts. And the third extension is the introduction of bounded quantifiers.

Intuitively, bounded quantifiers are quantifiers ranging over some finite domains, in particular over finite lists or sets. Expressions containing bounded quantifiers give natural and elegant examples of executable specifications. Consider, for example, the specification of disjoint sets

$\mathtt{disjoint}(\mathtt{S}_1,\mathtt{S}_2)$ iff $(\forall \mathtt{x}_1 \in \mathtt{S}_1)(\forall \mathtt{x}_2 \in \mathtt{S}_2)\mathtt{x}_1 \neq \mathtt{x}_2$

This specification implies the obvious way to check if two given sets are disjoint. Similar uses of the bounded quantifiers can be found already in the language SETL [19] in which the concept of a set is a first-class concept. It seems very natural to use such expressions in logic programs as well. Combining the technique of finite search with the logic programming techniques (a variant of SLD-resolution and unification) allows one to use such specifications for constructing sets with given properties. The use of such specifications also makes logic programming more logical.

Although the semantics of the above expression is quite clear, its usual representation in Horn clause logic programming

```
disjoint([],S).
disjoint([A|As],S) :-
   nonmember(A,S),
   disjoint(As,S).

nonmember(A,[]).
nonmember(A,[B|Bs]) :-
   A≠B,
   nonmember(A,Bs).
```

is in contrast not so easy to understand. The Prolog program for `disjoint` also lacks in elegance compared to the specification using bounded quantifiers.

This paper is concerned with the logical justification of logic programming with bounded quantifiers. We restricted our attention only to two types of the bounded quantifiers, originally introduced in [12], but our techniques is quite general and is also applicable to another kinds of bounded quantifiers.

In this section we give the basic definitions of sorts, lists and generalized logic programs. Some natural examples of specifications using bounded quantifiers are given in section 2.4 In section 2 we introduce several semantics of logic programs with bounded quantifiers and prove their equivalence. Section 3 describes the procedural semantics of the language, which generalizes SLD-resolution. A unification algorithm for sorted expressions is also given. In section 5 we discuss the expressive power of the language with bounded quantifiers. In general it is equivalent to the expressive power of the Horn clauses – both kinds of languages can express exactly all computable predicates in the least model semantics . First in section 5.1 we show a natural translation from the language with bounded quantifiers to Horn clauses. This translation can for example be used to automatically obtain from the above definition of `disjoint` with bounded quantifiers a definition of `disjoint` in the language of Horn clauses. Then we construct a non-recursive metainterpreter for Horn clause programs, written in the language with bounded quantifiers. Section 6 is concerned with the use of negation in logic programs. Finally, in section 7 we discuss some other possible applications of logic programming with bounded quantifiers.

1.1 Related work

Bounded quantifiers were considered among others in the following papers[1] [12, 13, 14, 9, 10, 6, 7]. Also related are papers on introducing set constructs in logic programming [3, 9, 10, 8]. For us the main motivation was the series of papers on Σ-programming [12, 13, 14] and some of our results on semantics of Σ-programs and on a translation of Σ-programs into logic programs [22, 26, 27]. Bounded quantifiers were already introduced in the first of the above-mentioned papers [12] which was inspired by the Kripke-Platek formalization of the theory of admissible sets [2]. However the authors could not find a satisfiable procedural semantics for their language. Later ideas for defining such a semantics were introduced in our paper [26] based on the translation of Σ-programs to Horn clause programs described in [22, 27] (some ideas could even be found in even earlier preprints [24, 25], written in 1985).

Later bounded quantifier were introduced in [9, 10] with the purpose of enriching logic programming languages with sets. But the absence of the set constructor in Kuper's language leads to problems with the procedural semantics of his language. Set constructor was introduced later [6] where a more satisfiable procedural semantics for a logic programming language with finite sets had been constructed. However sets are not easy to handle: the unification problem for finite sets is NP-complete [6]. The language presented in our paper has a procedural semantics comparable with SLD-resolution for Horn clauses. As we try to show, most of the applications of logic programming with sets are easily expressed in our language.

[1]Some kind of bounded quantifiers had been implemented in the seventies in a Prolog-like language developed in Hungary (We do not have any exact references).

Bounded quantifiers were also considered in constraint logic programming [7]. In most of the programming literature they are called restricted quantifiers, but bounded quantifiers introduced earlier in the mathematical literature (e.g. [2]) seem to capture better the idea of the search on finite domains.

Bounded quantifiers are usually quantifiers over finite domains represented by lists or sets. In the papers [12, 13, 14, 9, 10] lists and sets are considered as a superstructure of the usual Herbrand universe, which prohibits using terms like f(`[]`), since `[]` is a list. In our paper we introduce a flexible sort structure which allows to treat lists as an ordinary sort. In [6] function symbols are allowed also to have set arguments. In that paper sorts are not introduced but they are used in the unification algorithm, and without sorts the procedural semantics from [6] becomes incorrect.

The proof theory is different in different approaches. In [12] an analog of Kripke-Platek theory for admissible sets [2] called GES is used for proving properties of Σ-programs. In [9] the underlying proof system is the calculus with the extensionality axiom `∀z(z∈x≡z∈y)⊃x=y`. We can use at least two types of calculi to provide a proof theory for our language. The first is an analog of GES, which treats lists as a special kind of objects with induction axioms for lists. The second is a theory of inductive definitions, which seems more flexible for proving properties of programs. However to prove completeness of our semantics any of the two theories is convenient.

Our approach to defining a procedural semantics for our language, which we call SLDB-resolution, is more efficient compared to the cited papers. In [6] an exponential unification algorithm for sets is used. In [9] no satisfiable procedural semantics is provided. For example given the program

```
q :- (∀x∈Y)fail.
```

and the query `?-q`. Kuper's system has to make a substitution `[Y←{}]` during unification of `q` from the query with `q` from the head of the clause. In [12] the proposed procedural semantic comprises the exhaustive search over infinite universes. According to their first papers, the answer to the query `?-X=5`, where `X` ranges over rational numbers, should be found by the exhaustive search for the substitution for `X` over all rational numbers. In the papers following [13, 14] it was noted that in some cases the search procedure can use other strategies, but in general case their strategy includes the exhaustive search over infinite domains.

However both [9] and [6] have other motivations and for some applications finite sets could be more appropriate than lists used in our paper.

Also similar to bounded quantifiers are some of Zermelo-Fraenkel set theory expressions of the functional programming language Miranda [21].

2 Logic programs with bounded quantifiers

In this section we introduce the main notions of the paper. In our presentation of generalized logic programs sorts are assigned to terms. In section 2.1 we introduce sort definitions and their semantics. In section 2.2 we define an instance of sorts called lists. Lists are used in defining bounded quantifiers and generalized logic programs in section 2.3. We give some examples in section 2.4 to show the practical importance of bounded quantifiers.

2.1 Sorts

Consider a simple expression $(\forall x{\in}l)\varphi(x)$ containing a bounded quantifier. Intuitively this expressions means that for every element x of the list l $\varphi(x)$ holds. If such an expression occurs in a query, we need to check that l is a list. However according to the logic programming philosophy l may be any term, for example a variable. Of course we can call a predicate stating that l is a list or a predicate generating all lists each time when such a query is posed. But it would be extremely inefficient and it would obscure the semantics of our language. So we need to distinguish lists from all other elements. To this end we introduced sorts in our language. Sorts are also needed to correctly handle built-in predicates.

The use of non-sorted structures for our purposes is inefficient and leads to some semantic problems. For example, in [12, 14] it is not clear what version of a (many sorted) predicate calculus is used in the list theory GES introduced there, which makes some considerations quite obscure. The operational semantics of the extended logic programming language introduced in [12, 13] is based on model theory and therefore comprises the exhaustive search over an infinite universe. The sort structure introduced here helps to provide an efficient operational semantics for the language with bounded quantifiers. The possibility of unification-based operational semantics was noted in earlier papers [22, 27], but in these papers we used an algorithm verifying if the terms from the binding expressions are lists, which is inefficient in general.

Lets us now give formal definitions. The set of sorts consists of <u>predefined sorts</u> and <u>defined sorts</u>. Defined sorts can be introduced via sort definitions. Suppose that we have two alphabets **Symb** and **Sort** for symbols and sorts and **P** is a distinguished subset of **Sort** (the subset of predefined sorts). Let **Dsort** (defined sorts) stands for **Sort**$\setminus$**P**. Then the sort definitions **Def** are defined as

$$\mathbf{Def} := \mathbf{Symb} \in \mathbf{Dsort} \mid \mathbf{Symb} \in \mathbf{Sort} \times \ldots \times \mathbf{Sort} \to \mathbf{Dsort} \mid \mathbf{Sort} \subseteq \mathbf{Dsort}$$

The intuitive semantics of the sort definitions is the following. A sort expression of the form $\mathtt{Symb}{\in}\mathtt{Sort}$ means that $\mathtt{Symb}$ is a constant of the sort $\mathtt{Sort}$. The expression $\mathtt{Symb}{\in}\mathtt{Sort}_1\times\ldots\times\mathtt{Sort}_n \to\mathtt{Sort}$ means that $\mathtt{Symb}$ is a function symbol of the sort $\mathtt{Sort}_1\times\ldots\times\mathtt{Sort}_n \to\mathtt{Sort}$ and $\mathtt{Sort}_1 \subseteq\mathtt{Sort}_2$ means that each term of the sort $\mathtt{Sort}_1$ is also a term of the sort $\mathtt{Sort}_2$.

Below are some examples of the sort definitions:

```
nil∈Atom
Integer⊂Atom
String⊂Atom
nil∈List
cons∈List×List→List
cons∈Atom×List→List
nil∈Words
cons∈Atom×Words→Words.
```

Here `Integer, Strings` are predefined sorts.

From now on we assume that a finite set **P** of predefined sorts and a finite set **D** of sort definitions are fixed.

The idea to restrict variables to sorts has also some semantic advantages. If the sorts are fixed then the semantics of programs is not sensitive to extensions of the Herbrand Universe. Such sensitivity may cause problems, e.g. in the semantics of negation as failure in logic programming.

The definition of terms of a sort α is straightforward from the definition of sorts. We assume that for each sort α we have a countable set C_α of variables of this sort. Let for each predefined sort α C_α denotes the set of all constants of this sort. The set $Term_\alpha$ of terms of the sort α is defined as follows:

1. If the sort α is predefined then $Term_\alpha = V_\alpha \cup C_\alpha$;

2. $V_\alpha \subseteq Term_\alpha$;

3. For every sort definition in **D** of the form $c \in \alpha$, we have $c \in Term_\alpha$;

4. For each definition of the form

   $$f \in \alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha$$

   and for each $t_1 \in Term_{\alpha_1}, \ldots, t_n \in Term_{\alpha_n}$, we have $f(t_1, \ldots, t_n) \in Term_\alpha$;

5. For each definition of the form $\alpha \subseteq \beta$, if $t \in Term_\alpha$, then we have $t \in Term_\beta$.

A term t or formula φ is ground iff it contains no free variables.

If a set D of sort definitions contains a definition of one of the following forms: $f \in \alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha$ or $c \in \alpha$, or $\beta \subseteq \alpha$, we shall say that α is defined in D.

We assume that for each predefined sort α there is at least one constant in C_α.

Let $HU(\alpha)$ (the Herbrand universe for the sort α) be the set of all ground terms of the sort α.

From now on we impose a restriction on sort definitions: for any sort α, $HU(\alpha)$ must be non-empty. This property of the sort definitions is easy to check.

The notion of sorts introduced here is sufficiently rich to express many interesting domains. A term can belong to several sorts. However this sort structure admits an efficient unification algorithm. The set of the most general unifiers can be infinite. If the set of basic (predefined) sorts is closed under intersection, then two terms can have only finite (up to renaming) number of the most general unifiers.

The domain definitions of PDC-Prolog [17] are very similar to the sort definitions from this paper. The main difference is that we allow function symbols and constants to belong to several sorts.

2.2 Lists

In this paper we shall use the notion of a (many-sorted) *model* in the model-theoretical sense (see e.g. [5]), i.e. as sets with some constants, functions and predicates on this sets. The idea of an interpretation for built-in predicates is introduced via basic models. A basic model $\mathcal{M}$ is a model with the following restrictions:

1. There are no function symbols in the signature of $\mathcal{M}$;

2. Each element of $\mathcal{M}$ is the value of one and only one constant from the signature of $\mathcal{M}$.

The basic models are needed to define interpretations for built-in domains. Many practical built-in domains satisfy this properties, for example integers and strings in programming languages may have a unique representation as suitable constants. In what follows we shall refer to the sorts, predicates etc. of basic models as predefined sorts, predefined predicates etc.

Among all possible sorts we shall distinguish the sort of lists in the following way. Let $\mathcal{M}$ be a basic model, D a set of sort definitions such that `List`, `Atom` and `Univ` are not defined in D and there is no definition of the binary function symbol `cons`. (Note. Each element of the model $\mathcal{M}$ can belong to several sorts). Let $\beta_1, \ldots, \beta_n$ are all sorts which are in the model $\mathcal{M}$ or are defined by D. Then $L(\mathcal{M}, D)$ will denote the following set of sort definitions:

$\mathtt{nil} \in \mathtt{Atom}$
$\beta_1 \subseteq \mathtt{Atom}$
...
$\beta_n \subseteq \mathtt{Atom}$
$\mathtt{nil} \in \mathtt{List}$
$\mathtt{Atom} \subseteq \mathtt{Univ}$
$\mathtt{List} \subseteq \mathtt{Univ}$
$\mathtt{cons} \in \mathtt{Univ} \times \mathtt{List} \to \mathtt{List}$

Thus we allow all built-in and defined elements to be atoms and we add the constant `nil` to atoms. `Univ` is a universal sort comprising both atoms and lists. Lists are built from atoms and lists by applying the list constructor `cons`.

Instead of writing `nil` and `cons` we shall adopt the standard Prolog notation:

`[]`	stands for	`nil;`
`[s|t]`	stands for	`cons(s,t);`
$[\mathtt{s}_1, \ldots, \mathtt{s}_n \vert \mathtt{t}]$	stands for	$\mathtt{cons}(\mathtt{s}_1, \ldots, \mathtt{cons}(\ldots, \mathtt{cons}(\mathtt{s}_n, \mathtt{t}) \ldots))$;
$[\mathtt{s}_1, \ldots, \mathtt{s}_n]$	stands for	$\mathtt{cons}(\mathtt{s}_1, \ldots, \mathtt{cons}(\ldots, \mathtt{cons}(\mathtt{s}_n, \mathtt{nil}) \ldots))$.

<u>Lists</u> are elements of $HU(\mathtt{List})$. We do not define here lists as a superstructure of the ordinary terms as it has been done in [Goncharov 85] following the theory of admissible sets [Barwise 75]. Such a superstructure is convenient for more theoretical purposes (to distinguish sets from urelements in the theory of admissible sets), but from the viewpoint of programming it has some disadvantages. In particular, it forbids to use terms with subterms containing lists, e.g. f(`[]`,`[a]`). Our sorts allow to use such terms. For example, we can define the sort of such a function f via e.g.

$f \in \mathtt{List} \times \mathtt{List} \to \mathtt{Pairs}$

We can introduce the function `cons` on lists in a natural way. We define also two relations $\in$, $\sqsubseteq$ on lists as follows:

$\mathtt{x} \in [\mathtt{y}_1, \ldots, \mathtt{y}_n]$ iff for some $\mathtt{y} = 1, \ldots, n$ we have $\mathtt{x} = \mathtt{y}_i$;
$\mathtt{x} \sqsubseteq [\mathtt{y}_1, \ldots, \mathtt{y}_n]$ iff $\mathtt{x} = []$ or for some $i = 1, \ldots, n$ we have $\mathtt{x} = [\mathtt{y}_i, \ldots, \mathtt{y}_n]$.

2.3 Bounded quantifiers and generalized logic programs

Here we introduce bounded quantifiers. Apart from sorts they are basically equivalent to bounded quantifiers introduced in [12].

Bounded quantifiers are expressions of the form $(\forall$x$\in$t$)$, $(\exists$x$\in$t$)$, $(\forall$x$\sqsubseteq$t$)$, $(\exists$x$\sqsubseteq$t$)$, where the variable `x` is of the universal sort `Univ` and does not occur in the term `t` of the sort `List`.

A Σ-formula is any formula constructed from atoms using $\wedge$, $\vee$, $\exists$, and bounded quantifiers. To distinguish the ordinary existential quantifier $\exists$ from bounded quantifiers we shall call the former unrestricted existential quantifier. Δ_0-formula is a Σ-formula containing no occurences of the unrestricted quantifier $\exists$.

A generalized logic program $\mathcal{P}$ is a triple $\langle \mathcal{M}, \mathbf{D}, \mathbf{C}\rangle$ where $\mathcal{M}$ is a basic model with a signature σ, $\mathbf{D}$ a set of sort definitions, $\mathbf{C}$ a set of clauses of the form

$$P_i(\bar{x}_i) \colon -\varphi_i(\bar{x}_i),$$

where $i = 0, \ldots, n$, P_i are predicate symbols which are not in the signature σ, φ_i are Σ-formulas with free variables from $\bar{x}_i$ in the following signature δ:

> the constants of δ consist of `nil`, of the constants from $\mathcal{M}$, and of the constants defined in $\mathbf{D}$;
> the function symbols of δ are `cons` and the function symbols defined in $\mathbf{D}$;
> the predicate symbols of δ are $P_0, \ldots, P_n$,= and the predicate symbols of σ.

To make the generalized logic programs shorter and to make the syntax closer to that of the ordinary logic programs we shall introduce some notations. The set of expressions

$P(\bar{t}_1)$:- φ_1
...
$P(\bar{t}_n)$:- φ_n

will denote the clause

$$P(\bar{x}) \texttt{ :- } \exists \bar{y}_1(\bar{x} = \bar{t}_1 \wedge \varphi_1) \vee \ldots \vee \exists \bar{y}_n(\bar{x} = \bar{t}_n \wedge \varphi_n),$$

where $\bar{x}$ are new variables, $\bar{y}_i$ are all the variables of t_i.

Query to a generalized logic program $\mathcal{P}$ is any Σ-formula.

In what follows we assume that a generalized logic program $\mathcal{P} = \langle \mathcal{M}, \mathbf{D}, \mathbf{C}\rangle$ is fixed.

In the literature on logic programming with sets only one kind of bounded quantifiers is used, namely $\forall\in$. The quantifier $\forall\sqsubseteq$ was introduced in [12]. This quantifier is very expressive. For example in section 5.2 we present a non-recursive metainterpreter for Horn clause logic programs, which uses only unrestricted existential quantifiers and the bounded quantifier $\forall\sqsubseteq$. It is difficult to introduce such bounded quantifier for sets, because there is no analog of the relation $\sqsubseteq$ on sets.

2.4 Examples

To explain the use of the bounded quantifiers we give some examples below. The interesting property of these examples is that they are not recursive.

Note. In all examples of this paper we use the standard Prolog notation: variable names start from the upper-case letters, while all other symbols start from the lower-case letters.

Example 2.1 *A program computing whether a list L is ordered:*

```
ordered(L) :-
   (∀X⊑L)(X=[]∨singleton(X)∨ordered2(X)).

singleton([X]).

ordered2([X,Y|Z]) :-
   X≤Y.
```

We assume that $\leq$ is a built-in relation.

Example 2.2 *A program computing whether all elements of a list* `L` *satisfy a property* `p`*:*

```
allp(L) :-
   (∀X∈L)p(X).
```

Example 2.3 *A program computing the subset relation:*

```
subset(L1,L2) :-
   (∀X1∈L1)(∃X2∈L2)X1=X2.
```

Example 2.4 *A program verifying whether a given list* `L` *has no repetitions:*

```
norep(L) :-
   (∀X⊑L)norep1(X).

norep1([A|As]) :-
   (∀X∈As)A≠X.
```

Here `X1≠X2` is a built-in relation which is true iff `X1` and `X2` are not equal.

Example 2.5 *A program finding a route in a graph. We assume that vertices* `a` *and* `b` *are connected in the graph iff the fact* `arc(a,b)` *is in the program.*

```
route(A,B,C) :-
   path(C)∧start(A,C)∧finish(B,C).

path(P) :-
   (∀S⊑P)(S=[]∨singleton(S)∨connected(S)).

singleton([X]).
```

```
connected([X,Y|Z]) :-
   arc(X,Y).

start(X,[X|Xs]).

finish(X,Xs) :-
   (∃S⊑Xs)(S=[X]).
```

3 Semantics of generalized logic programs

The theory of logic programming is based on the fundamental fact that the (declarative) model-theoretical semantics coincides with the provability by SLD-resolution and some other kinds of provabilities, e.g. those of intuitionistic and classical logics.

In this section we shall adapt these semantics to generalized logic programs. The features specific to generalized logic programs are sorts, built-in predicates and bounded quantifiers. The semantics of Horn clause programs must be modified so as to accommodate these features.

In section 3.1 we introduce a model theoretical semantics for generalized logic programs, which simply expresses the intended declarative meaning of programs with bounded quantifiers. In section 3.2 a least fixedpoint semantics is introduced which serves as a bridge between the declarative model-theoretical semantics and the procedural interpretation introduced later. Then, in sections 3.3 and 3.4, we define two types of classical and constructive provabilities — one with axioms for lists from [12], and another one which considers lists as elements generated by inductive definitions.

Section 3.5 presents so called natural semantics of the generalized logic programs which was originally introduced in [26]. The calculus $\mathbf{Nat}(\mathcal{P})$ from this section represents in a declarative way ideas from the procedural semantics introduced in section 4.

3.1 Model-theoretical semantics

The main semantics of generalized logic programs is the model theoretical semantics which allows a declarative reading of programs. Our semantics is similar to the semantics introduced in the other papers [12, 9, 6]. The main difference between our semantics and that of [12] is that we allow function symbols to be constructors, which means that we can define new terms, whereas in [12] the model is fixed - the set of all lists with atoms from the basic model. Our semantics also treats built-in sorts, while both [9] and [6] do not consider built-in sorts.

To introduce the model-theoretical semantics of our programs we have to define the meaning of quantified expressions because we use a non-standard sort structure and because we introduce bounded quantifiers as primitives.

Let us note that we did not introduce any restrictions on usage of sorts for the arguments of predicates. In other words, we can use any terms as arguments. However terms have sorts, which leads to the definitions given below.

Now we define the notion of truth for the quantified formulas. Let x be a variable of the sort α, y a variable of the sort Univ, $\mathcal{N}$ a model. Then

1. $\mathcal{N}\models\forall x\varphi(x)$ iff for all elements a of $HU(\alpha)$ we have $\mathcal{N}\models\varphi(a)$;

2. $\mathcal{N}\models\forall x\varphi(x)$ iff for some element a of $HU(\alpha)$ we have $\mathcal{N}\models\varphi(a)$;

3. $\mathcal{N}\models(\forall y\in b)\varphi(y)$ iff for every element a of b we have $\mathcal{N}\models\varphi(a)$;

4. $\mathcal{N}\models(\exists y\in b)\varphi(y)$ iff for some element a of b we have $\mathcal{N}\models\varphi(a)$;

5. $\mathcal{N}\models(\forall y\sqsubseteq b)\varphi(y)$ iff for every $a\sqsubseteq b$ we have $\mathcal{N}\models\varphi(a)$;

6. $\mathcal{N}\models(\forall y\sqsubseteq b)\varphi(y)$ iff for some $a\sqsubseteq b$ we have $\mathcal{N}\models\varphi(a)$.

A clause $A(\bar{x})$:- $\varphi(\bar{x})$ is <u>true</u> on a model $\mathcal{N}$ iff such is the formula $\varphi(\bar{x})\supset A(\bar{x})$.

Let $\mathcal{N}$ be a many-sorted model, $\langle\mathcal{M},\mathbf{D},\mathbf{C}\rangle$ be a generalized logic program. Then $\mathcal{N}$ is a <u>model of</u> $\langle\mathcal{M},\mathbf{D},\mathbf{C}\rangle$ iff

1. The sorts of $\mathcal{N}$ are the sorts of $\mathcal{M}$ and the sorts defined in $L(\mathcal{M},\mathbf{D})$;

2. The interpretation of any predefined sort is the set of all constants of this sort;

3. The predicates of $\mathcal{N}$ are those of $\mathcal{M}$ and those of $\langle P_0,\ldots,P_n,=\rangle$;

4. The interpretation of any defined sort α is $HU(\alpha)$;

5. The interpretation of any predefined predicate symbol in $\mathcal{N}$ is the same as in $\mathcal{M}$;

6. The interpretation of $=$ is equality;

7. All clauses from $\mathbf{C}$ are true on $\mathcal{N}$.

As usual we introduce the relation $\subseteq$ between models of a generalized logic program $\mathcal{P}$ in the following way: $\mathcal{N}_1\subseteq\mathcal{N}_2$ iff for any ground atomic formula φ, $\mathcal{N}_1\models\varphi$ implies $\mathcal{N}_2\models\varphi$.

Lemma 3.1 *Let $\mathcal{N}_1,\mathcal{N}_2$ be models of a generalized logic program $\mathcal{P}$. Then $\mathcal{N}_1\subseteq\mathcal{N}_2$ iff for any ground Σ-formula φ, from $\mathcal{N}_1\models\varphi$ follows $\mathcal{N}_2\models\varphi$.* ∎

We omit proofs of all lemmas and theorems in this paper.

Theorem 3.2 *For every generalized logic program $\mathcal{P}$ there exists a model $\mathcal{N}$ of $\mathcal{P}$ which is minimal w.r.t. $\subseteq$.* ∎

Given a generalized logic program $\mathcal{P}$, let $\mathbf{Mod}(\mathcal{P})$ denotes the set of all ground Σ-formulas that are true in the minimal model of $\mathcal{P}$.

3.2 The least fixedpoint semantics

The definition of the least fixedpoint semantics is similar to the definitions given in [12] and to the formalization of logic programming proposed in [1]: a (generalized) logic program is considered as a monotonic mapping from interpretations to interpretations.

An interpretation $\Im$ is any set of ground Σ-formulas satisfying the following properties:

1. $\varphi \wedge \psi \in \Im$ iff $\varphi \in \Im$ and $\psi \in \Im$;
2. $\varphi \vee \psi \in \Im$ iff $\varphi \in \Im$ or $\psi \in \Im$;
3. $\exists \mathtt{x} \varphi(\mathtt{x}) \in \Im$ iff for some term $\mathtt{t}$ of the same sort as $\mathtt{x}$, $\varphi(\mathrm{t}) \in \Im$;
4. $(\exists \mathtt{x} \in \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ iff for some term $\mathtt{t} \in \mathtt{s}$, $\varphi(\mathtt{t}) \in \Im$;
5. $(\forall \mathtt{x} \in \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ iff for every term $\mathtt{t} \in \mathtt{s}$, $\varphi(\mathtt{t}) \in \Im$;
6. $(\exists \mathtt{x} \sqsubseteq \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ iff for some term $\mathtt{t} \sqsubseteq \mathtt{s}$, $\varphi(\mathtt{t}) \in \Im$;
7. $(\forall \mathtt{x} \sqsubseteq \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ iff for every term $\mathtt{t} \sqsubseteq \mathtt{s}$, $\varphi(\mathtt{t}) \in \Im$;

An interpretation $\Im$ is an <u>interpretation for the model $\mathcal{M}$</u> iff

1. For every predefined predicate symbol P and any tuple of terms $\bar{t}$, $P(\bar{t}) \in \Im$ iff $\mathcal{M} \models P(\bar{t})$.
2. For every terms $\mathtt{s,t}$, $\mathtt{s=t} \in \Im$ iff $\mathtt{s}$ is identical to $\mathtt{t}$.

Note. There is a minimal interpretation for $\mathcal{M}$ which coincides with the set of all Σ-formulas true on $\mathcal{M}$ extended by equality. We shall denote it by $\Im_0$.

Lemma 3.3 *Any interpretation is uniquely characterized by the set of its atomic formulas.* ■

Let $\mathbf{C}^*$ be the set of all ground instances of clauses in $\mathbf{C}$. The <u>immediate consequence operator</u> defined by a program $\mathcal{P}$ is the function $\mathbf{I}_\mathcal{P}$ on the set of all interpretations defined as follows: for an interpretation $\Im$ and a ground atomic formula ψ, $\psi \in \mathbf{I}_\mathcal{P}(\Im)$ iff $\mathbf{C}^*$ contains a clause $\varphi \supset \psi$ such that $\varphi \in \Im$.

Lemma 3.4 *The operator $\mathbf{I}_\mathcal{P}$ is monotonic.* ■

Theorem 3.5 *There is the least fixed point $\mathbf{Lfp}(\mathcal{P})$ of the operator $\mathbf{I}_\mathcal{P}$ among all interpretations containing $\Im_0$. It can be computed as*

$$\mathbf{Lfp}(\mathcal{P}) = \bigcup_{i=0}^{\infty} \Im_i,$$

where $\Im_0$ is as defined, and $\Im_{i+1} = \Im_i \bigcup \mathbf{I}_\mathcal{P}(\Im_i)$. Moreover $\mathbf{Lfp}(\mathcal{P})$ coincides with $\mathbf{Mod}$*($\mathcal{P}$).* ■

3.3 Classical provability

Classical proof systems for lists are obtained from the classical predicate calculus by adding axioms expressing properties of lists. There are two ways to define appropriate extensions of the predicate calculus. The first approach is similar to the approach used in [12] for lists and in [2] for hereditarily finite sets. According to this approach all elements but lists are considered as urelements and some axioms expressing properties of list are added (the calculus $\mathbf{Clt}(\mathcal{P})$ below). The second possibility is to treat sort definitions (including the definition of lists) as inductive definitions. Inductive definitions for sorts define universes for these sorts and also give induction rules for proving properties of elements of the sorts. These induction rules are given in the calculus $\mathbf{CInd}(\mathcal{P})$ below.

The many-sorted predicate calculus we use differs from the ordinary non-sorted predicate calculus in the following restrictions on the axioms for the quantifiers. In the axioms $\forall x\varphi(x)\supset\varphi(t)$ and $\varphi(t)\supset\exists x\varphi(x)$, if the variable x is of the sort α, then t must be a term of the same sort.

We also do not consider in this section bounded quantifiers as primitives, but as notations:

$$\begin{array}{lll} (\forall x\lambda t)\varphi(x) & \text{stands for} & \forall x(x\lambda t \supset \varphi(x)), \\ (\exists x\lambda t)\varphi(x) & \text{stands for} & \exists x(x\lambda t\wedge\varphi(x)), \end{array}$$

where λ denotes $\in$ or $\sqsubseteq$.

3.3.1 The calculus $\mathbf{Clt}(\mathcal{P})$

The calculus $\mathbf{Clt}(\mathcal{P})$ (the classical list theory) is obtained from the classical predicate calculus with equality by adding the following axioms:

1. Axioms for $\mathcal{M}$: all ground atomic formulas true on $\mathcal{M}$;

2. Axioms for $\mathbf{C}$. If a clause $P(\bar{x})$ `:-` $\varphi(x)$ belongs to $\mathbf{C}$, then the formula $\forall\bar{x}(\varphi(\bar{x})\supset P(\bar{x}))$ is an axiom of $\mathbf{Clt}(\mathcal{P})$;

3. Axioms for lists:

$$\begin{array}{l} [\mathtt{s}_1|\mathtt{t}_1]=[\mathtt{s}_2|\mathtt{t}_2]\supset\mathtt{s}_1=\mathtt{t}_1\wedge\mathtt{s}_2=\mathtt{t}_2 \\ \neg\mathtt{s}=[\,] \\ \mathtt{r}\in[\mathtt{s}|\mathtt{t}]\equiv(\mathtt{r}=\mathtt{s}\vee\mathtt{r}\in\mathtt{t}) \\ \mathtt{t}\sqsubseteq[\,]\equiv\mathtt{t}=[\,] \\ \mathtt{t}_1\sqsubseteq[\mathtt{s}|\mathtt{t}_2]\equiv\mathtt{t}_1=[\mathtt{s}|\mathtt{t}_2]\vee\mathtt{t}_1\sqsubseteq\mathtt{t}_2 \end{array}$$

   where $\mathtt{s}_i$,$\mathtt{r}_i$ are terms of the sort `Univ`, $\mathtt{t}_i$ terms of the sort `List`.

4. Induction axioms:

$$\varphi([\,])\wedge\forall\mathtt{x}\forall\mathtt{y}(\varphi(\mathtt{y})\supset\varphi([\mathtt{x}|\mathtt{y}]))\supset\forall\mathtt{y}\varphi(\mathtt{y}),$$

   where φ is any formula, `x` a variable of the sort `Univ`, `y` a variable of the sort `List`.

5. Foundation axiom:

$$\forall \mathtt{y}((\forall \mathtt{x}{\in}\mathtt{y})\varphi(\mathtt{x})\supset\varphi(\mathtt{y}))\supset\forall \mathtt{y}\varphi(\mathtt{y})$$

where φ is any formula, `x` a variable of the sort `Univ`, `y` a variable of the sort `List`.

This theory is almost identical to GES defined in [12].

Theorem 3.6 *A ground Σ-formula φ is provable in* **Clt**$(\mathcal{P})$ *iff $\varphi \in$***Mod***$(\mathcal{P})$.* ■

3.3.2 The calculus CInd($\mathcal{P}$)

The calculus **CInd**$(\mathcal{P})$ treats lists not as a special sort, but simply as an instance of a sort definition. A set of sort definitions gives an induction scheme to prove the properties of the defined sorts, including lists. A somewhat similar approach to the semantics of logic programming had been used in [15]. But in this article inductive definitions are not sorts definitions, but clauses of logic programs.

The calculus **CInd**$(\mathcal{P})$ is obtained from the classical predicate calculus with equality by adding the following axioms and rules. The first three groups of axioms are identical to those of **Clt**$(\mathcal{P})$. The other axioms are

4. Induction rules. Let $\alpha_1, \ldots, \alpha_n$ be all built-in and $\alpha_{n+1}, \ldots, \alpha_k$ all defined sorts from $L(\mathcal{M}, \mathbf{D})$, $\varphi_1(x), \ldots, \varphi_k(x)$ be formulas. Then the following is a rule of **Clt**$(\mathcal{P})$.

$$\frac{\forall x_1\varphi_1(x_1)\ldots\forall x_n\varphi_n(x_n) \quad \Gamma_{n+1}\ldots\ \Gamma_k}{\forall z_i\varphi_i(z_i)}$$

where x_j are variables of the sorts α_j, the variable z_i may be of any sort. The sets Γ_i of formulas are formed as follows (below z_i are variables of the sorts α_i).

If $c \in \alpha_i$ is a definition in $L(\mathcal{M}, \mathbf{D})$, then $\varphi_i(c) \in \Gamma_i$;

If $f \in \alpha_m \times \ldots \times \alpha_p \to \alpha_i$ is a definition in $L(\mathcal{M}, \mathbf{D})$, then $\forall z_m \ldots \forall z_p(\varphi_m(z_m)\wedge\ldots\wedge\varphi_p(z_p)\supset\varphi_i(f(z_m, \ldots, z_p))) \in \Gamma_i$;

If $\alpha_m \subseteq \alpha_i$ is a definition in $L(\mathcal{M}, \mathbf{D})$, then $\forall z_m(\varphi_m(z_m)\supset\varphi_i(z_m)) \in \Gamma_i$;

There are no other formulas in Γ_i.

As we can easily see, if we have the following set of sort definitions, describing natural numbers

$0 \in$`Nat`
$s \in$`Nat`$\to$`Nat`

then the usual induction scheme for natural numbers

$$\frac{\varphi(0) \qquad \forall n(\varphi(n) \supset \varphi(s(n)))}{\forall n\varphi(n)},$$

where n is a variable of the sort `Nat`, can be derived from our rules. However we prefer to consider natural numbers as a built-in sort.

Theorem 3.7 *A ground Σ-formula φ is provable in* **Clt**$(\mathcal{P})$ *iff $\varphi \in$***Mod***$(\mathcal{P})$.* ■

3.4 Constructive provability

The systems $\mathbf{Ilt}(\mathcal{P})$ (intuitionistic list theory) and $\mathbf{IInd}(\mathcal{P})$ have the same axioms and rules as $\mathbf{Clt}(\mathcal{P})$ and $\mathbf{CInd}(\mathcal{P})$ but both are based on the intuitionistic predicate logic instead of the classical one. The intuitionistic variant of list theory was introduced in [23]. In [28] we proved that it is constructive from the viewpoint of a constructive semantics, which in particular means that it has a variant of the existential property: if a ground formula $\exists x\varphi(x)$ is provable then it is possible to effectively find a term t such that $\varphi(t)$ holds.

Theorem 3.8 *For a ground atomic Σ-formula φ, $\mathbf{Ilt}(\mathcal{P})\models\varphi$ iff $\varphi\in\mathbf{Mod}(\mathcal{P})$.* ■

Theorem 3.9 *For a ground atomic Σ-formula φ, $\mathbf{IInd}(\mathcal{P})\models\varphi$ iff $\varphi\in\mathbf{Mod}(\mathcal{P})$.* ■

3.5 The natural calculus

We call this calculus natural because it gives a natural semantics to formulas with bounded quantifiers. The rules of the natural calculus treat these formulas in a very natural and elegant way. The calculus $\mathbf{Nat}(\mathcal{P})$ introduced below is similar to the ground positive hyperresolution on Horn clauses. At the same time the natural calculus serves as a basis for the procedural semantics of generalized logic programs. The natural semantics for Σ-programs [12] was introduced in [26] with the aim of showing that Σ-programs can be efficiently executed using unification instead of the exhaustive search.

The calculus $\mathbf{Nat}(\mathcal{P})$ consists of the following axioms and inference rules:

1. Axioms.
   (a) All ground atomic formulas true on $\mathcal{M}$;
   (b) All axioms of the form `t=t`, where `t` is a ground term.
2. Rules for $\mathbf{C}$:
   $$\frac{\varphi}{A},$$
   if A`:-`φ belongs to $\mathbf{C}^*$.
3. Rules for the logical connectives:
   $$\frac{\varphi}{\varphi\vee\psi}\qquad\frac{\psi}{\varphi\vee\psi}\qquad\frac{\varphi\quad\psi}{\varphi\wedge\psi}$$
4. A rule for the existential quantifier:
   $$\frac{\varphi(t)}{\exists x\varphi(x)}$$
   where x is a variable of a sort α, t a term of the same sort.

5. Rules for bounded quantifiers (here `t` is a term of the sort `List`, `s` an arbitrary term):

$$\frac{\varphi(\mathtt{s})}{(\exists \mathtt{x}\in\mathtt{[s|t]})\,\varphi(\mathtt{x})} \qquad \frac{(\exists \mathtt{x}\in\mathtt{t})\,\varphi(\mathtt{x})}{(\exists \mathtt{x}\in\mathtt{[s|t]})\,\varphi(\mathtt{x})}$$

$$\frac{}{(\forall \mathtt{x}\in\mathtt{[]})\,\varphi(\mathtt{x})} \qquad \frac{\varphi(\mathtt{s}) \quad (\forall \mathtt{x}\sqsubseteq\mathtt{t})\,\varphi(\mathtt{x})}{(\forall \mathtt{x}\in\mathtt{[s|t]})\,\varphi(\mathtt{x})}$$

$$\frac{\varphi(\mathtt{t})}{(\exists \mathtt{x}\sqsubseteq\mathtt{t})\,\varphi(\mathtt{x})} \qquad \frac{(\exists \mathtt{x}\sqsubseteq\mathtt{t})\,\varphi(\mathtt{x})}{(\exists \mathtt{x}\sqsubseteq\mathtt{[s|t]})\,\varphi(\mathtt{x})}$$

$$\frac{\varphi(\mathtt{[]})}{(\forall \mathtt{x}\sqsubseteq\mathtt{[]})\,\varphi(\mathtt{x})} \qquad \frac{\varphi(\mathtt{[s|t]}) \quad (\forall \mathtt{x}\sqsubseteq\mathtt{t})\,\varphi(\mathtt{x})}{(\forall \mathtt{x}\sqsubseteq\mathtt{[s|t]})\,\varphi(\mathtt{x})}$$

Theorem 3.10 *For any ground Σ-formula φ, $\varphi \in$***Mod***$(\mathcal{P})$ iff* **Nat**$(\mathcal{P})\vdash \varphi$. ■

4 Procedural semantics: SLDB-resolution

The natural calculus of section 3.5 represents in a declarative way ideas of the operational semantics. This calculus treats only ground formulas. To produce the operational semantics from the calculus it is sufficient to show how to treat non-ground formulas and how to formalize the top-down search. To this end we introduce a unification algorithm for our terms in section 4.1 and SLDB-resolution in section 4.2.

4.1 Unification

The sorts introduced here require a special unification algorithm. Such an algorithm is possible if we impose some restrictions on the set of all predefined (or basic) sorts. The restrictions are the following:

1. There is an algorithm verifying whether a given constant c belongs to a given basic sort α;
2. There is an algorithm which by given predefined sorts $\alpha, \ldots, \beta$ decides if $C_\alpha \bigcap \ldots \bigcap C_\beta = \emptyset$.

Note. An algorithm for 2 always exists if there is only a finite number of the predefined sorts.

The first restriction is needed to unify a variable of a predefined sort with a constant. The second restriction is necessary to unify variables of different predefined sorts.

For simplicity we assume that the set of all predefined sorts is closed under intersection, i.e. for each two predefined sorts α, β if $C_\alpha \bigcap C_\beta \neq \emptyset$, then there is a predefined sort γ such that $C_\gamma = C_\alpha \bigcap C_\beta$.

The following theorem gives a theoretical basis for the unification algorithm.

Theorem 4.1 *The following statements are true:*

1. *There exists an algorithm verifying whether two sorts have a nonempty intersection.*
2. *If the set of all predefined sorts is closed under intersection, then the set of all sorts is closed under intersection. More precisely, if D is a set of sort definitions, and α, β are sorts with $HU(\alpha) \bigcap HU(\beta) \neq \emptyset$, then there is a set $D' \supseteq D$ of sort definitions which defines a sort γ such that*
   (a) *The Herbrand universe $HU(\xi)$ for any sort ξ defined in D is identical in D and in D';*
   (b) *$HU(\alpha) \bigcap HU(\beta) = HU(\gamma)$ in D'.* ■

According to theorem 4.1 we can assume that the set of all sorts is closed under intersection. Before defining the unification algorithm for our sorts we introduce some definitions.

A <u>substitution</u> is any expression of the form $[t_1/x_1, \ldots, t_n/x_n]$, $n \geq 0$, such that for every i, the variable x_i has the same sort as the term t_i . Such a substitution with $n = 0$ is the <u>empty substitution</u>. The <u>application</u> $t\theta$ of a substitution θ to a term or formula t is the result of the simultaneous replacement of all free occurences of x_i in t by t_i. We denote by $\theta_1 \circ \theta_2$ the substitution θ such that for each term t, $t\theta = (t\theta_2)\theta_1$. A substitution θ is a <u>unifier</u> of terms t_1 and t_2 iff $t_1\theta = t_2\theta$. Two terms are <u>unifiable</u> iff they have a unifier. A <u>covering set of unifiers</u> of t_1 and t_2 is a set $\{\theta_1, \ldots, \theta_n\}$ of unifiers of these terms such that

1. for any other unifier θ of t_1 and t_2 there exist $i \in \{0, \ldots, n\}$ and a substitution θ' with $\theta = \theta' \circ \theta_i$;
2. Any proper subset of $\{\theta_1, \ldots, \theta_n\}$ does not satisfy 1.

A substitution θ is called a <u>most general unifier</u> of t_1 and t_2 iff it belongs to some covering set of unifiers of t_1 and t_2.

Theorem 4.2 *If the set of all sorts is closed under intersection then there is an algorithm verifying whether two given terms t_1 and t_2 are unifiable, and if they are unifiable giving a finite covering set of unifiers of t_1 and t_2.* ■

Now we shall give the unification algorithm. The algorithm consists of repeated transformations of pairs $\langle S, \theta \rangle$, consisting of a set of equations S and a substitution θ. The initial set of equations consists of one equation $t_1 = t_2$, and the initial substitution is the empty substitution ε. The algorithm is as follows:

If the set of equations is empty then the algorithm stops and gives the substitution as the result. Otherwise, let $s_1 = s_2$ be the first equation, S be the rest of the equations and θ be the current substitution. Consider the following cases.

1. s_1 is a constant.
   (a) s_2 is a constant. Then if s_1 and s_2 are identical, we change the current state to S, θ.

(b) s_2 is a variable x of a sort α. Then if s_1 is of the sort α then we change the current state to $\langle S, \theta\rangle[s_1/x]$, otherwise the algorithm fails on this state.

(c) Otherwise the algorithm fails on this state.

2. s_1 is a variable x of a predefined sort α.

   (a) s_2 is a constant. This case is symmetrical to case 1b.

   (b) s_2 is a variable y of a sort β. Then if α and β have a non-empty intersection γ, we change the state to $\langle S, \theta\rangle[z/x, z/y]$, where z is a new variable of the sort γ, otherwise the algorithm fails on this state.

   (c) Otherwise the algorithm fails on this state.

3. s_1 is a variable x of a defined sort α.

   (a) s_2 is a constant. This case is symmetrical to case 1b.

   (b) s_2 is a variable y of a sort β. This case is similar to case 2b.

   (c) s_2 is a term $f(t_1, \ldots, t_n)$. Then if there is a definition $f \in \alpha_1 \times \ldots \times \alpha_n \to \alpha$ then we take n new variables $x_1, \ldots, x_n$ of the sorts $\alpha_1, \ldots, \alpha_n$ and change the state to $\langle S \bigcup \{x_1 = t_1, \ldots, x_n = t_n\}, \theta\rangle[f(x_1, \ldots, x_n)/x]$, otherwise the algorithm fails on this state.

4. s_1 is a term $f(t_1, \ldots, t_n)$.

   (a) s_2 is a variable x of a defined sort α. This case is symmetrical to case 3c.

   (b) s_2 is a term $f(r_1, \ldots, r_n)$. Let α be a sort such that there is a sort definition of the form $f \in \alpha_1 \times \ldots \times \alpha_n \to \alpha$ and let $x_1, \ldots, x_n$ be new variables of the sorts $\alpha_1, \ldots, \alpha_n$. Then we change the state to the states of the form $\langle S \bigcup \{t_1 = r_1, \ldots, t_n = r_n, x_1 = r_1, \ldots, x_n = r_n\}, \theta\rangle$ for all such sorts α.

   (c) Otherwise the algorithm fails on this state.

The only inefficient part of the algorithm is case 4b, where the algorithm can branch and give several unifiers. If we do not allow definitions of the same function symbol as belonging to different sorts, then the most general unifier (if there is one) is unique w.r.t renaming. The domain definitions of PDC-Prolog [17] do not allow multiple declarations of the same function symbol.

4.2 SLDB-resolution

In this section we give a procedural semantics of the language which generalizes SLD-resolution for Horn clause programs. We call it <u>SLDB-resolution</u> (SLD-resolution with Bounded quantifiers). There is no difference between SLDB-resolution and SLD-resolution in the treatment of program clauses, but there are special features in processing built-in predicates and complex formulas.

<u>Goal</u> is any list of Σ-formulas. We assume that the reader is familiar with the notion of SLD-resolution (see e.g. [11]. As usually, the computation rule is a function from the set of all non-empty goals to the set of Σ-formulas such that the value of the function

on a goal is a formula, called <u>selected</u> formula, in that goal. Let $\varphi_1,\ldots,\varphi_n$ be a goal and φ_i is the selected formula. Then the goal $(\varphi_1,\ldots,\varphi_{i-1},\Gamma,\varphi_{i+1},\ldots,\varphi_n)\theta$, where θ is a substitution, Γ is a list of Σ-formulas, is a <u>successor</u> of the goal $\varphi_1,\ldots,\varphi_n$ with the substitution θ iff one of the following conditions holds:

1. φ_i is a built-in atom (an atom of the form $P(\bar{t})$, where P is from the signature of the basic model), Γ the empty list, and θ a substitution which makes φ_i true on $\mathcal{M}$;
2. φ_i takes the form $\psi_1\vee\psi_2$, Γ is ψ_i, $i=1,2$, θ is the empty substitution.
3. φ_i takes the form $\psi_1\wedge\psi_2$, Γ is ψ_1,ψ_2, θ the empty substitution.
4. φ_i takes the form $\exists x\psi(x)$, Γ is $\psi(y)$, θ the empty substitution, y a new variable of the same sort as x.

   In the following y is a new (not occuring in the original goal) variable of the sort `List`, z a new variable of the sort `Univ`.
5. φ_i takes the form $(\exists x{\in}t)\psi(x)$, Γ is $\psi(z)$, θ the most general unifier of t and $[z \mid y]$.
6. φ_i takes the form $(\exists x{\in}t)\psi(x)$, Γ is $(\exists x{\in}y)\psi(x)$, θ is the most general unifier of t and $[z \mid y]$.
7. φ_i takes the form $(\forall x{\in}t)\psi(x)$, Γ is $\psi(z)$,$(\forall x{\in}y)\psi(x)$, θ is the most general unifier of (x,t) and $(z,[z \mid y])$
8. φ_i takes the form $(\forall x{\in}t)\psi(x)$, Γ is empty, θ is the most general unifier of t and `[]`.
9. φ_i takes the form $(\exists x{\sqsubseteq}t)\psi(x)$, Γ is $\psi(y)$, θ is the subsitution $[t/y]$.
10. φ_i takes the form $(\exists x{\sqsubseteq}t)\psi(x)$, Γ is $(\exists x{\sqsubseteq}y)\psi(x)$, θ is the most general unifier of t and $[z \mid y]$.
11. φ_i takes the form $(\forall x{\sqsubseteq}t)\psi(x)$, Γ is $\psi([z \mid y])$,$(\forall x{\sqsubseteq}y)\psi(x)$, θ is the most general unifier of (x,t) and $([z \mid y],[z \mid y])$
12. φ_i takes the form $(\forall x{\sqsubseteq}t)\psi(x)$, Γ is $\psi($`[]`$)$, θ is the most general unifier of t and `[]`.
13. φ_i is an atom $P(\bar{t})$, Γ is $\varphi(\bar{t})$, θ is the empty substitution, where there is a clause of the form $P(\bar{x})$`:-`$\varphi(\bar{x})$ in $\mathbf{C}$.
14. φ_i is an atom $\mathtt{t}_1$`=`$\mathtt{t}_2$, Γ is empty, θ is a most general unifier of $\mathtt{t}_1$ and $\mathtt{t}_2$.

<u>SLDB-derivation</u> is any sequence of pairs $\langle\Gamma_0,\theta_0\rangle,\ldots,\langle\Gamma_n,\theta_n\rangle$ of goals and substitutions such that for every $i\in\{1,\ldots,n\}$ there exists a substitution θ such that

1. Γ_i is a successor of Γ_{i-1} with the substitution θ;
2. $\theta_i=\theta\circ\theta_{i-1}$.

We say that a goal Γ is provable with the substitution θ iff there is an SLDB-derivation starting from $\langle \Gamma, \varepsilon \rangle$ and finishing at $\langle \Lambda, \theta \rangle$, where ε is the empty substitution and Λ the empty goal.

The following theorem states that SLDB-resolution is independent of the computation rule.

Theorem 4.3 *If a goal Γ is provable with a substitution θ under one computation rule, than Γ is provable with θ under any other computation rule.* ■

Let for any formula φ, $\forall\varphi$ denotes the formula $\forall\bar{x}\varphi$, where $\bar{x}$ is the sequence of all free variables of φ. The following theorem states completeness and correctness of SLDB-derivations:

Theorem 4.4 *The following statements are true:*

1. *(Correctness.) If a Σ-formula φ is provable with a substitution θ, then the formula $\forall(\varphi\theta)$ is true in* **Mod***($\mathcal{P}$).*
2. *(Completeness.)If φ is a Σ-formula and ψ is its ground instance true in* **Mod***($\mathcal{P}$), then there are substitutions θ, θ_1, such that φ is provable with the substitution θ and $\varphi\theta\theta_1$ is identical to ψ.* ■

Corollary 4.5 *If φ is a ground Σ-formula, then the following conditions are equivalent.*

1. *φ is provable (with the empty substitution ε);*
2. *φ is true in* **Mod***($\mathcal{P}$).* ■

5 Expressive power

In this section we prove some results about the expressive power of generalized logic programs. In section 5.1 we show a natural translation of generalized logic programs into Horn clause programs. During the translation some new predicates may be defined in the programs. However the obtained Horn clause programs can be recursive, while the initial generalized programs are not recursive. In section 5.2 we construct a non-recursive metainterpreter for Horn clauses in the language of generalized logic programs. It shows the expressive power of generalized logic programs — every computable predicate can be expressed by a non-recursive generalized programs, which means that it can be expressed by a generalized program consisting of one nonrecursive definition. We also prove that this can not be achieved using only bounded quantifiers or only the unbounded existential quantifier.

5.1 Translation to Horn clauses

In this section we will show that the generalized logic programs can be naturally translated to Horn clause programs by adding new predicate symbols. A similar translation can be done for the {log} language of [6], but in this language the only allowed bounded quantifiers are those over elements of a set, which corresponds to our $(\forall x \in t)$. We prove correctness and completeness of such a translation. The existence of the translation is not surprising, because Horn clauses form a universal programming language (a language in which all the computable predicates can be expressed). The interesting features of our translation are that it is quite natural and that non-recursive programs with the bounded quantifiers may be translated into recursive Horn clause programs. In section 5.2 we will show that it can not be avoided in general.

Since our programs are sorted, then we shall assume that the corresponding Horn clause programs are sorted in the same way, and that their semantics is a restriction of our semantics when we omit the bounded quantifiers.

The details of the translation are well known in cases of disjunction, conjunction and the existential quantifier. Suppose that we have a generalized logic program with the set of clauses $\mathbf{C}$. We shall define its translation - a Horn clause program $\widehat{\mathbf{C}}$ in the following way. If there is a non-Horn definition in $\mathbf{C}$, we change it to one or more definitions according to the rules given below in the table, until we get a Horn clause program.

Sentence: / **Its translation:**

```
A(x̄) :- B(x̄)∧C(x̄)            A(x̄) :- D(x̄),E(x̄)
                               D(x̄) :- B(x̄)
                               E(x̄) :- C(x̄)

A(x̄) :- ∃vB(x̄,v)              A(x̄) :- B(x̄,v)

A(x̄) :- (∃y∈t)B(x̄,y)          A(x̄) :- D(x̄,t)
                               D(x̄,[y|z]) :- B(x̄,y)
                               D(x̄,[y|z]) :- D(x̄,z)

A(x̄) :- (∀y∈t)B(x̄,y)          A(x̄) :- D(x̄,t)
                               D(x̄,[])
                               D(x̄,[y|z]) :- B(x̄,y),D(x̄,z)

A(x̄) :- (∃y⊑t)B(x̄,y)          A(x̄) :- D(x̄,t)
                               D(x̄,z) :- B(x̄,z)
                               D(x̄,[y|z]) :- D(x̄,z)

A(x̄) :- (∀y⊑t)B(x̄,y)          A(x̄) :- D(x̄,t)
                               D(x̄,[]) :- B(x̄,[])
                               D(x̄,[y|z]) :- B(x̄,[y|z]),D(x̄,z)
```

Here `D`,`E` are new predicate symbols, `x̄` are all free variables of the clause in the left part of the table. In the right part of the table `y` is a new variable of the sort `Univ`, `z` a new variable of the sort `List`.

Theorem 5.1 *For any generalized logic program* $\mathbf{C}$*,* $\mathbf{C}$ *is equivalent to* $\widehat{\mathbf{C}}$ *in the sense that for any atomic formula* φ*,* φ *is provable by SLDB-resolution from* $\mathbf{C}$ *with a substitution* θ *iff it is provable by SLDB-resolution from* $\widehat{\mathbf{C}}$ *with the same substitution.* ■

Let us note that this equivalence implies the semantic equivalence - the least models of the two programs are identical. It is also easy to see that if to omit sorts and built-in predicates, then SLDB-resolution on Horn clauses is identical to SLD resolution.

5.2 A metainterpreter for Horn clause programs

In the examples from section 2.4 we already had shown that many iterative programs, which are usually expressed in Prolog via recursion, have simple non-recursive definitions using bounded quantifiers. Here we give a more interesting example: a metainterpreter for Horn clause programs. We assume that the clauses of the object level Horn clause program of the form `A:-B`$_1$`,...,B`$_n$ are represented as facts of the form `rule(A,[B`$_1$`,...,B`$_n$`])`, and the facts of the form `A` are represented as `rule(A,[])`. The definition of the metainterpreter is given below:

```
call(Goal) :-
   (∃List)(trace_of_execution(List,[])∧starts(List,[Goal])).

starts([X|Xs],X).
trace_of_execution(List,Last_element) :-
   (∀Sublist⊑List)(Sublist=[]
                    ∨Sublist=[Last_element]
                    ∨step_of_execution(Sublist)).

step_of_execution([State1,State2|States]) :-
   transition(State1,State2).

transition([Atom|Atoms],NewAtoms) :-
   rule(Atom,Tail),
   append(Tail,Atoms,NewAtoms).

append(L1,L2,L3) :-
   (∃List)trace_of_append(L1,L2,L3,List).

trace_of_append(L1,L2,L3,List) :-
   starts(List,[L3,L1]),
   (∀L⊑List))(L=[]∨L=[[[],L2]]∨step_of_append(L)).

step_of_append([[X|Xs],[X|Ys]],[Xs,Ys]|States]).
```

Theorem 5.2 *Let* **C** *be a Horn clause logic program (possibly with sort declarations). Let* **R** *is obtained from* **C** *by replacing each rule* `A:-B`$_1$`,...,B`$_n$ *(each fact* `A` *resp.) with the facts* `rule(A,[B`$_1$`,...,B`$_n$`])` *(facts* `rule(A,[])` *resp.) and by adding the above clauses (with the same sort declarations). Then for any atomic φ, φ is provable by SLDB-resolution from* **C** *with a substitution θ iff the goal* `call(`φ`)` *is provable from* **R** *with the same substitution.*

■

From this theorem we can easily infer that any computable predicate can be expressed by a non-recursive generalized logic program (and hence by a non-recursive generalized logic program consisting of only one definition).

Theorem 5.3 *Let* **C** *be a non-recursive Horn clause program with sort declarations. Then the minimal model computed by* **C** *is decidable (i.e. there is an algorithm verifying if a given ground atomic formula belongs to the model).* ■

Corollary 5.4 *The class of predicates that are computable by the non-recursive generalized logic programs is strictly larger then the class of predicates computable by non-recursive Horn clause programs.* ■

We can also note that the use of bounded quantifiers is essential:

Theorem 5.5 *The class of predicates computable by non-recursive Horn clause programs coincides with the class of programs computable by non-recursive generalized logic programs without bounded quantifiers.* ■

6 Negation

Throughout this section negation means classical negation, unless the inverse is explicitly stated. By classical negation we mean the following: the negation of a formula is true on some elements, if the formula is not true on this elements.

The traditional approaches to handling negation in logic programming are not satisfactory. The main reason is very easy - there can not be a complete and correct implementation of classical negation in logic programs. In the literature concerning negation usually some conditions are given which show when the negation of a predicate defined by Horn clauses satisfies some desirable properties. There are two aspects of using negation: the first concerns computability and the second concerns semantic issues. Let us briefly consider the two aspects.

1. Negation is very hostile to computability. The main reason is the universality of the Horn clause language. Any computable predicate can be represented as a Horn clause program, which means that the negation of predicates defined by Horn clauses is not computable. Even if the negation of a predicate is computable in the Herbrand universe of a program, then the program computing the negation for this Herbrand universe is not correct for extensions of the universe. The usual solution of the computability problem is negation as failure, which is incomplete. It is also difficult to find satisfiable general cases for which negation as failure is complete.

2. As for the semantic aspects, one of the usual solutions is to restrict the class of admissible programs to so called stratified programs or some other classes. These programs have a (stratified) least model, but this model is not computable in general.

Let us informally call predicates with the computable negation <u>negatable</u> predicates. The desirable solution of the two aspects can be summarized as follows: to find a class of programs which is sufficiently rich, but which defines only negatable predicates. We show one such class of generalized logic programs. This class of programs is sufficiently rich, for instance, all the examples of section 2.4 are in this class.

The bounded quantifiers can easily be negated using the following equivalencies:

$$\neg(\forall \mathtt{x}{\in}\mathtt{t})\varphi \equiv (\exists \mathtt{x}{\in}\mathtt{t})\neg\varphi$$
$$\neg(\exists \mathtt{x}{\in}\mathtt{t})\varphi \equiv (\forall \mathtt{x}{\in}\mathtt{t})\neg\varphi$$
$$\neg(\forall \mathtt{x}{\sqsubseteq}\mathtt{t})\varphi \equiv (\exists \mathtt{x}{\sqsubseteq}\mathtt{t})\neg\varphi$$
$$\neg(\exists \mathtt{x}{\sqsubseteq}\mathtt{t})\varphi \equiv (\forall \mathtt{x}{\sqsubseteq}\mathtt{t})\neg\varphi$$

Let us note, that this property is already sufficient to use negation for a wide class of predicates. For instance, all examples of section 2.4 can easily be transformed into non-recursive programs without unrestricted quantifiers. Some of the clauses of the examples contain unrestricted quantifiers, e.g.

```
ordered2(A[X,Y|Z]) :-
   X≤Y.
```

which is a notation for

```
ordered2(L) :-
   ∃X∃Y∃Z(L=[X,Y|Z]∧X≤Y).
```

However this clause is equivalent to the clause

```
ordered2(L) :-
   (∃X∈L)(∃Y∈L)(∃Z⊑L)(L=[X,Y|Z]∧X≤Y).
```

which contains only bounded quantifiers.

It is interesting, that we can validate such a use of patterns in the head of clauses without rewriting them into expressions with bounded quantifiers, using one general theorem proved below. Let us first give precise definitions. A predicate P defined by a generalized logic program $\mathcal{P}$ is called negatable iff there is a generalized logic program $\mathcal{R}$ computing the negation R of P, i.e. the predicate R such that for any tuple t of ground terms from the Herbrand universe of $\mathcal{P}$, $P(t)$ is true in the minimal model for $\mathcal{P}$ iff $R(t)$ is false in the minimal model for $\mathcal{R}$.

Theorem 6.1 *Let* $\mathbf{N}$ *be the class of the generalized logic programs with the following properties:*

1. *No recursion is used in the programs from* $\mathbf{N}$*;*

2. *All occurences of the unrestricted existential quantifier in the programs from* $\mathbf{N}$ *take the form*

   $$\exists \mathtt{y}(\varphi(\bar{\mathtt{x}},\mathtt{y})\wedge\psi(\bar{\mathtt{x}},\mathtt{y})),$$

   where $\bar{\mathtt{x}}$*are all free variables of* φ,ψ *and*

   (a) φ *is negatable;*

   (b) *For any tuple of ground terms* $\mathtt{s}$ *there is at most one term* $\mathtt{t}$ *such that* $\varphi(\mathtt{t},\mathtt{s})$.

Then every defined in $\mathbf{N}$ *predicate is negatable.* ■

Following [12], we shall call the generalized logic programs without unrestricted quantifiers Δ_0-programs.

Corollary 6.2 *If $\mathcal{P}$ is a Δ_0-program without recursion, all built-in predicates are negatable and the equality* `=` *is negatable, then any predicate defined in $\mathcal{P}$, is negatable.* ■

Now we apply theorem 6.2 to extensions of Δ_0-programs:

Theorem 6.3 *Let the equality* `=` *and all built-in predicates be negatable. Let $\mathcal{P}$be a non-recursive generalized logic program with clauses of the form*

$P(\bar{t})$ `:-` φ,

where φ is a Δ_0-formula, and all free variables of φ occur it $\bar{t}$. Then for every predicate P defined in $\mathcal{P}$, P is negatable. ■

Note. If there are no infinite predefined sorts, then the equality is negatable.

Now we shall consider the case of recursive Δ_0-programs. The first result about negation in such programs is negative.

Theorem 6.4 [2] *For every computable set S of ground terms there is a Δ_0-program $\mathcal{P}$ defining a predicate P such that for every ground term t, $P(t)$ belongs to* **Mod***($\mathcal{P}$) iff $t \in S$.*

This result shows that there can be no correct and complete implementations of negation for Δ_0-programs in general. However the (recursive) Δ_0-programs have interesting properties related to negation as failure. There are approaches to solve the problem of negation in logic programs by constructing programs which computes the finite failure state of a given Horn clause program (see e.g.[18]). However the programs generated in such a way can be very complicated even when the original programs are very simple. The programs with bounded quantifiers admit a very elegant solution for constructing such a dual program.

Let every built-in predicate P is negatable and the predicate $\widehat{P}$ represents its negation. Let $=$ be also negatable and $\neq$ is the negation of the equality. Let $\mathbf{C}$ be a set of clauses which do not contain unrestricted quantifiers and which defines predicates $P_1, \dots, P_n$ We introduce new dual predicate symbols $\widehat{P}_1, \dots, \widehat{P}_n$. For each clause $C \in \mathbf{C}$ the dual clause $\widehat{C}$ is constructed as follows: all atomic formulas $R(\bar{t})$ are replaced by $\widehat{R}(\bar{t})$, $t_1 = t_2$ by $t_1 \neq t_2$, $\wedge$ by $\vee$, $\vee$ by $\wedge$, all occurences of $\exists$ by $\forall$, and all occurences of $\forall$ by $\exists$. For example, the dual clause to

```
ordered(L) :-
   (∀X⊑L)(X=[]∨singleton(X)∨ordered2(X)).
```

is

[2]This theorem had been proved by Starchenko and myself

```
not_ordered(L) :-
   (∃X⊑L)(X≠[]∨not_singleton(X)∨not_ordered2(X)).
```

where we denoted the dual predicate symbols with the prefix `not_`. The dual program $\widehat{\mathbf{C}}$ to $\mathbf{C}$ consist of all such dual clauses together with the definitions of predicates that are dual to built-in predicates and to equality.

Theorem 6.5 *For any predicate P defined in the program $\mathbf{C}$ the predicate $\widehat{P}$ from the program $\widehat{\mathbf{C}}$ computes the finite failure set for P.* ■

7 Concluding remarks

There are other aspects of the programming with bounded quantifiers, which are not considered in this paper. In this section we briefly mention some possible research issues on logic programming with bounded quantifiers.

7.1 Other kinds of bounded quantifiers

Intuitively, from the viewpoint of programming bounded quantifiers represent the idea of the iterative search over finite domains. There are domains of different sorts that have not been considered in this paper. For example, the quantification over subsets of a finite set may be needed. If we represent sets by lists, then all our results can be easily generalized for bounded quantifiers over subsets. For example, the step of the translation of the following expression, containing $\forall_\subseteq$-quantifier, where $\subseteq$ is the subset relation

```
A(x̄) :- (∀y⊆t)φ(x̄,y).
```

to Horn clauses gives

```
A(x̄) :- D(x̄,[],t).
D(x,l,[]) :- φ(x̄,l).
D(x̄,l,[y|z]) :- D(x,l,z), D(x,[y|l],z).
```

Bounded quantification over an integer interval `[k..n]` consisting of the numbers from `k` to `n`:

```
A(x̄) :- (∀m∈[k..n])φ(x̄,m).
```

can be translated to

```
A(x̄) :- D(x̄,k,n).
D(x̄,k,n) :- k>n.
D(x̄,k,n) :- k≤n,
            φ(x̄,k),
            k1 is k+1,
            D(x̄,k1,n).
```

The other results of this paper can be formulated for the integer intervals as well, with the difference that integers are considered as a predefined sort.

7.2 Other applications

There are many other applications of bounded quantifiers. It had been noted that they can be used in the constrained logic programming over finite domains [7]. Constraints using the bounded quantifiers can be used in constraint logic programming to keep the set of constraints in a smaller size. To this end it is interesting to develop resolution-like calculus for the formulas with bounded quantifiers to resolve upon constraints similar to the theorem proving technique developed in [4].

The most obvious application of bounded quantifiers is parallel and concurrent logic programming, as was also noted in [14]. The bounded universal quantifier captures AND-parallelism, while the bounded existential quantifier - OR-parallelism. The kind of AND-parallelism inherent to bounded universal quantifiers is similar to FOR-ALL-parallelism from [16]. As for bounded existential quantifiers, their procedural treatment is completely different from the interpretation of unrestricted existential quantifier, which serves only for unification purposes. At the same time the bounded existential quantifiers are easily expressed via the unrestricted quantifier, e.g.

```
(∃x∈y)G :- ∃x(member(x,y)∧G)
```

which has a different procedural interpretation. It shows that the bounded quantifiers can also be used for expressing in a declarative way the control of program execution.

In our opinion bounded quantifiers can also be applied in deductive and relational databases. If we consider databases as finite objects, then bounded quantifiers seems to capture the intuitive semantics of databases better then unrestricted quantifiers. Variants of SLDB-resolution can be also used to formalize different kinds of finite search in databases.

8 Acknowledgements

I am grateful to Francois Bry, who had made many comments on a preliminary version of this paper.

References

[1] K.Apt and M.van Emden. Contributions to the theory of logic programming. *Jornal of the Association for Computing Machinery*, 29(3), 1982.

[2] J.Barwise. *Admissible Sets and Structures.* Springer Verlag, 1975.

[3] C.Beeri, Sh.Naqvi, R.Ramakrishnan, O.Shmueli, and Sh.Tsur. Sets and negation in a logic database language (LDL1). In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 21–36. ACM Press, 1987.

[4] H.-J.Bürkert. A resolution principle for clauses with constraints. In M.E.Stickel, editor, *Proc. 10th CADE*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 178–192, 1990.

[5] C.C.Chang and H.J.Keisler. *Model theory.* North Holland, 1977.

[6] A.Dovier, E.G.Omodeo, E.Pontelli, and G.Rossi. {log}: A logic programming language with finite sets. In *Proc. ICLP'91*, pages 109–124. MIT Press, 1991.

[7] P.van Hentenryck, V.Saraswat, and Y.Deville. Constraint processing in cc(fd). Technical report, Brown University, December 1991.

[8] B.Jayaraman and D.A.Plaisted. Programming with equations, subsets and relations. In *Proc. NACLP'89*, Cleveland, 1989. MIT Press.

[9] G.M.Kuper. Logic programming with sets. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–20. ACM Press, 1987.

[10] G.M.Kuper. On the expressive power of logic programming languages with sets. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 10–14. ACM Press, 1988.

[11] J.W.Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[12] S.S.Goncharov and D.I.Sviridenko. *Σ-programming (in Russian)*, volume 120 of *Vychislitelnye Systemy*, pages 3–29. Novosibirsk, 1985.

[13] S.S.Goncharov and D.I.Sviridenko. Theoretical aspects of σ-programming. In *Mathematical Methods of Specification and Synthesis of Software Systems'85*, volume 215 of *Lecture Notes in Computer Science*, pages 169–179, 1986.

[14] S.S.Goncharov, Yu.L.Ershov, and D.I.Sviridenko. Semantic programming. In *IFIP'86*, pages 1093–1100. Elsevier Science, 1986.

[15] M.Hagiya and T.Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2(1):59–77, 1984.

[16] R.Kowalski. Logic programming. In *Proc.IFIP'83*, pages 133–145. Elsevier Science, 1983.

[17] M.Alexander, P.Bilse, L.Jensen, and e.a. *PDC Prolog User's Guide.* Prolog Development Center, 1990.

[18] T.Sato and H.Tamaki. Transformational logic program synthesis. In *Proc. of the Conference on Fifth Generation Computer Systems*, pages 195–201. ICOT, 1984.

[19] J.T.Schwartz, R.B.K.Devar, E.Dubinski, and E.Schonberg. *Programming with sets: an Introduction to SETL.* Springer Verlag, 1986.

[20] Sh.Tsur and C.Zaniolo. LDL: A logic-based language. In *Proc. 12th International Conference on Very Large Databases*, pages 33–40, Kyoto, Japan, 1986.

[21] D.A.Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[22] A.Voronkov. Program execution methods in σ-programming (in Russian). In *Proc. 4th Soviet Conf. on Applications of Mathematical Logic*, pages 51–53, Tallinn, 1986.

[23] A.Voronkov. Intuitionistic list theory (in Russian). In *Proc. 8th Soviet Conf. on Mathematical Logic*, page 32, Moscow, 1986.

[24] A.Voronkov. Logic programs and their synthesis (in Russian). Technical Report 23, Institute of Mathematics, Novosibirsk, 1986.

[25] A.Voronkov. Synthesis of logic programs (in Russian). Technical Report 24, Institute of Mathematics, Novosibirsk, 1986.

[26] A.Voronkov. *A natural calculus for Σ-programs (in Russian)*, volume 120 of *Vychislitelnye Systemy*, pages 14–23. Novosibirsk, 1987.

[27] A.Voronkov. Logic programming and Σ-programming (in Russian). *Kibernetika*, (1):67–72, 1989.

[28] A.Voronkov. N-realizability: one more constructive semantics. Technical Report 71, Monash University, Department of Computer Science, Clayton, Australia, 1991.