technical report ECRC–92–29

# Logic Programming with Bounded Quantifiers

Andrei Voronkov

EUROPEAN COMPUTER-INDUSTRY RESEARCH CENTRE

ECRC GMBH, ARABELLASTR. 17 D-8000 MÜNCHEN 81, GERMANY - TEL +49 89/926 99 0 - FAX 926 99 170 - TLX 521 6910

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

# 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 the theory of admissible sets [Barwise 75]. Later some variants of bounded quantifiers had been introduced in logic programming languages [Goncharov 85, Schwartz 86, Turner 86, Kuper 87, Dovier 91, Hentenryck 91]. 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. The 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.

# Contents

# List of Figures

# 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 have 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 with recursion leads to non-computable predicates. Similar problems 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 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-theoretic semantics while still being efficiently implementable.

This paper extends [Voronkov 92a]. In [Voronkov 92a] we used a different treatment of sorts which, in particular, allowed us to consider built-in sorts and predicates. The extended sorts require a complicated unification algorithm. Here we do not consider built-in sorts for simplicity.

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

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

This specification implies an obvious way to check if two given sets are disjoint. Similar uses of bounded quantifiers can be found already in the language SETL [Schwartz 86] in which the concept of a set is a first-class concept. SETL is an effort to incorporate sets in a logical way into the procedural language paradigm. It seems very natural to use expressions with bounded quantifiers in logic programs as well. Combining the technique of finite search with the logic programming technique (a variant of SLD-resolution) 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 bounded quantifiers, originally introduced in [Goncharov 85] following [Moor 81], but our technique is quite general and is also applicable to other kinds of bounded quantifiers.

In this Section 2 we give the basic definitions of sorts, lists and generalized logic programs. Some natural examples of specifications using bounded quantifiers are given in Section 2.3. In Section 3 we introduce several semantics of logic programs with bounded quantifiers and prove their equivalence. Section 4 describes the procedural semantics of the language, which generalizes SLD-resolution. In Section 5 we discuss the expressive power of subsets of the language with bounded quantifiers. For the class of all generalized logic programs it is equivalent to the expressive power of the Horn clauses — both kinds of languages can express exactly all computable predicates in the least Herbrand 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] [Goncharov 85, Goncharov 86a, Goncharov 86b, Kuper 87, Kuper 88, Dovier 91, Hentenryck 91, Barklund 92]. Also related are papers on introducing set constructs in logic programming [Beeri 87, Kuper 87, Kuper 88, Jayaraman 89]. Somewhat similar in spirit are *safe formulas* [Nicolas 83, Topor 87]. For us the main motivation was the series of papers on $\Sigma$-programming [Goncharov 85, Goncharov 86a, Goncharov 86b] and some of our results on semantics of $\Sigma$-programs and on a translation of $\Sigma$-programs into logic programs [Voronkov 86a, Voronkov 87, Voronkov 89]. Bounded quantifiers were already introduced in the first of the above-mentioned papers [Goncharov 85] which was inspired by the Kripke-Platek formalization of the theory of admissible sets [Barwise 75]. However there were no satisfiable procedural semantics in [Goncharov 85]. Later ideas for defining such a semantics were introduced in our paper [Voronkov 87] based on the translation of $\Sigma$-programs to Horn clause programs described in [Voronkov 86a, Voronkov 89] (some ideas could even be found in even earlier preprints [Voronkov 86c, Voronkov 86d], written in 1985).

Later bounded quantifier were introduced in [Kuper 87, Kuper 88] 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. The set constructor was

[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).

introduced later [Dovier 91] where a more satisfiable procedural semantics for a logic programming language with finite sets was defined. However sets are not easy to handle: the unification problem for finite sets is NP-complete [Dovier 91]. 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.

Bounded quantifiers were also considered in constraint logic programming [Hentenryck 91]. In most of the programming literature they are called restricted quantifiers, but bounded quantifiers introduced earlier in the mathematical literature (e.g. [Barwise 75]) seem to better capture the idea of the search on finite domains.

Bounded quantifiers are usually quantifiers over finite domains represented by lists or sets. In the papers [Goncharov 85, Goncharov 86a, Goncharov 86b, Kuper 87, Kuper 88] lists and sets are considered as a superstructure of the usual Herbrand universe, which prohibits using terms like $f$(`[]`), since `[]` is a list. Our two-sorted models allow lists to be treated as an ordinary sort. In [Dovier 91] function symbols are also allowed to have set arguments. In that paper sorts are not explicitely introduced but they are used in the unification algorithm, and without sorts the procedural semantics from [Dovier 91] becomes incorrect.

The proof theory varies from approach to approach. In [Goncharov 85] an analog of Kripke-Platek theory for admissible sets [Barwise 75] called GES is used for proving properties of $\Sigma$-programs. In [Kuper 87] the underlying proof system is the calculus with the extensionality axiom $\forall$`z(z`$\in$`x`$\equiv$`z`$\in$`y)`$\supset$`x=y`. In [Voronkov 92a] we used 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. For the two-sorted structures of this paper a theory like GES or the list theory of [Moor 81] is more appropriate.

Our approach to defining a procedural semantics for our language, which we call SLDB-resolution, is more efficient compared to the cited papers. In [Dovier 91] an exponential unification algorithm for sets is used. In [Kuper 87] 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`$\leftarrow\{\}$`]` during unification of `q` from the query with `q` from the head of the clause. In [Goncharov 85] the proposed procedural semantic comprises an exhaustive search over infinite universes. According to that paper, 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.

However both [Kuper 87] and [Dovier 91] have other motivations and for some applications finite sets could be more appropriate than lists used in our paper.

In [Barklund 92] bounded quantifiers are defined via formulas of the form $\forall x\{\Theta[x] \rightarrow \Phi[x]\}$, where $\Theta$ is a formula, "which is "obviously" true for only a finite number of values of $x$". Such quantifiers may easily be translated to our quantifiers, if we add the `findall` predicate to the language. (As far as we understand, "obviousness" should allow the `findall` construction.) to be used. An essential difference of our use of bounded quantifiers is that it also allows sets with the given properties to be constructed, while the approach from [Barklund 92] can only be used for given sets. The paper [Barklund 92] is interesting, because it contains an extensive treatment of bounded quantifiers from the viewpoint of (concurrent) implementations.

The advantages of lists over sets are illustrated by examples given in this paper and by results about expressiveness from Section 5, which essentially use bounded quantification over tails of lists, which has no analog for sets. It can also be noted that lists form a structure extensively used in programming languages.

Also similar to bounded quantifiers are some of Zermelo-Frenkel set theory expressions of the functional programming language Miranda [Turner 86].

# 2 Logic programs with bounded quantifiers

In this section we introduce main notions of the paper. In our presentation of generalized logic programs sorts are assigned to terms. In Section 2.1 we introduce sorts and lists. Lists are used in defining bounded quantifiers and generalized logic programs in Section 2.2. We give some examples in Section 2.3 to show the practical importance of bounded quantifiers.

## 2.1 Sorts and lists

Consider a simple expression $(\forall \mathrm{x} \in \mathrm{l})\varphi(x)$ 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 in general be inefficient and it would obscure the semantics of our language. So we need to distinguish lists from all other elements. To this end we introduce a two-sorted language. In [Voronkov 92a] we used complicated sort structures with the aim of handling lists and built-in predicates. The sort definitions that we used in that paper are similar to sorts of PDC-prolog [PDC 90]. However we prefer to use two-sorted language which is sufficient for our purposes.

The use of non-sorted structures for our purposes is inefficient and leads to some semantic problems. For example, in [Goncharov 85, Goncharov 86b] 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 [Goncharov 85, Goncharov 86a] is based on model theory and therefore comprises an exhaustive search over an infinite universe. The two-sorted language of our paper 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 [Voronkov 86a, Voronkov 89], but in those papers we used an algorithm verifying if the terms from the binding expressions are lists.

The first order language of this paper contains two sorts: the universal sort $\mathfrak{univ}$ and the sort of lists $\mathfrak{list}$. We assume that we have countable sets of variables $V_{\mathfrak{univ}}$ and $V_{\mathfrak{list}}$. Let $\mathfrak{L}$ be a language consisting of two sets $C$ of constants and $F$ of function symbols with arities. We assume that $C$ contains the constant `nil` and $F$ contains a binary function symbol `cons`. Below we define the sets of terms for both sorts.

**Definition 2.1 (The sets $\mathrm{Term}_{\mathfrak{univ}}$ and $\mathrm{Term}_{\mathfrak{list}}$ of terms of the language $\mathfrak{L}$)**

1. *If* $t \in \mathrm{Term}_{\mathfrak{list}}$ *then* $t \in \mathrm{Term}_{\mathfrak{univ}}$.
2. `nil` $\in \mathrm{Term}_{\mathfrak{list}}$.
3. $V_{\mathfrak{univ}} \subseteq \mathrm{Term}_{\mathfrak{univ}}$ *and* $V_{\mathfrak{list}} \subseteq \mathrm{Term}_{\mathfrak{list}}$
4. $C \subseteq \mathrm{Term}_{\mathfrak{univ}}$.
5. *If* $f \in F$ *is an* $n$*-ary function symbol and* $t_1, \ldots, t_n \in \mathrm{Term}_{\mathfrak{univ}}$, *then* $f(t_1, \ldots, t_n) \in \mathrm{Term}_{\mathfrak{univ}}$.
6. *If* $s \in \mathrm{Term}_{\mathfrak{univ}}$ *and* $t \in \mathrm{Term}_{\mathfrak{list}}$ *then* `cons(s,t)` $\in \mathrm{Term}_{\mathfrak{list}}$.

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

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

**Definition 2.2 (Ground terms and formulas)** *A term is <u>ground</u> iff no variable occurs in it. A formula is ground iff all variables in it are bound.*

In mathematical logic ground formulas are usually called *closed.* We call them *ground formulas* for the sake of uniformity.

**Definition 2.3 (Herbrand universe $HU$)** *The <u>Herbrand universe</u> $HU$ is the set of all ground terms from* $\mathrm{Term}_{\mathsf{univ}}$.

**Definition 2.4 (Lists)** *<u>Lists</u> are ground terms of* $\mathrm{Term}_{\mathsf{list}}$.

We do not define here lists as a superstructure of the ordinary terms as it was 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.

We define two relations $\in$, $\sqsubseteq$ on lists as follows:

**Definition 2.5 (Relations $\in$ and $\sqsubseteq$)**

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

## 2.2 Bounded quantifiers and generalized logic programs

Here we introduce bounded quantifiers. Apart from sorts they are basically equivalent to bounded quantifiers introduced in [Goncharov 85].

**Definition 2.6 (Bounded quantifiers)** *<u>Bounded quantifiers</u> 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` *does not occur in the term* `t` *of the sort* $\mathsf{list}$.

**Definition 2.7 ($\Sigma$-formulas and $\Delta_0$-formulas)** *A <u>$\Sigma$-formula</u> 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 <u>unrestricted</u> existential quantifier. A <u>$\Delta_0$-formula</u> is a $\Sigma$-formula containing no occurrences of the unrestricted quantifier $\exists$.*

**Definition 2.8 (Generalized logic programs)** *A generalized logic program $\mathcal{P}$ is a set of <u>clauses</u> of the form*

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

*where $i = 0,\ldots,n$, $P_i$ are predicate symbols, $\varphi_i$ are $\Sigma$-formulas, whose all free variables are in $\bar{x}_i$ and whose predicate symbols are in the set $P_0,\ldots,P_n$,*`=`.

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

$$P(\bar{t}_1) \ \text{:-}\ \varphi_1$$
$$\vdots$$
$$P(\bar{t}_n) \ \text{:-}\ \varphi_n$$

will denote the clause

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

where $\bar{x}$ are new variables, $\bar{y}_i$ are all the variables of $t_i$. This notation is similar to the translation used in [Clark 78].

**Definition 2.9 (Queries)** *Query to a generalized logic program $\mathcal{P}$ is any $\Sigma$-formula.*

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 [Goncharov 85] following [Moor 81]. 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 this bounded quantifier for sets, because there is no analog of the relation $\sqsubseteq$.

## 2.3 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. Before giving the examples we will make the following agreement about syntax.

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

We also need notation to distinguish sorts of variables in programs. By default we assume the following:

*A variable* `v` *is of sort* $\mathfrak{list}$ *iff it occurs in a bounded quantifier in one of the following expressions:*

1. $(Qx\in\mathtt{v})$;
2. $(Qx\in[\mathtt{t}_1,\ldots,\mathtt{t}_n|\mathtt{v}])$;
3. $(Qx\sqsubseteq\mathtt{v})$;
4. $(Qx\sqsubseteq[\mathtt{t}_1,\ldots,\mathtt{t}_n|\mathtt{v}])$;
5. $(Q\mathtt{v}\sqsubseteq\mathtt{t})$.

*where $Q$ is $\forall$ or $\exists$*

**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 defined separately.

**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)(X=[]∨norep1(X)).

norep1([A|As]) :-
   (∀X∈As)A≠X.
```

**Example 2.5** *A program finding a route in a graph. We assume that two 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-theoretic 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 feature specific to generalized logic programs are bounded quantifiers. The semantics of Horn clause programs must be modified so as to handle them.

In Section 3.1 we introduce a model-theoretic 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-theoretic 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 [Goncharov 85], and another one which considers lists as elements generated by inductive definitions.

Section 3.5 presents the so called natural semantics for generalized logic programs which was originally introduced in [Voronkov 87]. 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-theoretic semantics

The main semantics of generalized logic programs is the model theoretic semantics which allows for a declarative reading of programs. Our semantics is similar to the semantics introduced in the papers [Goncharov 85, Kuper 87, Dovier 91]. The main difference between our semantics and that of [Goncharov 85] is that we allow function symbols to be constructors, which means that we can define new terms, whereas in [Goncharov 85] the model is fixed — the set of all lists with atoms from the basic model.

To treat lists we have to restrict the class of models. To this end we introduce so called *admissible models*:

**Definition 3.1 (Admissible models)** *Let $\mathfrak{M}$ be a two-sorted model with sorts $\mathfrak{univ}$ and $\mathfrak{list}$. Let atoms be either `[]` or elements of $\mathfrak{M}$ which are not of the sort $\mathfrak{list}$. The model $\mathfrak{M}$ is admissible iff the following statements are true:*

1. *The interpretation of the relation `=` is equality.*
2. *The interpretation of the sort $\mathfrak{list}$ is a subset of the interpretation of the sort $\mathfrak{univ}$.*
3. *All elements of $\mathfrak{M}$ of the sort $\mathfrak{list}$ either are `[]` or can be built from atoms by a finite number of applications of `cons`.*
4. $\mathfrak{M} \quad [x\,|\,y]=[u\,|\,v]\supset x=u\land y=v$.
5. $\mathfrak{M} \quad \neg[x\,|\,y]=[\,]$.

In other words, admissible models are models the sort $\mathfrak{list}$ is generated by free constructors `nil` and `cons`.

Now to introduce the model-theoretic semantics of our programs we have to define the meaning of quantified expressions. First let us note that it is possible to introduce the relations $\in$ and $\sqsubseteq$ on admissible models in the same way as in Definition 2.5.

**Definition 3.2 (Truth)** *Let $\mathfrak{M}$ be an admissible model. The notion of truth for formulas with bounded quantifiers is defined similar to the standard definition [Chang 77] with the following items for quantifiers:*

1. $\mathfrak{M} \quad \forall x\varphi(x)$ *iff for every element $a$ of $\mathfrak{M}$ of the same sort as $x$ we have* $\mathfrak{M} \quad \varphi(a)$.
2. $\mathfrak{M} \quad \exists x\varphi(x)$ *iff for some element $a$ of $\mathfrak{M}$ of the same sort as $x$ we have* $\mathfrak{M} \quad \varphi(a)$.
3. $\mathfrak{M} \quad (\forall x{\in}t)\varphi(x)$ *iff* $\mathfrak{M} \quad \forall x(x{\in}t\supset\varphi(x))$.
4. $\mathfrak{M} \quad (\exists x{\in}t)\varphi(x)$ *iff* $\mathfrak{M} \quad \exists x(x{\in}t\wedge\varphi(x))$.
5. $\mathfrak{M} \quad (\forall x{\sqsubseteq}t)\varphi(x)$ *iff* $\mathfrak{M} \quad \forall x(x{\sqsubseteq}t\supset\varphi(x))$.
6. $\mathfrak{M} \quad (\exists x{\sqsubseteq}t)\varphi(x)$ *iff* $\mathfrak{M} \quad \exists x(x{\sqsubseteq}t\wedge\varphi(x))$.

**Definition 3.3** *A clause $A(\bar{x})\text{:-}\varphi(\bar{x})$ is true on a model $\mathfrak{N}$ iff the formula $\forall\bar{x}(\varphi(\bar{x})\supset A(\bar{x}))$ is true on $\mathfrak{M}$.*

**Definition 3.4 (Model of a program)** *A model $\mathfrak{M}$ is a model of the program $\mathcal{P}$ iff all clauses from $\mathcal{P}$ are true on $\mathfrak{M}$.*

**Definition 3.5 (Relation $\subseteq$)** *Relation $\subseteq$ between models of a program $\mathcal{P}$ is defined in the following way: $\mathfrak{N}_1 \subseteq \mathfrak{N}_2$ iff for any ground $\Sigma$-formula $\varphi$, $\mathfrak{N}_1 \models \varphi$ implies $\mathfrak{N}_2 \models \varphi$.*

The main notion for this section is *Herbrand models.*

**Definition 3.6 (Herbrand models)** *An admissible model $\mathfrak{M}$ is a Herbrand model iff the interpretation of the sort* $\mathfrak{univ}$ *on $\mathfrak{M}$ is the set $HU$ of all ground terms and the relation* = *is interpreted as identity on the set of terms.*

**Lemma 3.1** *Let $\mathfrak{N}_1, \mathfrak{N}_2$ be Herbrand models of a program $\mathcal{P}$. Then $\mathfrak{N}_1 \subseteq \mathfrak{N}_2$ iff for any ground atomic formula $\varphi$, from $\mathfrak{N}_1 \models \varphi$ follows $\mathfrak{N}_2 \models \varphi$.*

*Proof.* In one direction obvious, because each atomic formula is a $\Sigma$-formula. In the other direction follows from the fact that on Herbrand models the truth of a ground quantified expression $Qx\varphi(x)$, where $Q$ is either $\forall$ or $\exists$, is fully determined by the truth of ground formulas of the form $\varphi(t)$.

■

Generalized logic programs do not have the model intersection property for $\Sigma$-formulas. Indeed, if is straightforward to construct two Herbrand models, such that the formula $A{\vee}B$ is true on both, but neither $A$ nor $B$ are.

However we can prove that the model intersection property holds for atomic formulas, using the following two lemmas.

**Definition 3.7 (Intersection of models)** *Let $\{\mathfrak{M}_i \mid i \in I\}$ be Herbrand models. The* <u>*intersection*</u> $\bigcap_{i\,\in\,I} \mathfrak{M}_i$ *of* $\mathfrak{M}_i$ *is the model* $\mathfrak{M}$ *such that for any* atomic *ground formula* $\varphi$*, we have* $\mathfrak{M} \quad \varphi$ *iff* $\mathfrak{M}_i \quad \varphi$*, for all* $i \in I$*.*

**Lemma 3.2** *Let $\varphi$ be a ground $\Sigma$-formula which is false on $\mathfrak{M}_i$, for some $i \in I$. Then* $\bigcap_{i\,\in\,I} \mathfrak{M}_i \quad \varphi$.

*Proof.* Denote $\bigcap_{i\,\in\,I} \mathfrak{M}_i$ by $\mathfrak{M}$. We will use induction on the depth of $\varphi$.

1. For atomic $\varphi$, straightforward from definitions.
2. Let $\varphi$ take the form $\psi \vee \chi$ and $\mathfrak{M}_i \quad \psi \vee \chi$. Then $\mathfrak{M}_i \quad \psi$ and $\mathfrak{M}_i \quad \chi$. The induction hypothesis gives $\mathfrak{M} \quad \psi$ and $\mathfrak{M} \quad \chi$ and hence $\mathfrak{M} \quad \varphi \vee \chi$.
3. Let $\varphi$ take the form $\psi \wedge \chi$ and $\mathfrak{M}_i \quad \psi \wedge \chi$. Then $\mathfrak{M}_i \quad \psi$ or $\mathfrak{M}_i \quad \chi$. The induction hypothesis gives $\mathfrak{M} \quad \psi$ or $\mathfrak{M} \quad \chi$ and hence $\mathfrak{M} \quad \varphi \wedge \chi$.
4. Let $\varphi$ take the form $\exists x \psi(x)$ and $\mathfrak{M}_i \quad \exists x \psi(x)$. Then, $\mathfrak{M}_i \quad \psi(t)$, for all ground terms $t$. The induction hypothesis gives $\mathfrak{M} \quad \psi(t)$, for all ground terms $t$. Since $\mathfrak{M}$ is a Herbrand model, then $\mathfrak{M} \quad \exists x \psi(x)$.
5. Let $\varphi$ take the form $(\exists x \in [t_1, \ldots, t_n])\psi(x)$. Then $\varphi$ is equivalent to disjunction $\psi(t_1) \vee \ldots \vee \psi(t_n)$ and the proof is equivalent to case 2.

The other cases are similar.

■

**Lemma 3.3** *Let $\{\mathfrak{M}_i \mid i \in I\}$ be Herbrand models of a generalized logic program $\mathcal{P}$. Then* $\bigcap_{i\,\in\,I} \mathfrak{M}_i$ *is also a model of* $\mathcal{P}$*.*

*Proof.* Denote $\bigcap_{i\,\in\,I} \mathfrak{M}_i$ by $\mathfrak{M}$. Let $P(\bar{x})\texttt{:-}\varphi(\bar{x})$ be a clause from $\mathcal{P}$. We have to show $\mathfrak{M} \quad \varphi(\bar{t}) \supset P(\bar{t})$, for every tuple $\bar{t}$ of ground terms. Assume that $\mathfrak{M} \quad \varphi(\bar{t}) \supset P(\bar{t})$. Then $\mathfrak{M} \quad \varphi(\bar{t})$ and $\mathfrak{M} \quad P(\bar{t})$. By Lemma 3.2 we have that $\mathfrak{M}_i \quad \varphi(\bar{t})$, for all $i \in I$. From $\mathfrak{M} \quad P(\bar{t})$ it follows that for some $j \in I$, $\mathfrak{M}_j \quad P(\bar{t})$. Hence $\mathfrak{M}_j$ is not a model for $\mathcal{P}$.

■

**Theorem 1** *For every generalized logic program $\mathcal{P}$ there exists a Herbrand model $\mathfrak{M}$ of $\mathcal{P}$ which is minimal among Herbrand models w.r.t. $\subseteq$.*

*Proof.* By Lemma 3.3 the intersection of all Herbrand models of $\mathcal{P}$ is a model of $\mathcal{P}$. Obviously this model is minimal.

■

**Definition 3.8 (Mod($\mathcal{P}$))** *Given a generalized logic program $\mathcal{P}$, let* **Mod**$(\mathcal{P})$ *denote the set of all ground $\Sigma$-formulas which are true in the minimal Herbrand model of $\mathcal{P}$.*

We shall use the following definition extensively throughout the paper:

**Definition 3.9** *Let $\mathcal{P}$ be a program. Then predicate $P$* <u>*computes*</u> *or* <u>*defines*</u> *the set $S$ in $\mathcal{P}$ iff* $S = \{t \mid P(t) \in \mathbf{Mod}(\mathcal{P})\}$.

### 3.2 Least fixedpoint semantics

The definition of the least fixedpoint semantics is similar to the definitions given in [Goncharov 85] and to the formalization of logic programming proposed in [Apt 82]: a (generalized) logic program is considered as a monotonic mapping from interpretations to interpretations.

**Definition 3.10 (Interpretations)** *An interpretation* $\Im$ *is any set of ground* $\Sigma$*-formulas satisfying the following properties:*

1. *For all terms* $s, t$, $s{=}t \in \Im$ *iff* $s$ *is identical to* $t$.
2. $\varphi \wedge \psi \in \Im$ *iff* $\varphi \in \Im$ *and* $\psi \in \Im$;
3. $\varphi \vee \psi \in \Im$ *iff* $\varphi \in \Im$ *or* $\psi \in \Im$;
4. $\exists \mathtt{x} \varphi(\mathtt{x}) \in \Im$ *iff for some term* $\mathtt{t}$ *of the same sort as* $\mathtt{x}$, $\varphi(\mathtt{t}) \in \Im$;
5. $(\exists \mathtt{x} \in \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ *iff for some term* $\mathtt{t} \in \mathtt{s}$ *of the same sort as* $\mathtt{x}$, $\varphi(\mathtt{t}) \in \Im$;
6. $(\forall \mathtt{x} \in \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ *iff for every term* $\mathtt{t} \in \mathtt{s}$ *of the same sort as* $\mathtt{x}$, $\varphi(\mathtt{t}) \in \Im$;
7. $(\exists \mathtt{x} \sqsubseteq \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ *iff for some term* $\mathtt{t} \sqsubseteq \mathtt{s}$ *we have* $\varphi(\mathtt{t}) \in \Im$;
8. $(\forall \mathtt{x} \sqsubseteq \mathtt{s}) \varphi(\mathtt{x}) \in \Im$ *iff for every term* $\mathtt{t} \sqsubseteq \mathtt{s}$ *we have* $\varphi(\mathtt{t}) \in \Im$;

**Definition 3.11** ($\mathrm{Atom}(\Im)$) *For an interpretation* $\Im$, *we denote by* $\mathrm{Atom}(\Im)$ *the set of all atomic formulas from* $\Im$.

**Lemma 3.4** *Let* $\Im$, $\mathfrak{J}$ *be two interpretations. Then* $\Im \subseteq \mathfrak{J}$ *iff* $\mathrm{Atom}(\Im) \subseteq \mathrm{Atom}(\mathfrak{J})$.

*Proof.* Straightforward.

■

**Lemma 3.5** *Any interpretation is uniquely characterized by the set of its atomic formulas.*

*Proof.* Directly from Lemma 3.4.

■

**Definition 3.12** ($\mathcal{P}^*$) *Let* $\mathcal{P}$ *be a program. We use* $\mathcal{P}^*$ *to denote the set of all ground instances of clauses from* $\mathcal{P}$.

**Definition 3.13 (The immediate consequence operator $\mathbf{I}_{\mathcal{P}}$)** *The immediate consequence operator 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* $\psi$, $\psi \in \mathbf{I}_{\mathcal{P}}(\Im)$ *iff* $\mathcal{P}^*$ *contains a clause* $\varphi \supset \psi$ *such that* $\varphi \in \Im$.

**Lemma 3.6** *The operator $\mathbf{I}_{\mathcal{P}}$ is monotonic, i.e. for any two interpretations $\mathfrak{I}$,$\mathfrak{J}$, from $\mathfrak{I} \subseteq \mathfrak{J}$ it follows that $\mathbf{I}_{\mathcal{P}}(\mathfrak{I}) \subseteq \mathbf{I}_{\mathcal{P}}(\mathfrak{J})$.*

*Proof.* Straightforward from Lemma 3.4.

∎

**Definition 3.14 (The minimal interpretation $\mathfrak{I}_0$)** *The minimal interpretation $\mathfrak{I}_0$ is defined by*

$$\mathrm{Atom}(\mathfrak{I}_0) = \{t{=}t \mid t \text{ is a ground term}\}$$

From Lemma 3.5 follows that $\mathfrak{I}_0$ is unique.

**Theorem 2** *There is the least fixed point $\mathbf{Lfp}(\mathcal{P})$ of the operator $\mathbf{I}_{\mathcal{P}}$ among all interpretations containing $\mathfrak{I}_0$. It can be computed as*

$$\mathrm{Atom}(\mathbf{Lfp}(\mathcal{P})) = \bigcup_{i=0}^{\infty} \mathrm{Atom}(\mathfrak{I}_i),$$

*where $\mathfrak{I}_0$ is as defined, and $\mathrm{Atom}(\mathfrak{I}_{i+1}) = \mathrm{Atom}(\mathfrak{I}_i) \bigcup \mathrm{Atom}(\mathbf{I}_{\mathcal{P}}(\mathfrak{I}_i))$. Moreover $\mathbf{Lfp}(\mathcal{P})$ coincides with $\mathbf{Mod}(\mathcal{P})$.*

*Proof.*

1. $\mathbf{Lfp}(\mathcal{P})$ is a fixedpoint.
   (a) Obviously, $\mathbf{I}_{\mathcal{P}}(\mathfrak{I}_i) \subseteq \mathfrak{I}_{i+1}$, and thus $\mathbf{I}_{\mathcal{P}}(\mathbf{Lfp}(\mathcal{P})) \subseteq \mathbf{Lfp}(\mathcal{P})$.
   (b) By the definition of $\mathfrak{I}_j$ and by Lemma 3.6, we have $\mathfrak{I}_i \subseteq \mathfrak{I}_{i+1}$, and thus $\mathbf{Lfp}(\mathcal{P}) \subseteq \mathbf{I}_{\mathcal{P}}(\mathbf{Lfp}(\mathcal{P}))$.
2. $\mathbf{Lfp}(\mathcal{P})$ is a least fixedpoint. Let $\mathfrak{J}$ be any fixedpoint of $\mathbf{I}_{\mathcal{P}}$, containing $\mathfrak{I}_0$. Then, by repeated applications of the monotonicity of $\mathbf{I}_{\mathcal{P}}$(Lemma 3.6), we have that $\mathfrak{I}_i \subseteq \mathfrak{J}$, and hence $\mathbf{Lfp}(\mathcal{P}) \subseteq \mathfrak{J}$.
3. $\mathbf{Lfp}(\mathcal{P}) = \mathbf{Mod}(\mathcal{P})$. By induction on $i$ one can prove that all atoms from $\mathfrak{I}_i$ belong to any model of $\mathcal{P}$. It is straightforward to prove that any fixedpoint is a model.

∎

## 3.3 Classical provability

Classical proof systems for lists are obtained from the classical predicate calculus by adding axioms expressing properties of lists. In [Voronkov 92a] we used two ways to define appropriate extensions of the predicate calculus. The first approach is similar to the approach used in [Barwise 75] for hereditarily finite sets, in [Goncharov 85] for lists and in [Kuper 88] for finite sets. According to this approach all elements except lists are considered as urelements and some axioms expressing properties of list are added to the predicate calculus. The second approach from [Voronkov 92a] is to treat sort definitions (including the definition of lists) as inductive

Figure 3.1: List theories $\mathbf{Clt}(\mathcal{P})$ and $\mathbf{Ilt}(\mathcal{P})$

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

2. Axioms for lists:

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

   where $\mathtt{s}_i$,$\mathtt{r}_i$ are arbitrary terms of the sort $\mathfrak{univ}$, $\mathtt{t}_i$ arbitrary terms of the sort $\mathfrak{list}$.

3. 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 $\varphi$ is any formula, $\mathtt{x}$ a variable of the sort $\mathfrak{univ}$, $\mathtt{y}$ a variable of the sort $\mathfrak{list}$.

definitions. Inductive definitions for sorts define universes for these sorts and also give induction rules for proving properties of elements of the sorts. Here we have only two sorts, so the first approach is more appropriate.

The order-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 $\alpha$, 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:

| | | |
|---|---|---|
| $(\forall x\lambda t)\varphi(x)$ | stands for | $\forall x(x\lambda t\supset\varphi(x))$, |
| $(\exists x\lambda t)\varphi(x)$ | stands for | $\exists x(x\lambda t\wedge\varphi(x))$, |

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

**Definition 3.15 (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 axioms given in Figure 3.1 on page 15.*

This theory is almost identical to GES defined in [Goncharov 85]. We have omitted the foundation axiom from [Goncharov 85]. The use of the foundation axiom requires changing the system of sorts. One possibility is to use one-sorted logic as in [Barwise 75] which is not convenient for our purposes. Another possibility a use of the foundation axiom is the introduction of a more complicated sort structure as in [Voronkov 92a].

Theory $\mathbf{Clt}(\mathcal{P})$ can be used as the proof theory for logic programming with bounded quantifiers. Using this theory one can prove properties of lists and programs. The following theorem shows completeness and correctness of $\mathbf{Clt}(\mathcal{P})$ w.r.t. other semantics.

**Theorem 3** *A ground $\Sigma$-formula $\varphi$ is provable in* $\mathbf{Clt}(\mathcal{P})$ *iff* $\varphi \in \mathbf{Mod}(\mathcal{P})$*.*

*Proof.*

1. In one direction straightforward: it is easy to see that a formula, provable in $\mathbf{Clt}(\mathcal{P})$, is true in any model of $\mathcal{P}$.
2. We shall the use characterization of the minimal model proved in Theorem 2: a ground $\Sigma$-formula $\varphi$ is true in the minimal model iff $\varphi \in \mathfrak{I}_i$, for some $i$. Thus it suffices to show that all formulas from $\mathfrak{I}_i$ are provable in $\mathbf{Clt}(\mathcal{P})$, for all $i$. Using axioms for lists and the induction axiom of $\mathbf{Clt}(\mathcal{P})$, one can prove the following properties of $\mathbf{Clt}(\mathcal{P})$(compare with Definition 3.10).
   (a) For every ground term $t$ we have $\mathbf{Clt}(\mathcal{P}) \vdash t{=}t$.
   (b) $\mathbf{Clt}(\mathcal{P}) \vdash \varphi \wedge \psi$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$ and $\mathbf{Clt}(\mathcal{P}) \vdash \psi$.
   (c) $\mathbf{Clt}(\mathcal{P}) \vdash \varphi \vee \psi$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$ or $\mathbf{Clt}(\mathcal{P}) \vdash \psi$.
   (d) $\mathbf{Clt}(\mathcal{P}) \vdash \exists x \varphi(x)$, if for some term $t$ of the same sort as $x$, $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$.
   (e) $\mathbf{Clt}(\mathcal{P}) \vdash (\exists x \in s)\varphi(x)$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$ for some term $t \in s$.
   (f) $\mathbf{Clt}(\mathcal{P}) \vdash (\forall x \in s)\varphi(x)$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$ for every term $t \in s$.
   (g) $\mathbf{Clt}(\mathcal{P}) \vdash (\exists x \sqsubseteq s)\varphi(x)$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$ for some term $t \sqsubseteq s$.
   (h) $\mathbf{Clt}(\mathcal{P}) \vdash (\forall x \sqsubseteq s)\varphi(x)$, if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$ for every term $t \sqsubseteq s$.

   Using these properties, one can prove that $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$, for all $\varphi \in \mathfrak{I}_0$. Similarly, from the axioms for $\mathcal{P}$ and the above mentioned properties, one can prove, that if $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$, for all $\varphi \in \mathfrak{I}_i$, then $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$, for all $\varphi \in \mathfrak{I}_{i+1}$.

■

This theorem has an interesting corollary which shows that $\mathbf{Clt}(\mathcal{P})$, being based on classical logic, has some constructive properties:

**Corrollary 3.1 (Disjunction property and explicit definability property of $\mathbf{Clt}(\mathcal{P})$)**

1. *If $\varphi \vee \psi$ is a ground $\Sigma$-formula, and $\mathbf{Clt}(\mathcal{P}) \vdash \varphi \vee \psi$, then $\mathbf{Clt}(\mathcal{P}) \vdash \varphi$ or $\mathbf{Clt}(\mathcal{P}) \vdash \psi$.*
2. *If $\exists x \varphi(x)$ is a ground $\Sigma$-formula, and $\mathbf{Clt}(\mathcal{P}) \vdash \exists x \varphi(x)$, then $\mathbf{Clt}(\mathcal{P}) \vdash \varphi(t)$, for a ground term $t$.*

*Proof.* Straightforward from Theorem 3.

■

### 3.4 Constructive provability

**Definition 3.16 (System $\mathbf{Ilt}(\mathcal{P})$)** *The system $\mathbf{Ilt}(\mathcal{P})$ (intuitionistic list theory) has the same axioms and rules as $\mathbf{Clt}(\mathcal{P})$ but is based on intuitionistic predicate logic instead of the classical one.*

The intuitionistic variant of list theory was introduced in [Voronkov 86b]. In [Voronkov 91] we proved that it is constructive from the viewpoint of a constructive semantics, which in particular means that it has a variant of the explicit definability 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. Below we shall prove a stronger form of this property.

For (suitably represented) Horn clauses, provability in classical logic coincides with the provability in intuitionistic logic. We can prove the same result for the generalized programs, if we consider $\mathbf{Ilt}(\mathcal{P})$ instead of the intuitionistic predicate calculus.

**Theorem 4** *For a ground atomic $\Sigma$-formula $\varphi$, $\mathbf{Ilt}(\mathcal{P})\quad\varphi$ iff $\varphi\in\mathbf{Mod}(\mathcal{P})$.*

*Proof.* The same as for Theorem 3.

■

The theory $\mathbf{Clt}(\mathcal{P})$ has the disjunction property and the explicit definability property for ground $\Sigma$-formulas. Below we show that $\mathbf{Ilt}(\mathcal{P})$ has this important properties for arbitrary formulas. The constructiveness of $\mathbf{Ilt}(\mathcal{P})$ is important, because it allows one to synthesize logic programs from proofs in this theory, in the style of [Voronkov 86c, Voronkov 86d] or [Wiggins 91].

**Theorem 5 (Disjunction property and explicit definability property of $\mathbf{Ilt}(\mathcal{P})$)**

1. *If $\varphi\vee\psi$ is a ground formula, and $\mathbf{Ilt}(\mathcal{P})\vdash\varphi\vee\psi$, then $\mathbf{Ilt}(\mathcal{P})\vdash\varphi$ or $\mathbf{Ilt}(\mathcal{P})\vdash\psi$.*
2. *If $\exists x\varphi(x)$ is a ground formula, and $\mathbf{Ilt}(\mathcal{P})\vdash\exists x\varphi(x)$, then $\mathbf{Ilt}(\mathcal{P})\vdash\varphi(t)$, for a ground term $t$.*

*Proof.* We shall use the technique, introduced by Kleene [Kleene 62]. First we introduce a few definitions.

The set of *Harrop formulas* [Harrop 60] is defined as follows.

**Definition 3.17 (Harrop formulas)**

1. *Any atomic formula is a Harrop formula;*
2. *If $\varphi,\psi$ are Harrop formulae and $\chi$ is an arbitrary formula then the formulae $\varphi\wedge\psi$, $\forall x\varphi$, $\chi\supset\psi$, $\neg\chi$ are Harrop formulae.*

The relation $|$ (Kleene's slash) between sets of formulae and formulae is defined in the following way. During the definition we assume that $T\quad\varphi$ means $T|\varphi$ and $T\vdash\varphi$, where $\vdash$ stands for the provability in intuitionistic logic.

**Definition 3.18 (Kleene's slash $|$)**

1. *For atomic formulas $\varphi$, $T|\varphi$ iff $T\vdash\varphi$;*
2. *$T|\varphi\wedge\psi$ iff $T|\varphi$ and $T|\psi$;*
3. *$T|\varphi\vee\psi$ iff $T\quad\varphi$ or $T\quad\psi$;*
4. *$T|\varphi\supset\psi$ iff from $T\quad\varphi$ follows $T|\psi$;*
5. *$T|\neg\varphi$ iff not $T\quad\varphi$;*
6. *$T|\forall x\varphi(x)$ iff for every ground term $t$ we have $T|\varphi(t)$;*
7. *$T|\exists x\varphi(x)$ iff for some ground term $t$, we have $T\quad\varphi(t)$.*

The relation  is identical to Kleene's slash from [Kleene 62] except for minor changes for the quantifier cases.

As in [Kleene 62] it is possible to prove that logic obtained from intuitionistic logic by adding a set of formulae $S$ as axioms has the disjunction property and the explicit definability property iff for any $\varphi \in S$ we have $S \,|\, \varphi$.

Thus it suffices to show that for any axiom $\varphi$ of $\mathbf{Ilt}(\mathcal{P})$ we have $\mathbf{Ilt}(\mathcal{P}) \,|\, \varphi$.

1. All axioms for $\mathcal{P}$ are Harrop formulas. For Harrop formulas the proof is trivial as in e.g. [Kleene 62].
2. Axioms for lists. Consider, for example, the formula $r \in \texttt{[}s \,|\, t\texttt{]} \equiv (r{=}s \vee r \in t)$. It suffices to show that $\mathbf{Ilt}(\mathcal{P}) \,|\, r \in \texttt{[}s \,|\, t\texttt{]} \equiv (r{=}s \vee r \in t)$, for any ground terms $r, s, t$ of appropriate sorts. To this end assume $\mathbf{Ilt}(\mathcal{P}) \quad r \in \texttt{[}s \,|\, t\texttt{]}$. Thus, in particular, $\mathbf{Ilt}(\mathcal{P}) \vdash r \in \texttt{[}s \,|\, t\texttt{]}$. By Theorem 4 the formula $r \in \texttt{[}s \,|\, t\texttt{]}$ is true in the minimal model. Hence, either $r$ is identical to $s$, or $r \in t$. Applying Theorem 4 once more, we get that either $\mathbf{Ilt}(\mathcal{P}) \vdash r{=}s$ or $\mathbf{Ilt}(\mathcal{P}) \vdash r \in t$. In both cases $\mathbf{Ilt}(\mathcal{P}) \,|\, r{=}s \vee r \in t$.
3. Induction axioms. One has to show that $\mathbf{Ilt}(\mathcal{P}) \,|\, \varphi(\texttt{[]}) \wedge \forall x \forall y (\varphi(y) \supset \varphi(\texttt{[}x \,|\, y\texttt{]})) \supset \forall y \varphi(y)$. To this end assume $\mathbf{Ilt}(\mathcal{P}) \quad \varphi(\texttt{[]})$ and $\mathbf{Ilt}(\mathcal{P}) \quad \forall x \forall y (\varphi(y) \supset \varphi(\texttt{[}x \,|\, y\texttt{]}))$, and prove $\mathbf{Ilt}(\mathcal{P}) \,|\, \varphi(t)$ for every ground term $t$ of the sort $\mathsf{list}$.

   The proof is by induction on the length of $t$.

   (a) $t{=}\texttt{[]}$. Straightforward from assumptions.

   (b) $t{=}\texttt{[}r \,|\, s\texttt{]}$. The induction hypothesis gives $\mathbf{Ilt}(\mathcal{P}) \quad \varphi(s)$. From this and $\mathbf{Ilt}(\mathcal{P}) \quad \forall x \forall y (\varphi(y) \supset \varphi(\texttt{[}x \,|\, y\texttt{]}))$ one can easily show $\mathbf{Ilt}(\mathcal{P}) \,|\, \varphi(\texttt{[}r \,|\, s\texttt{]})$.

   The other cases are considered similarly.

■

If we extend the system of sorts to include sorts from [Voronkov 92a], we may also prove these properties for the system with the foundation axiom [Barwise 75, Goncharov 85].

### 3.5 The natural calculus

We call this calculus natural because it gives a natural semantics to the 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 $\Sigma$-programs [Goncharov 85] was introduced in [Voronkov 87] with the aim of showing that $\Sigma$-programs can be efficiently executed using unification instead of the exhaustive search.

Another reason for introducing the natural calculus as an intermediate semantics between the denotational (model theoretic) semantics and the procedural (SLDB-resolution of Section 4) semantics, is that the least fixedpoint construction does not directly correspond to computations, as in the case of SLD-resolution [Lloyd 84]. Indeed, from the results of Section 5 it follows that there are non-recursive generalized logic programs, consisting of only one definition, which can express arbitrary computable sets. For such programs the least fixedpoint construction stabilizes on the first step ($\mathfrak{I}_1$), which does not correspond to real computations. The natural semantics introduced here does reflect computations by SLDB-resolution, which is the main procedure to execute generalized logic programs.

**Definition 3.19 (Calculus $\mathbf{Nat}(\mathcal{P})$)** *Calculus* $\mathbf{Nat}(\mathcal{P})$ *consists of the axioms and inference rules given in Figure 3.2 on page 19.*

Figure 3.2: Natural calculus $\mathbf{Nat}(\mathcal{P})$

1. Axioms are all formulas of the form `t=t`, where `t` is a ground term.
2. Rules for $\mathcal{P}$:

   $$\frac{\varphi}{A},$$

   if $A$`:-`$\varphi$ belongs to $\mathcal{P}^*$.
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 $\alpha$, $t$ is a term of the same sort.
5. Rules for the bounded quantifiers (here $t$ is a term of the sort $\mathsf{list}$, $s$ is an arbitrary term):

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

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

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

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

**Theorem 6** *For any ground $\Sigma$-formula $\varphi$, $\varphi \in$* $\mathbf{Mod}$*($\mathcal{P}$) iff* $\mathbf{Nat}(\mathcal{P}) \vdash \varphi$.

*Proof.*

1. It is easy to see that the rules of $\mathbf{Nat}(\mathcal{P})$ are admissible in $\mathbf{Clt}(\mathcal{P})$. Thus by Theorem 3 we have that $\mathbf{Nat}(\mathrm{P}) \vdash \varphi$ implies $\varphi \in \mathbf{Mod}(\mathcal{P})$.
2. Consider interpretations $\mathfrak{I}_i$ from Theorem 2. Obviously, all atomic formulas from $\mathfrak{I}_0$ are provable in $\mathbf{Nat}(\mathcal{P})$ (by axioms for equality). Rules for the connectives, the existential quantifier and bounded quantifiers allow all $\Sigma$-formulas from an interpretation to be proved from atomic formulas, true in this interpretation (see Definition 3.10 of interpretations). Finally, rules for $\mathcal{P}$ allow all atomic formulas of $\mathfrak{I}_{i+1}$ to be obtained from formulas of $\mathfrak{I}_i$. Application of Theorem 2 on coincidence of $\mathbf{Lfp}(\mathcal{P})$ with $\mathbf{Mod}(\mathcal{P})$ concludes the proof.

$\blacksquare$

# 4 Procedural semantics: SLDB-resolution

The natural calculus of Section 3.5 represents in a declarative way the main 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 two-sorted terms in Section 4.1 and SLDB-resolution in Section 4.2.

## 4.1 Unification

The sorts introduced here require a special unification algorithm, which reflects two-sorted structures. However the algorithm is not very different from the standard unification algorithm, and we shall only sketch the differences. These are:

1. Unification of a variable `v` of the sort $\mathfrak{list}$ with a variable `u` of the sort $\mathfrak{univ}$ gives the substitution `[u←v]` (but not `[v←u]`).
2. Unification of a variable `v` of the sort $\mathfrak{list}$ with a non-variable term different from `[]` and `[s|t]` fails.

One can easily prove standard statements about unification [Eder 85] for our two-sorted structures[1].

## 4.2 SLDB-resolution

In this section we give a procedural semantics of our language which generalizes SLD-resolution for Horn clause programs. We call it *SLDB-resolution* (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.

**Definition 4.1 (Goals)** *A goal is any list of $\Sigma$-formulas.*

We assume that the reader is familiar with the notion of SLD-resolution (see e.g. [Lloyd 84]).

**Definition 4.2 (Computation rule)** *The computation rule is a function from the set of all non-empty goals to the set of $\Sigma$-formulas such that the value of the function on a goal is a formula, called the selected formula, in that goal.*

**Definition 4.3 (Successor)** *Let $\varphi_1, \dots, \varphi_n$ be a goal, $\mathfrak{R}$ a computation rule and $\varphi_i$ be the selected formula in that goal under $\mathfrak{R}$. Then the goal $(\varphi_1, \dots, \varphi_{i-1}, \Gamma, \varphi_{i+1}, \dots, \varphi_n)\theta$, where $\theta$ is a substitution and $\Gamma$ is a list of $\Sigma$-formulas, is a* successor *of the goal $\varphi_1, \dots, \varphi_n$ with the substitution $\theta$ under the computation tule $\mathfrak{R}$ iff one of the following conditions holds:*

*1. $\varphi_i$ is an atom $\mathtt{t}_1 \mathtt{=} \mathtt{t}_2$, $\Gamma$ is empty, $\theta$ is a most general unifier of $\mathtt{t}_1$ and $\mathtt{t}_2$.*

[1] See also [Walther 90] for more references on order-sorted unification

2. $\varphi_i$ *takes the form* $\psi_1 \vee \ldots \vee \psi_n$, $\Gamma$ *is* $\psi_i$, $i \in \{1,\ldots,n\}$, $\theta$ *is the empty substitution.*

3. $\varphi_i$ *takes the form* $\psi_1 \wedge \ldots \wedge \psi_n$, $\Gamma$ *is* $\psi_1,\ldots,\psi_n$, $\theta$ *the empty substitution.*

4. $\varphi_i$ *takes the form* $\exists x \psi(\bar{x})$, $\Gamma$ *is* $\psi(\bar{y})$, $\theta$ *the empty substitution,* $\bar{y}$ *are new variables of the same sort as* $\bar{x}$.

   *In the following* $y$ *is a new (not occurring in the original goal) variable of the sort* $\mathfrak{list}$, $z$ *a new variable of the sort* $\mathfrak{univ}$.

5. $\varphi_i$ *takes the form* $(\exists x{\in}t)\psi(x)$, $\Gamma$ *is* $\psi(z)$, $\theta$ *a most general unifier of* $t$ *and* $\texttt{[}z\,|\,y\texttt{]}$.

6. $\varphi_i$ *takes the form* $(\exists x{\in}t)\psi(x)$, $\Gamma$ *is* $(\exists x{\in}y)\psi(x)$, $\theta$ *is a most general unifier of* $t$ *and* $\texttt{[}z\,|\,y\texttt{]}$.

7. $\varphi_i$ *takes the form* $(\forall x{\in}t)\psi(x)$, $\Gamma$ *is empty,* $\theta$ *is a most general unifier of* $t$ *and* $\texttt{[]}$.

8. $\varphi_i$ *takes the form* $(\forall x{\in}t)\psi(x)$, $\Gamma$ *is* $\psi(z)$,$(\forall x{\in}y)\psi(x)$, $\theta$ *is a most general unifier of the pair* $(x,t)$ *and* $(z,\texttt{[}z\,|\,y\texttt{]})$.

9. $\varphi_i$ *takes the form* $(\exists x{\sqsubseteq}t)\psi(x)$, $\Gamma$ *is* $\psi(x)$, $\theta$ *is the substitution* $[x{\leftarrow}t]$.

10. $\varphi_i$ *takes the form* $(\exists x{\sqsubseteq}t)\psi(x)$, $\Gamma$ *is* $(\exists x{\sqsubseteq}y)\psi(x)$, $\theta$ *is a most general unifier of* $t$ *and* $\texttt{[}z\,|\,y\texttt{]}$.

11. $\varphi_i$ *takes the form* $(\forall x{\sqsubseteq}t)\psi(x)$, $\Gamma$ *is* $\psi(\texttt{[]})$, $\theta$ *is a most general unifier of* $t$ *and* $\texttt{[]}$.

12. $\varphi_i$ *takes the form* $(\forall x{\sqsubseteq}t)\psi(x)$, $\Gamma$ *is* $\psi(\texttt{[}z\,|\,y\texttt{]})$,$(\forall x{\sqsubseteq}y)\psi(x)$, $\theta$ *is a most general unifier of the pair* $(x,t)$ *and* $(\texttt{[}z\,|\,y\texttt{]},\texttt{[}z\,|\,y\texttt{]})$.

13. $\varphi_i$ *is an atom* $P(\bar{t})$, $\Gamma$ *is* $\varphi(\bar{t})$, $\theta$ *is the empty substitution, if* $\mathcal{P}$ *contains a clause of the form* $P(\bar{x})\texttt{:-}\varphi(\bar{x})$.

**Definition 4.4 (SLDB-derivation)** *Let* $\mathfrak{R}$ *be a computation rule. An* <u>*SLDB-derivation*</u> *under* $\mathfrak{R}$ *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* $\theta$ *such that*

1. $\Gamma_i$ *is a successor of* $\Gamma_{i-1}$ *with the substitution* $\theta$ *under* $\mathfrak{R}$;

2. $\theta_i = \theta \circ \theta_{i-1}$.

**Definition 4.5 (Provability by SLDB-resolution)** *We say that a goal* $\Gamma$ *is* <u>*provable*</u> *under a computation rule* $\mathfrak{R}$ *with the answer substitution* $\theta$ *iff there is an SLDB-derivation under this computation rule, which starts from* $\langle\Gamma,\varepsilon\rangle$ *and ends at* $\langle\Lambda,\theta\rangle$, *where* $\varepsilon$ *is the empty substitution and* $\Lambda$ *is the empty goal.*

SLDB-resolution is different from SLD-resolution in several aspects. Firstly, special rules for handling bounded quantifiers are added, and secondly, some steps of SLDB-resolution are not explicitly mentioned in the usual definitions of SLD-resolution. These correspond to steps which are implicit in SLD-resolution.

Let us give an example of SLDB-derivation.

**Example 4.1** *Let* $\mathcal{P}$ *be the program obtained from the program from Example 2.5 by adding two facts*

```
arc(a,b).
arc(b,c).
```

*We remember that these clauses denote the program*

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

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

singleton(Y) :-
   ∃X(Y=[X]).

connected(U) :-
   ∃X∃Y∃Z(U=[X,Y|Z]∧arc(X,Y)).

start(X,Y) :-
   ∃Xs(Y=[X|Xs]).

finish(X,Xs) :-
   (∃S⊑Xs)(S=[X]).

arc(X,Y) :-
   (X=a∧Y=b) ∨ (X=b∧Y=c).
```

*Then, given the query*

```
?-route(X,c,Y).
```

*the following is an SLDB-derivation. The computed answer substitution is written at the end of each goal. The selected formula is put in a box.*

1. $\boxed{\texttt{route(X,c,Y)}}$ `[]`
2. $\boxed{\texttt{path(Y)}\land\texttt{start(X,Y)}\land\texttt{finish(c,Y)}}$ `[]`
3. `path(Y),`<br>$\boxed{\texttt{start(X,Y)}}$,<br>`finish(c,Y)` `[]`
4. `path(Y),`<br>$\boxed{\exists\texttt{Xs(Y=[X|Xs])}}$,<br>`finish(c,Y)` `[]`
5. `path(Y),`<br>$\boxed{\texttt{Y=[X|Z]}}$,<br>`finish(c,Y)` `[]`
6. $\boxed{\texttt{path([X|Z])}}$,<br>`finish(c,[X|Z])` `[Y←[X|Z]]`
7. $\boxed{(\forall\texttt{S}\sqsubseteq\texttt{[X|Z])(S=[]}\lor\texttt{singleton(S)}\lor\texttt{connected(S))}}$,<br>`finish(c,[X|Z])` `[Y←[X|Z]]`

8. $\boxed{\mathtt{[X|Z]=[]}\lor\mathtt{singleton([X|Z])}\lor\mathtt{connected([X|Z])}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{Z})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[X|Z])}\qquad[\mathtt{Y}\leftarrow\mathtt{[X|Z]}]$

9. $\boxed{\mathtt{connected([X|Z])}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{Z})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[X|Z])}\qquad[\mathtt{Y}\leftarrow\mathtt{[X|Z]}]$

10. $\boxed{\exists\mathtt{U}\exists\mathtt{V}\exists\mathtt{W}(\mathtt{[X|Z]=[U,V|W]}\land\mathtt{arc(U,V)})}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{Z})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[X|Z])}\qquad[\mathtt{Y}\leftarrow\mathtt{[X|Z]}]$

11. $\boxed{\mathtt{[X|Z]=[U,V|W]}\land\mathtt{arc(U,V)}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{Z})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[X|Z])}\qquad[\mathtt{Y}\leftarrow\mathtt{[X|Z]}]$

12. $\boxed{\mathtt{[X|Z]=[U,V|W]}}$,
$\mathtt{arc(U,V)}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{Z})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[X|Z])}\qquad[\mathtt{Y}\leftarrow\mathtt{[X|Z]}]$

13. $\boxed{\mathtt{arc(U,V)}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{[V|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[U,V|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[U,V|W]},\mathtt{X}\leftarrow\mathtt{U}]$

14. $\boxed{(\mathtt{U=a}\land\mathtt{V=b})\lor(\mathtt{U=b}\land\mathtt{V=c})}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{[V|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[U,V|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[U,V|W]},\mathtt{X}\leftarrow\mathtt{U}]$

15. $\boxed{(\mathtt{U=a}\land\mathtt{V=b})}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{[V|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[U,V|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[U,V|W]},\mathtt{X}\leftarrow\mathtt{U}]$

16. $\boxed{\mathtt{U=a}}$,
$\mathtt{V=b}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{[V|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[U,V|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[U,V|W]},\mathtt{X}\leftarrow\mathtt{U}]$

17. $\boxed{\mathtt{V=b}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{[V|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[a,V|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[a,V|W]},\mathtt{X}\leftarrow\mathtt{a}]$

18. $\boxed{(\forall\mathtt{S}\sqsubseteq\mathtt{[b|W]})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})}$,
$\mathtt{finish(c,[a,b|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[a,b|W]},\mathtt{X}\leftarrow\mathtt{a}]$

19. $\boxed{\mathtt{[b|W]=[]}\lor\mathtt{singleton([b|W])}\lor\mathtt{connected([b|W])}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{W})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[a,b|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[a,b|W]},\mathtt{X}\leftarrow\mathtt{a}]$

20. $\boxed{\mathtt{connected([b|W])}}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{W})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[a,b|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[a,b|W]},\mathtt{X}\leftarrow\mathtt{a}]$

21. $\boxed{\exists\mathtt{X}\exists\mathtt{Y}\exists\mathtt{Z}(\mathtt{[b|W]=[X,Y|Z]}\land\mathtt{arc(X,Y)})}$,
$(\forall\mathtt{S}\sqsubseteq\mathtt{W})(\mathtt{S=[]}\lor\mathtt{singleton(S)}\lor\mathtt{connected(S)})$,
$\mathtt{finish(c,[a,b|W])}\qquad[\mathtt{Y}\leftarrow\mathtt{[a,b|W]},\mathtt{X}\leftarrow\mathtt{a}]$

22. $\boxed{\texttt{[b|W]=[X1,Y1|Z1]}\wedge\texttt{arc(X1,Y1)}}$,
(∀S⊑W)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b|W])   [Y←[a,b|W],X←a]

23. $\boxed{\texttt{[b|W]=[X1,Y1|Z1]}}$,
arc(X1,Y1),
(∀S⊑W)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b|W])   [Y←[a,b|W],X←a]

24. $\boxed{\texttt{arc(b,Y1)}}$,
(∀S⊑[Y1|Z1])(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,Y1|Z1])   [Y←[a,b,Y1|Z1],X←a]

25. $\boxed{(\texttt{b=a}\wedge\texttt{Y1=b})\vee(\texttt{b=b}\wedge\texttt{Y1=c})}$,
(∀S⊑[Y1|Z1])(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,Y1|Z1])   [Y←[a,b,Y1|Z1],X←a]

26. $\boxed{\texttt{b=b}\wedge\texttt{Y1=c}}$,
(∀S⊑[Y1|Z1])(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,Y1|Z1])   [Y←[a,b,Y1|Z1],X←a]

27. $\boxed{\texttt{b=b}}$,
Y1=c,
(∀S⊑[Y1|Z1])(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,Y1|Z1])   [Y←[a,b,Y1|Z1],X←a]

28. $\boxed{\texttt{Y1=c}}$,
(∀S⊑[Y1|Z1])(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,Y1|Z1])   [Y←[a,b,Y1|Z1],X←a]

29. $\boxed{(\forall \texttt{S}\sqsubseteq\texttt{[c|Z1]})(\texttt{S=[]}\vee\texttt{singleton(S)}\vee\texttt{connected(S)})}$,
finish(c,[a,b,c|Z1])   [Y←[a,b,c|Z1],X←a]

30. $\boxed{\texttt{[c|Z1]=[]}\vee\texttt{singleton([c|Z1])}\vee\texttt{connected([c|Z1])}}$,
(∀S⊑Z1)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,c|Z1])   [Y←[a,b,c|Z1],X←a]

31. $\boxed{\texttt{singleton([c|Z1])}}$,
(∀S⊑Z1)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,c|Z1])   [Y←[a,b,c|Z1],X←a]

32. $\boxed{\exists\texttt{X([c|Z1]=[X])}}$,
(∀S⊑Z1)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,c|Z1])   [Y←[a,b,c|Z1],X←a]

33. $\boxed{\texttt{[c|Z1]=[X1]}}$,
(∀S⊑Z1)(S=[]∨singleton(S)∨connected(S)),
finish(c,[a,b,c|Z1])   [Y←[a,b,c|Z1],X←a]

34. $\boxed{(\forall \texttt{S}\sqsubseteq\texttt{[]})(\texttt{S=[]}\vee\texttt{singleton(S)}\vee\texttt{connected(S)})}$,
finish(c,[a,b,c])   [Y←[a,b,c],X←a]

35. $\boxed{\texttt{[]=[]}\vee\texttt{singleton([])}\vee\texttt{connected([]))}}$,
finish(c,[a,b,c])   [Y←[a,b,c],X←a]

36. $\boxed{\texttt{[]=[]}}$,
finish(c,[a,b,c])   [Y←[a,b,c],X←a]

37. $\boxed{\texttt{finish(c,[a,b,c])}}$   [Y←[a,b,c],X←a]

38. $\boxed{(\exists \mathtt{S}\sqsubseteq\mathtt{[a,b,c]})\mathtt{(S=[c])}}$ $\mathtt{[Y\leftarrow[a,b,c],X\leftarrow a]}$

39. $\boxed{(\exists \mathtt{S}\sqsubseteq\mathtt{[b,c]})\mathtt{(S=[c])}}$ $\mathtt{[Y\leftarrow[a,b,c],X\leftarrow a]}$

40. $\boxed{(\exists \mathtt{S}\sqsubseteq\mathtt{[c]})\mathtt{(S=[c])}}$ $\mathtt{[Y\leftarrow[a,b,c],X\leftarrow a]}$

41. $\boxed{\mathtt{[c]=[c]}}$ $\mathtt{[Y\leftarrow[a,b,c],X\leftarrow a]}$

42. $\Lambda$ $\mathtt{[Y\leftarrow[a,b,c],X\leftarrow a]}$

The derivation is quite lengthy for the reasons explained above: some steps of SLDB-resolution correspond to steps hidden in SLD-derivations. One can use WAM-based compilation [Ait Kaci 90] or partial evaluation [PE 91] for more efficient execution models. It is interesting to build a WAM-based implementation of logic programming with bounded quantifiers.

Theorem 7 below states that SLDB-resolution is independent of the computation rule. Proof will be similar to that of [Lloyd 84]. Before proving this theorem we shall prove a technical lemma.

**Lemma 4.1 (Switching lemma)** *Let $\mathfrak{G} = \Gamma, \varphi, \psi$ be a goal provable by SLDB-resolution. Let $\mathfrak{R}_1, \mathfrak{R}_2$ be two computation rules such that $\mathfrak{R}_1(\Gamma, \varphi, \psi) = \varphi$ and $\mathfrak{R}_2(\Gamma, \varphi, \psi) = \psi$. Let $\mathfrak{G}_1 = (\Gamma, \Delta, \psi)\theta_1$ be a successor of $\mathfrak{G}$ under $\mathfrak{R}_1$, which is provable by SLDB-resolution. Then there exist computation rules $\mathfrak{R}'_1$ and $\mathfrak{R}'_2$ such that for every successor $\mathfrak{G}_{12}$ of $\mathfrak{G}_1$ under $\mathfrak{R}'_1$ there is a successor $\mathfrak{G}_2 = (\Gamma, \varphi, \Sigma)\theta_2$ of $\mathfrak{G}$ under $\mathfrak{R}_2$ such that $\mathfrak{G}_{12}$ is also a successor of $\mathfrak{G}_2$ under $\mathfrak{R}'_2$. (See the picture below).*

$$\begin{tikzcd}
\mathfrak{G} \arrow[d, "\mathfrak{R}_1{,}\theta_1"'] & & \mathfrak{G} \arrow[d, dashed, "\mathfrak{R}_2{,}\theta_2"] \\
\mathfrak{G}_1 \arrow[dr, "\mathfrak{R}'_1"'] & & \mathfrak{G}_2 \arrow[dl, dashed, "\mathfrak{R}'_2"] \\
& \mathfrak{G}_{12} &
\end{tikzcd}$$

*Proof.* Straightforward, but tedious, by case analysis on the structure of the formulas $\varphi, \psi$. In all the cases we let $\mathfrak{R}'_1$ be any computation rule with $\mathfrak{R}_1(\mathfrak{G}_1) = \psi\theta_1$ and $\mathfrak{R}'_2$ be any computation rule with $\mathfrak{R}_2(\mathfrak{G}_2) = \varphi\theta_2$.

$$\begin{tikzcd}
\Gamma, \boxed{\varphi}, \psi \arrow[d, "\mathfrak{R}_1{,}\theta_1"'] & & \Gamma, \varphi, \boxed{\psi} \arrow[d, dashed, "\mathfrak{R}_2{,}\theta_2"] \\
(\Gamma\theta_1, \Delta\theta_1, \boxed{\psi\theta_1}) \arrow[dr, "\mathfrak{R}'_1"'] & & (\Gamma\theta_2, \boxed{\varphi\theta_2}, \Sigma\theta_2) \arrow[dl, dashed, "\mathfrak{R}'_2"] \\
& (\Gamma\theta, \Delta\theta, \Sigma\theta) &
\end{tikzcd}$$

We consider only some cases out of the 81 possible combinations.

1. Both $\varphi, \psi$ are equalities $s_1{=}s_2$ and $t_1{=}t_2$. In this case $\theta_1$ is a most general unifier of $s_1, s_2$. Terms $t_1\theta_1$ and $t_2\theta_1$ are unifiable, since $\mathfrak{G}_1$ is provable. Thus, $t_1$ and $t_2$ are unifiable (with a most general unifier $\theta_2$). Let $\theta$ be a most general unifier of pairs $\langle s_1, t_1\rangle$ and $\langle s_2, t_2\rangle$. By taking $\mathfrak{G}_{12} = \Gamma\theta$ we conclude the proof.
2. $\varphi$ takes the form $t_1{=}t_2$, $\psi$ takes the form $(\forall x{\in}t)\chi(x)$. Let $\theta_1$ be a most general unifier of $t_1$ and $t_2$. Consider all possible cases.
   (a) $t\theta_1 = \texttt{[]}$. Let $\theta_2$ be a most general unifier of $x$ and $\texttt{[]}$. Take $\mathfrak{G}_2 = (\Gamma, \varphi, \chi(x))\theta_2$.
   (b) $t\theta_1 = \texttt{[}r\,|\,s\texttt{]}$. Let $\theta_2$ be a most general unifier of $x$ and $\texttt{[}y\,|\,z\texttt{]}$ with new $y, z$. Take $\mathfrak{G}_2 = (\Gamma, \varphi, \chi(x), (\forall x{\in}z)\chi(x))\theta_2$.
   (c) $t\theta_1$ is a variable $v$ of the sort $\mathfrak{list}$. In this case there are two possible successors of $\mathfrak{G}_1$ under $\mathfrak{R}_1$.
      i. $\Sigma = \chi(\texttt{[]})$. In this case take $\theta_2, \mathfrak{G}_2$ as in case 2a.
      ii. $\Sigma = \chi(\texttt{[}y\,|\,z\texttt{]}), (\forall x{\in}z)\chi(x)$. In this case take $\theta_2, \mathfrak{G}_2$ as in case 2b.
3. $\varphi$ takes the form $\varphi_1 \vee \ldots \vee \varphi_n$ and $\Delta$ is $\varphi_i$. In this case for every possible successor $\mathfrak{G}_{12} = (\Gamma, \Delta, \Sigma)\theta$ of $\mathfrak{G}_1$ under $\mathfrak{R}'_1$ take $\mathfrak{G}_2 = (\Gamma, \varphi, \Sigma)\theta$.

The other cases are considered more or less similarly.

■

**Theorem 7 (Independence of the computation rule)** *If a goal $\mathfrak{G}$ is provable with a substitution $\theta$ under a computation rule $\mathfrak{R}_1$, than $\mathfrak{G}$ is provable with $\theta$ under any other computation rule $\mathfrak{R}_2$.*

*Proof.* We shall prove an even stronger statement: if there is a proof of $\mathfrak{G}$ under $\mathfrak{R}_1$ with $\theta$ of length $n$, then there is a proof of $\mathfrak{G}$ under $\mathfrak{R}_2$ with $\theta$ of length $n$. The proof is by induction on $n$.

1. $n = 0, 1$. In this case $\mathfrak{G}$ consists of at most one formula, and all computation rules behave equally on $\mathfrak{G}$.
2. $n > 1$. We assume that for all $m < n$ the statement is true. Let $\mathfrak{G}$ takes the form $\Gamma, \varphi, \psi$, and let $\mathfrak{R}_1(\Gamma, \varphi, \psi) = \varphi$ and $\mathfrak{R}_2(\Gamma, \varphi, \psi) = \psi$. Then there is a successor $\mathfrak{G}_1 = (\Gamma, \Delta, \psi)\theta_1$ of $\mathfrak{G}$ under $\mathfrak{R}_1$, which is provable by SLDB-resolution in $n-1$ steps under any computation rule. Apply Lemma 4.1 and take as $\mathfrak{G}_2$ the successor of $\mathfrak{G}_1$ under $\mathfrak{R}'_1$ which occurs in the SLDB-derivation of $\mathfrak{G}_1$ of length $n-1$. Then we have that $\mathfrak{G}_2$ has an SLDB-derivation of length $n-1$ under some computation rule, and hence under $\mathfrak{R}_2$. Thus, $\mathfrak{G}$ has an SLDB-derivation of length $n$ under $\mathfrak{R}_2$.

■

We shall introduce a technical definition and prove several lemmas which are needed to prove soundness and completeness of SLDB-derivations.

**Definition 4.6** *Let $\mathfrak{G}_1, \mathfrak{G}_2$ be two goals. Then we write $\mathfrak{G}_1 \prec \mathfrak{G}_2$ iff there is a substitution $\theta$ such that $\mathfrak{G}_2\theta$ coincides with $\mathfrak{G}_1$.*

**Lemma 4.2** *Let $\mathfrak{G}_1, \mathfrak{G}_2$ be two goals and $\mathfrak{G}_1 \prec \mathfrak{G}_2$. Then for every successor $\mathfrak{G}'_1$ of $\mathfrak{G}_1$ under some computation rule there is a successor $\mathfrak{G}'_2$ of $\mathfrak{G}_2$ under a (possibly different) computation rule, such that $\mathfrak{G}'_1 \prec \mathfrak{G}'_2$.*

*Proof.* Straightforward by case analysis on the definition of a successor.

■

**Lemma 4.3** *Let the goal $\mathfrak{G}'$ be a successor of $\mathfrak{G}$ with the substitution $\theta_1$. Then for every substitution $\theta_2$, the goal $\mathfrak{G}'\theta_2$ is a successor of $\mathfrak{G}\theta_1\theta_2$ with the empty substitution.*

*Proof.* Consider, for example, the case when the $\mathfrak{G}$ takes the form $\Gamma, t_1{=}t_2$. In this case $\theta_1$ is a most general unifier of $t_1$ and $t_2$. We have that $t_1\theta_1\theta_2 = t_2\theta_1\theta_2$. The other cases are similar.

■

**Lemma 4.4** *Let a goal $\mathfrak{G}$ be provable by SLDB-resolution with a substitution $\theta$. Then any goal $\mathfrak{G}'$ with $\mathfrak{G}' \prec \mathfrak{G}\theta$ is provable with the empty substitution.*

*Proof.* Straightforward by repeated applications of Lemma 4.3.

■

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

**Theorem 8 (Soundness and completeness of SLDB-resolution)** *The following statements are true:*

**Soundness** *If a $\Sigma$-formula $\varphi$ is provable with the answer substitution $\theta$, then the formula $\forall(\varphi\theta)$ is true in* **Mod***($\mathcal{P}$).*

**Completeness** *If $\varphi$ is a $\Sigma$-formula and $\psi$ is its ground instance true in* **Mod***($\mathcal{P}$), then there are substitutions $\theta, \theta_1$, such that $\varphi$ is provable with the answer substitution $\theta$ and $\varphi\theta\theta_1$ is identical to $\psi$.*

*Proof.*

**Soundness** By Theorem 6 it is sufficient to prove that every ground instance of $\varphi$ is provable in **Nat**($\mathcal{P}$). It is easy to see that for the ground formulas all steps of SLDB-resolution, except for the case of unrestricted quantifiers, correspond to the rules of the natural calculus. Assume now that a goal $\mathfrak{G}$ takes the form $\Gamma, \exists x\psi(x)$, and $\psi(x)$ is the selected formula. In this case the successor of the goal is $\Gamma, \psi(y)$. Let it be provable with the answer substitution $\theta_1$. Let $\theta_2$ be any substitution such that $y\theta_1\theta_2$ is a ground term, say, $t$. Then, by Lemma 4.4, $\Gamma, \psi(t)$ is provable. In this case we can apply the rule for the unrestricted quantifiers of **Nat**($\mathcal{P}$).

**Completeness** Applying Theorem 6 we obtain that $\varphi$ is provable in $\mathbf{Nat}(\mathcal{P})$. The rest of the proof is straightforward by induction on lengths of derivations in $\mathbf{Nat}(\mathcal{P})$ using Theorem 7 and Lemma 4.2.

■

**Corrollary 4.1** *If $\varphi$ is a ground $\Sigma$-formula, then the following conditions are equivalent.*

1. *$\varphi$ is provable (with the empty substitution $\varepsilon$);*
2. *$\varphi$ is true in $\mathbf{Mod}(\mathcal{P})$.*

*Proof.* Straightforward from Theorem 8 and the fact that a ground formula can be provable only with the empty substitution.

■

# 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 Horn clause programs obtained may 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 program, which means that it can be expressed by a generalized program consisting of only one nonrecursive definition. We also prove that this cannot be achieved using only bounded quantifiers or only unrestricted existential quantifiers.

## 5.1 Translation to Horn clauses

In this section we will show that generalized logic programs can naturally be translated into Horn clause programs by adding new predicate symbols. A similar translation can be done for the {log} language of [Dovier 91], but in this language the only allowed bounded quantifier is that over elements of a set, which corresponds to our $(\forall x \in t)$. We prove correctness and completeness of the translation. The existence of the translation is not surprising, because Horn clauses form a universal programming language (a language in which all 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 show that it cannot be avoided in general. The original program and the translated program are equivalent in a strong sense: w.r.t. computed answer substitutions.

This equivalence is similar to the equivalence in *S-semantics*, introduced in [Falashi 89]. The equivalence in S-semantics entails a weaker equivalence in the sense of truth in the least Herbrand models.

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 bounded quantifiers.

The details of the translation are well known in cases of disjunction, conjunction and the unrestricted existential quantifier. Bounded quantifiers will be translated using iterative definitions. Suppose that we have a generalized logic program $\mathcal{P}$. We shall define its translation — a Horn clause program $\widehat{\mathcal{P}}$ in the following way. If there is a non-Horn definition in $\mathcal{P}$, we change it to one or more definitions according to the rules given in Figure 5.1 on page 31 until we get a Horn clause program.

**Theorem 9** *For any generalized logic program $\mathcal{P}$, $\mathcal{P}$ is equivalent to $\widehat{\mathcal{P}}$ in the following sense. For any $\Sigma$-formula $\varphi$ in the language of the first program, $\varphi$ is provable by SLDB-resolution from $\mathcal{P}$ with the substitution $\theta$ iff it is provable by SLDB-resolution from $\widehat{\mathcal{P}}$ with the same substitution.*

*Proof.* Let $\psi(\bar{x})$ be an arbitrary formula occurring in the body of a clause in $\mathcal{P}$. We note that there is a unique predicate symbol, corresponding to it in the translated program. (For example, in the case of conjunction $\wedge$, `D,E` correspond to `B,C`). We denote these predicate symbols by $P_\psi$. By induction on the lengths of derivations

Figure 5.1: Translation of bounded quantifiers into Horn clause logic programs

| **Sentence:** | **Its translation:** |
|---|---|
| A($\bar{x}$) :- B($\bar{x}$)$\wedge$C($\bar{x}$) | A($\bar{x}$) :- D($\bar{x}$),E($\bar{x}$).<br>D($\bar{x}$) :- B($\bar{x}$).<br>E($\bar{x}$) :- C($\bar{x}$). |
| A($\bar{x}$) :- B($\bar{x}$)$\vee$C($\bar{x}$) | A($\bar{x}$) :- B($\bar{x}$).<br>A($\bar{x}$) :- C($\bar{x}$). |
| A($\bar{x}$) :- $\exists$vB($\bar{x}$,v) | A($\bar{x}$) :- B($\bar{x}$,v). |
| A($\bar{x}$) :- ($\exists$y$\in$t)B($\bar{x}$,y) | A($\bar{x}$) :- D($\bar{x}$,t).<br>D($\bar{x}$,[y\|z]) :- B($\bar{x}$,y).<br>D($\bar{x}$,[y\|z]) :- D($\bar{x}$,z). |
| A($\bar{x}$) :- ($\forall$y$\in$t)B($\bar{x}$,y) | A($\bar{x}$) :- D($\bar{x}$,t).<br>D($\bar{x}$,[]).<br>D($\bar{x}$,[y\|z]) :- B($\bar{x}$,y),D($\bar{x}$,z). |
| A($\bar{x}$) :- ($\exists$y$\sqsubseteq$t)B($\bar{x}$,y) | A($\bar{x}$) :- D($\bar{x}$,t).<br>D($\bar{x}$,z) :- B($\bar{x}$,z).<br>D($\bar{x}$,[y\|z]) :- D($\bar{x}$,z). |
| A($\bar{x}$) :- ($\forall$y$\sqsubseteq$t)B($\bar{x}$,y) | A($\bar{x}$) :- D($\bar{x}$,t).<br>D($\bar{x}$,[]) :- B($\bar{x}$,[]).<br>D($\bar{x}$,[y\|z]) :- B($\bar{x}$,[y\|z]),D($\bar{x}$,z). |

Here D,E are new predicate symbols, $\bar{x}$ are all free variables of clauses in the left column. In the right column, y is a new variable of the sort $\mathfrak{univ}$ and z is a new variable of the sort $\mathfrak{list}$.

and using independence of the computation rule it is easy to prove the following statement about bisimulation of derivations:

> Let $\psi_1(\bar{x}), \ldots, \psi_n(\bar{x})$ be formulas occurring in bodies of clauses of $\mathcal{P}$. Then the goal $\psi_1(\bar{t_1}), \ldots, \psi_n(\bar{t_n})$ is provable by SLDB-resolution with a substitution $\theta$ in $\mathcal{P}$ iff the goal $P_{\psi_1(\bar{x})}(\bar{t_1}), \ldots, P_{\psi_n(\bar{x})}(\bar{t_n})$ is provable by SLDB-resolution with $\theta$ in $\widehat{\mathcal{P}}$.

The theorem easily follows from this statement.

∎

Let us note that this equivalence implies semantic equivalence — the least models of the two programs are identical. To extend this result to SLD-resolution one has to add the fact `x=x` expressing the equality relation.

## 5.2 A metainterpreter for Horn clause programs

In the examples from Section 2.3 we have already 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 non-recursive 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 facts of the form `A` are represented as facts of the form `rule(A,[])`. The definition of the metainterpreter is shown on Figure 5.2 on page 33.

**Theorem 10** *Let $\mathcal{P}$ be a Horn clause logic program. Let $\mathcal{R}$ is obtained from $\mathcal{P}$ by replacing each rule* `A:-B`$_1$`,...,B`$_n$ *(each fact* `A` *resp.) with facts* `rule(A,[B`$_1$`,...,B`$_n$`])` *(facts* `rule(A,[])` *resp.) and by adding the clauses from Figure 5.2. Then for any atomic $\varphi$, $\varphi$ is provable by SLDB-resolution from $\mathcal{P}$ with a substitution $\theta$ iff the goal* `call(`$\varphi$`)` *is provable from $\mathcal{R}$ with the same substitution.*

*Proof.* One can verify the following statements:

1. `trace_of_append(L1,L2,L3,L)` is true iff
   (a) `L` is a list every two consecutive elements of which take the forms `[[X|Xs],[Y|Ys]]` and `[Xs,Ys]`, for some terms `X,Xs,Y,Ys`.
   (b) The first and the last elements of `L` are `[L3,L1]` and `[[],L2]`.
2. `append(L1,L2,L3)` is true iff `L3` is obtained by concatenating `L1` and `L2`.
3. Let `S1` be a tuple of atomic formulas. Then `step_of_execution([S1,X2|X3]`, where `X2,X3` are variables not occurring in `S1`, is derivable with the substitution $\theta$ iff `X2`$\theta$ is a successor of S1 in the SLDB-derivation with the leftmost computation rule and `X3`$\theta$ is a variable not occurring in `X2`$\theta$`,S1`$\theta$.
4. Let `S` be a tuple of atomic formulas. Then `trace_of_execution([S|X])`, where `X` is a variable not occurring in `S`, is derivable with the substitution $\theta$ iff there is an SLD-derivation of the empty goal starting from `S` with the answer substitution $\gamma$ which coincides with $\theta$ on the set of variables of `S`.
5. `call(G)` is provable with the substitution $\theta$ iff `G` is provable by SLD-resolution with the answer substitution $\theta$.

Figure 5.2: Non-recursive metainterpreter for Horn clause programs

```
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]).
```

■

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):

**Corrollary 5.1** *For every computable set $S \subseteq HU$ there exists a non-recursive generalized logic program $\mathcal{P}$, defining a predicate* P*, such that for every* $\mathtt{t} \in HU$ *we have* $\mathtt{P(t)} \in \mathbf{Mod}(\mathcal{P}) \Leftrightarrow \mathtt{t} \in S$.

*Proof.* For Horn clause logic programs in was proved in [Tärnlund 77] for a Herbrand universe representing natural numbers and in [Andréka 78, Voronkov 92b] for arbitrary Herbrand universes. Using this fact, we apply Theorem 10 to find a generalized logic program with the required properties.

■

We can prove an even stronger property, connecting computability and answer substitutions computed by generalized logic programs.

**Corrollary 5.2** *Let $S$ be a set of tuples of terms of a finite signature $\sigma$. Then the following conditions are equivalent:*

1. *$S$ is computable and closed under renaming of variables.*
2. *There exists a non-recursive generalized logic program $\mathcal{P}$, defining a predicate* P*, such that* $\bar{\mathtt{t}} \in S$ *iff* $[\bar{\mathtt{x}} \leftarrow \bar{\mathtt{t}}]$ *is an answer substitution to* $\mathtt{P}(\bar{\mathtt{x}})$.

*Proof.* For Horn clause logic programs it was proved in [Voronkov 92b]. Then apply Theorem 10.

■

This corollary means that generalize logic programs can compute all computable predicates on the set of terms with variables, or w.r.t. S-semantics of [Falashi 89].

The metainterpreter from Figure 5.2 is in no way natural. One can more natural metainterpreter, using recursion:

```
call(G) :-
   rule(G,Gs) ∧
   (∀X∈Gs)call(X).
call(X=Y) :-
X = Y.
```

It is easy to see that the expressive power of Horn clause logic programs without recursion is very weak:

**Lemma 5.1** *Let $\mathcal{P}$ be a Horn clause logic program without recursion. Then there is a Horn clause program $\mathcal{P}_1$, consisting only of facts, which computes the same answer substitutions.*

*Proof.* Straightforward from the fixedpoint characterization of Horn clause logic programs [Apt 82, Lloyd 84].

∎

**Theorem 11** *Let $\mathcal{P}$ be a non-recursive Horn clause program. Then the minimal model computed by $\mathcal{P}$ is decidable (i.e. there is an algorithm verifying if a given ground atomic formula belongs to the model).*

*Proof.* Straightforward from Lemma 5.1.

∎

**Corrollary 5.3** *The class of predicates that are computable by non-recursive generalized logic programs is strictly larger then the class of predicates computable by non-recursive Horn clause programs.*

*Proof.* Straightforward from Corollary 5.1 and Theorem 11.

∎

It is interesting to find out the source of the expressive power of non-recursive generalized logic programs. The following theorem shows that the use of bounded quantifiers is essential:

**Theorem 12** *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.*

*Proof.* Note that a non-recursive generalized logic program without bounded quantifiers will be translated into a non-recursive Horn clause logic program (Figure 5.1). Then apply Theorem 9.

∎

However, bounded quantifiers only are not sufficient to express all computable predicates. To prove it, we consider now the expressive power of programs, all quantifiers of which are bounded. They are defined similar to $\Delta_0$-programs [Goncharov 85]:

**Definition 5.1 (Totally restricted programs)** *A <u>totally restricted program</u> is a program containing no occurrences of the unrestricted existential quantifier.*

Totally restricted programs are also interesting because they admit classical negation as explained below in Section 6. The next theorem shows that in presence of recursion we can still express all computable predicates without the use of unrestricted quantifiers:

**Theorem 13** [1] *For every computable set $S$ of ground terms there is a totally restricted program $\mathcal{P}$ defining a predicate $P$ such that for every ground term $t$, $P(t)$ belongs to* **Mod***($\mathcal{P}$) iff $t \in S$.*

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

*Proof.* By Corollary 5.1 and theorem on recursive completeness of Horn clause logic programs [Tärnlund 77, Voronkov 92b] all computable predicates on $HU$ can be expressed by a non-recursive generalized logic program $\mathcal{P}$. It is sufficient to show how to construct a generalized logic program $\mathcal{P}_1$ without unrestricted quantifiers, which computes the same relation $P$. To this end we will show how to transform each definition, which uses unrestricted quantification, into a definition without such quantification. To obtain $\mathcal{P}_1$, one should apply this transformation until we get rid of all unrestricted quantifiers. Using the transformation similar to that of Figure 5.1 we can restrict ourselves to clauses, in which all unrestricted quantifiers are not in the range of any other quantifiers or connectives.

Let

$$A(\bar{x}) \texttt{ :- } \exists y(B(\bar{x}, y)).$$

be such a clause. Let $a_1, \ldots, a_n$ be all constants and $f_1, \ldots, f_m$ all function symbols occurring in $\mathcal{P}$ (including `[]`, `[ | ]`). Let $C$ be a new predicate symbol. Consider the program, obtained from $\mathcal{P}$ by replacing the above clause by clauses with

$$\begin{array}{lll} A(\bar{x}) & \texttt{:-} & C(\bar{x}, \texttt{[]}). \\ C(\bar{x}, l) & \texttt{:-} & (\exists y \in l) B(\bar{x}, y) \ \vee \\ & & C(\texttt{[}a_1 \texttt{|} l\texttt{]} \ \vee \\ & & \ldots \\ & & C(\texttt{[}a_n \texttt{|} l\texttt{]}) \ \vee \\ & & (\exists y_1 \in l) \ldots (\exists y_k \in l) C(\texttt{[}f_1(y_1, \ldots, y_k) \texttt{|} l\texttt{]}) \ \vee \\ & & \ldots \\ & & (\exists y_1 \in l) \ldots (\exists y_p \in l) C(\texttt{[}f_m(y_1, \ldots, y_p) \texttt{|} l\texttt{]}). \end{array}$$

Let, for a list of terms $l$, $\mathfrak{ord}(l)$ means the number of occurrences of function symbols and constants in $l$. By definition of $C$, one can prove the following statement:

> If $C(\bar{t}, l)$ is true for a list $l$ with $\mathfrak{ord}(l) = j > 0$, then $C(\bar{t}, l_1)$ is true for a list $l_1$ with $\mathfrak{ord}(l_1) < j$.

From this it follows that if $C(\bar{t}, l)$ is true for a list of terms $l$, then $C(\bar{t}, \texttt{[]})$ is also true. Let $A(\bar{t})$ be true in the original program. Then there is a term $s$ such that $B(\bar{t}, s)$. From the definition of $C$ we have that $C(\bar{t}, \texttt{[}s\texttt{]})$ is also true. Hence, $C(\bar{t}, \texttt{[]})$ is true, and from the definition of $A$ in the new program, $A(\bar{t})$ is true in the new program.

It is straightforward to see also that, if $C(\bar{t}, l)$ is true in the second program, then $B(\bar{t}, s)$ is true for some $s$.

■

Now we consider the non-recursive case:

**Theorem 14** *Let $\mathcal{P}$ be a totally restricted program without recursion. Then the set* **Mod**$(\mathcal{P})$ *is decidable.*

*Proof.* It follows from the fact that in the absence of recursion and unrestricted quantifiers the lengths of SLD-derivations for any ground formula are limited.

■

However, totally restricted programs without recursion can be more expressive then Horn clause programs without recursion:

**Theorem 15** *There exists a non-recursive totally restricted program $\mathcal{P}$ such that for any non-recursive Horn clause program $\mathcal{P}$' we have*

$$\{P(t) \mid P(t) \in \mathbf{Mod}(\mathcal{P})\} \neq \{P(t) \mid P(t) \in \mathbf{Mod}(\mathcal{P}')\}$$

*Proof.* Consider the program $\mathcal{P}$ consisting of only one definition

```
P(x):-(∀y∈x)y=a
```

Assume that a non-recursive Horn clause program $\mathcal{P}$' gives the same minimal model. According to Lemma 5.1 we can assume that $\mathcal{P}$' consists only of facts. There is an infinite number of facts in $\mathbf{Mod}(\mathcal{P})$ of the form

```
P([])
P([a])
P([a,a])
P([a,a,a])
...
```

Thus there is a fact in $\mathcal{P}$', for which an infinite number of these facts are instances. This fact must be of the form

P([$\mathtt{t}_1$,...,$\mathtt{t}_n$|x]),

where `x` is a variable. Substituting `[[]]` for `x`, we obtain a fact, which is in the least model for $\mathcal{P}$', but not $\mathcal{P}$.

■

Horn clause programs without recursion can, in turn, be more expressive than totally restricted programs without recursion.

**Theorem 16** *There exists a non-recursive Horn clause program $\mathcal{P}$ such that for any non-recursive totally restricted program $\mathcal{R}$ we have*

$$\{P(t) \mid P(t) \in \mathbf{Mod}(\mathcal{P})\} \neq \{P(t) \mid P(t) \in \mathbf{Mod}(\mathcal{R})\}$$

*Proof.* Consider the Horn clause logic program consisting of the fact

```
P(f(x))
```

Its minimal model consists of all terms of the form `P(f(t))` for a suitable ground term `t`. Thus, there is an infinite number of terms on which `P` is true, and an infinite number of terms on which `P` is false. Assume that $\mathcal{P}$' gives the same minimal model. Since it is non-recursive, we can assume that it consists of only one definition

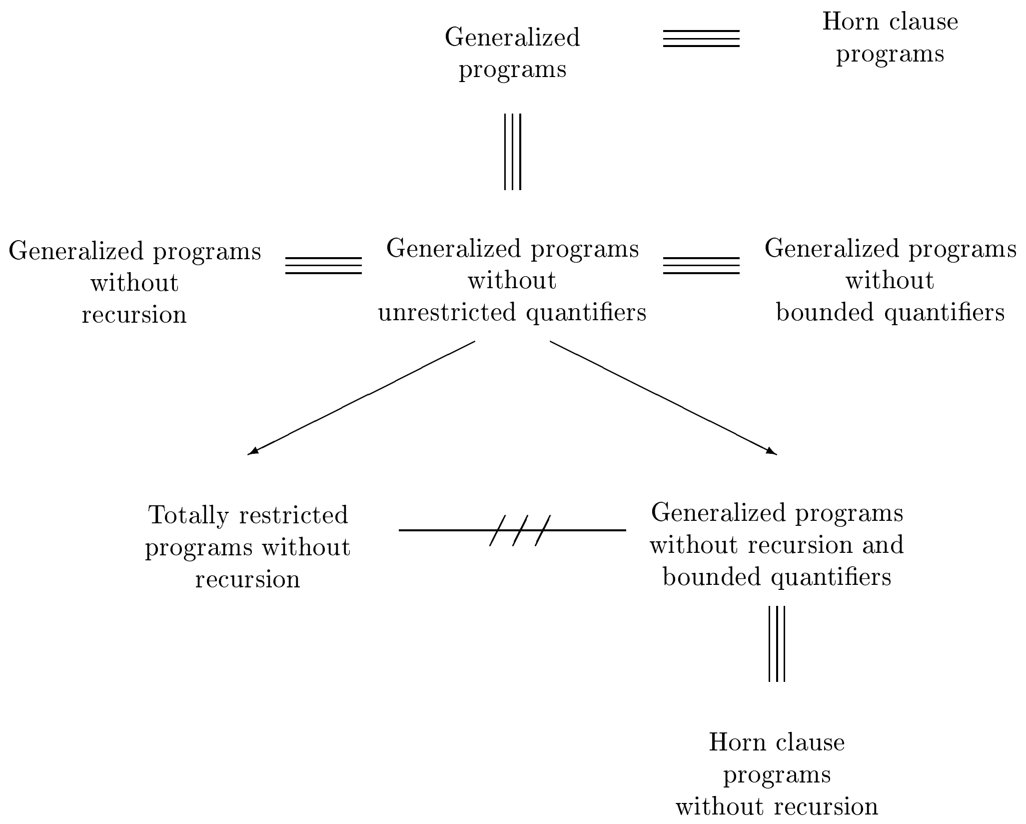
```
P(x) :- φ(x)
```

Since the variable `x` must be of the sort $\mathfrak{univ}$, the only bounded quantifiers in $\varphi$ have to be over finite lists [$\mathtt{t}_1$,...,$\mathtt{t}_n$], which may be changed to finite disjunctions and conjunctions. Thus, we can assume that $\varphi$ consists only of disjunctions and conjunctions of equalities with the only free variable `x`. It is easy to prove that such $\varphi$ is either true on only finite number of ground terms or is false on a finite number of ground terms.

■

The results of this section are summarized in Figure 5.3 on page 38.

Figure 5.3: Expressibility of subsets of Horn clause logic programs and generalized logic programs




Here

| | | |
|---|---|---|
| $L_1 \equiv L_2$ | denotes | "$L_1$ is equivalent to $L_2$"; |
| $L_1 \longrightarrow L_2$ | denotes | "$L_1$ is more expressive than $L_2$"; |
| $L_1$ —///— $L_2$ | denotes | "$L_1$ and $L_2$ are not compatible". |

# 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, iff the formula is not true on these elements.

Approaches to handling classical negation in logic programming are not (and cannot be) satisfactory. (An excellent survey of negation and computability can be found in [Sheperdson 87]). The main reason is very easy — there cannot be a sound and complete 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 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 a predicate defined by Horn clauses may be not computable. The usual solution of the computability problem is negation as failure, which is incomplete for classical negation.
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 *negatable* 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. From the viewpoint of recursion theory negatable predicates are exactly decidable predicates [Rogers 67]. However there is no syntactic criterion to recognize programs which define decidable predicates in the class of all programs. We shall introduce below a subclass of generalized logic programs which always defines only computable predicates. This class of programs is sufficiently rich, for instance, all examples of Section 2.3 are in this class.

Bounded quantifiers can easily be negated using the following equivalences:

$$\begin{aligned}
\neg(\forall x{\in}t)\varphi &\equiv (\exists x{\in}t)\neg\varphi \\
\neg(\exists x{\in}t)\varphi &\equiv (\forall x{\in}t)\neg\varphi \\
\neg(\forall x{\sqsubseteq}t)\varphi &\equiv (\exists x{\sqsubseteq}t)\neg\varphi \\
\neg(\exists x{\sqsubseteq}t)\varphi &\equiv (\forall x{\sqsubseteq}t)\neg\varphi
\end{aligned}$$

Let us note, that this property is already sufficient to use negation for a wide class of predicates. For instance, all examples of Section 2.3 can easily be transformed into non-recursive programs without unrestricted quantifiers. Some of the clauses of the examples contain unrestricted quantifiers, e.g.

```
ordered2([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 heads of clauses without turning them into expressions with bounded quantifiers, using Theorem 17 proved below.

Let us first give precise definitions.

**Definition 6.1 (Negatable predicates)** *A predicate $P$ defined by a generalized logic program $\mathcal{P}$ is called* <u>*negatable*</u> *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 $\bar{t}$ of ground terms from the Herbrand universe of $\mathcal{P}$, $P(\bar{t})$ is true in the minimal model for $\mathcal{P}$ iff $R(\bar{t})$ is false in the minimal model for $\mathcal{R}$.*

We assume that ≠ is the negation of =.

**Corrollary 6.1** *Let $\mathcal{P}$ be a totally restricted program without recursion and the equality = is negatable. Then any predicate, defined in $\mathcal{P}$, is negatable.*

> *Proof.* Using the equivalences of negations of bounded quantifiers from page 39 one can reduce all definitions of $\mathcal{P}$ to non-recursive definitions, where equality and inequality occur as the only undefined symbols.

■

We also note that the negation ≠ of equality = is negatable in every fixed Herbrand universe, so we shall freely use it in the rest of this section.

**Theorem 17** *Let* **N** *be the class of the generalized logic programs with the following properties:*

1. *No recursion is used in the programs from* **N***;*
2. *All occurrences of the unrestricted existential quantifier in the programs from* **N** *take the form*

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

   *where $\bar{\mathtt{x}}, y$ are all free variables of $\varphi, \psi$ and for any tuple of ground terms $\bar{\mathtt{s}}$ there is exactly one term $\mathtt{t}$ such that $\varphi(\bar{\mathtt{s}},\mathtt{t})$.*

*Then every defined in* **N** *predicate is negatable.*

> *Proof.* We have to show how to negate expressions containing existential quantifiers, i.e. how to express $\neg\exists y(\varphi(\bar{x},y)\wedge\psi(\bar{x},y))$ in the language of generalized logic programs. This negation is equivalent to $\forall y(\varphi(\bar{x},y)\supset\neg\psi(\bar{x},y))$. Let $\bar{t}$ be a tuple of ground terms. Then $\forall y(\varphi(\bar{t},y)\supset\neg\psi(\bar{t},y))$ means that $\neg\psi(\bar{t},y))$ is true for the $y$ for which $\varphi(\bar{t},y)$ is true. Thus, the formula $\forall y(\varphi(\bar{t},y)\supset\neg\psi(\bar{t},y))$ is equivalent to $\exists y(\varphi(\bar{t},y)\wedge\neg\psi(\bar{t},y))$. Since it holds for every tuple of ground terms $\bar{t}$, we conclude that $\neg\exists y(\varphi(\bar{x},y)\wedge\psi(\bar{x},y))$ is equivalent to $\exists y(\varphi(\bar{x},y)\wedge\neg\psi(\bar{x},y))$.

■

Theorem 17 is essentially related to functional computations. A similar treatment of negation can be found in [Naish 86].

Now we apply Theorem 17 to extensions of totally restricted programs.

**Definition 6.2 (Class $\mathrm{TR}_1$)** *Class $\mathrm{TR}_1$ of generalized logic programs is the class of programs whose clauses take the form*

$A(\bar{t})$ `:-` $\varphi$.

*where $\varphi$ is a $\Delta_0$-formula, $\bar{t}$ a tuple of terms (not necessarily variables).*

The class $\mathrm{TR}_1$ includes all totally restricted programs. However, there are programs in $\mathrm{TR}_1$, which are not totally restricted (when $\bar{t}$ includes non-variable terms). Theorem 16 from the previous section shows that $\mathrm{TR}_1$ is more expressible than the class of all totally restricted programs.

**Theorem 18** *Let $\mathcal{P}$ be a non-recursive generalized logic program from $\mathrm{TR}_1$. Then every predicate $P$ defined in $\mathcal{P}$ is negatable.*

*Proof.* Definitions of the form

$A(\bar{t})$ `:-` $\psi$.

denote

$A(\bar{y})$ `:-` $\exists\bar{x}(\bar{y}=\bar{t}\wedge\psi)$.

We note that the predicate

$\mathtt{nunif}_s\mathtt{(t)} \Leftrightarrow \mathtt{t}$ is not unifiable with $s$

can be expressed by a generalized Prolog program. For example, if all the symbols from the language are `a,f,h`, then $\mathtt{nunif}_{\mathtt{f(X,X)}}$ can be defined by

$\mathtt{nunif}_{\mathtt{f(X,X)}}\mathtt{(a)}$.
$\mathtt{nunif}_{\mathtt{f(X,X)}}\mathtt{(h(Y))}$.
$\mathtt{nunif}_{\mathtt{f(X,X)}}\mathtt{(f(Y,Z))}$ `:-` $\mathtt{Y} \neq \mathtt{Z}$.

The clause defining $A$ is equivalent to

$A(\bar{y})$ `:-` $\neg\mathtt{nunif}_t(\bar{y})\wedge\exists\bar{x}(\bar{y}=\bar{t}\wedge\psi)$.

Take = for $\varphi$ in Theorem 17.

■

Theorem 13 shows that there can be no correct and complete implementations of negation for totally restricted programs in general. However the (recursive) totally restricted programs have interesting properties related to negation as failure. In traditional logic programming there are approaches to solve the problem of negation by constructing programs which compute the finite failure set of a given Horn clause program (see e.g. [Sato 84]). However the programs generated in such a way may be very complicated even when the original programs are very simple. Programs with bounded quantifiers admit an elegant solution for constructing such a dual program.

First we define the finite failure set for a program $\mathcal{P}$:

**Definition 6.3 (Finite failure set $\mathfrak{FF}(\mathcal{P})$)** *The finite failure set $\mathfrak{FF}(\mathcal{P})$ of a program $\mathcal{P}$ is defined as follows. A goal $G$ belongs to $\mathfrak{FF}(\mathcal{P})$ iff there is a computation rule $\mathfrak{R}$ such that any SLDB-derivation under this computation rule, starting from the goal $G$ with the empty substitution, fails in a finite number of steps.*

*For a predicate $P$ defined in a program $\mathcal{P}$, the finite failure set $\mathfrak{FF}_{\mathcal{P}}(P)$ of $P$ in $\mathcal{P}$ is the set $\{\bar{t} \mid P(\bar{t}) \in \mathfrak{FF}(\mathcal{P})\}$.*

The notion of an SLDB-tree for a goal $\Gamma$ and a computation rule $\mathfrak{R}$ is defined similar to that of SLD-tree [Lloyd 84]:

**Definition 6.4 (SLDB-tree)** *Let $\mathcal{P}$ be a program, $G$ a goal and $\mathfrak{R}$ a computation rule. Then the SLDB-tree for $\mathcal{P}, G$ under $\mathfrak{R}$ is defined as follows:*

1. *Each node of the tree is a goal.*
2. *The root node is $G$.*
3. *For every node in the tree, its immediate successors are all successors of this node under $\mathfrak{R}$ (up to variable renaming).*

As in the case of SLD-trees, one can characterize the finite failure set in terms of SLDB-trees:

**Lemma 6.1** *For an atom $A$, $A \in \mathfrak{FF}(\mathcal{P})$ iff $A$ is not provable by SLDB-resolution and there is a computation rule $\mathfrak{R}$ such that the SLDB-tree for $G, \mathcal{P}$ under $\mathfrak{R}$ is finite.*

*Proof.*

1. Assume that there is an infinite SLDB-tree for a finitely failed goal $G$. Since the number of brances is finite for every node, we have that the SLDB-tree is finite, by König's lemma.
2. In the reverse direction trivial, since an SLDB-tree represents all possible computations under a computation rule.

■

Now we define the notion of an AND-tree for a given totally restricted program $\mathcal{P}$ and a given ground formula $\varphi$. AND-trees are orthogonal to SLDB-trees, which are essentially OR-trees.

**Definition 6.5 (AND-tree)** *An <u>AND-tree</u> for a given totally restricted program $\mathcal{P}$ and a given ground formula $\varphi$ is defined as follows:*

1. *If $\varphi$ has no successors, or $\varphi$ takes the form $\mathtt{t}\neq\mathtt{t}$ then the AND-tree for $\varphi$ consists of one leaf labeled by $\varphi$. In this case we say that this leaf <u>fails</u>.*
2. *If $\varphi$ has the empty successor or takes the form $\mathtt{t}_1\neq\mathtt{t}_2$ for unequal terms $\mathtt{t}_1,\mathtt{t}_2$, then the AND-tree for $\varphi$ consist of one leaf labeled by $\varphi$. In this case we say that this leaf <u>succeeds</u>.*
3. *Assume that there is a successor of the goal $\varphi$, consisting of formulas $\varphi_1, \ldots, \varphi_n$. An AND-tree for $\varphi$ consists of the root labeled by $\varphi$ and having AND-trees for all $\varphi_i$ as its immediate subtrees.*

Using this definition we can give another characterization of the success set and the finite failure set:

**Lemma 6.2** *Let $\varphi$ be a ground formula and $\mathcal{P}$ a totally restricted program. Then*

1. *$\varphi$ is provable iff there is an AND-tree for $\varphi$, whose all leaves succeed.*
2. *$\varphi$ finitely fails iff there is a natural number $n$, such that every AND-tree for $\varphi$ has a failed leaf on depth $\leq n$.*

*Proof.*

1. The first statement can be proved by a simple modification of Theorem 6 on completeness of $\mathbf{Nat}(\mathcal{P})$: it is easy to see that an AND-tree for a goal $G$ represents a proof of $G$ in $\mathbf{Nat}(\mathcal{P})$ (except for the case of $\neq$, which can be easily handled, too).
2. To prove the second statement, we shall introduce a definition and prove a few intermediate statements.

   An <u>$n$-thread</u> in an AND-tree is a set of nodes in this tree, defined inductively as follows.

   (a) If $G$ is the root of the tree, then $\{G\}$ is a 0-thread.

   (b) Let $G, \varphi$ be an $n$-thread and $G_1$ be the immediate successors of $\varphi$ in the tree. Then $G, G_1$ is an $n+1$-thread.

   (c) Let $G, \varphi$ be an $n$-thread, $\varphi$ either takes the form `t=t` or takes the form `r≠s` for not-identical `r,s`. Then $G$ is an $n+1$-thread.

   Using definitions of trees and threads, one can easily prove the following:

   A set of formulas $\varphi_1, \ldots, \varphi_n$ is an $n$-thread iff it is a goal, which can be obtained from the top goal $\varphi$ by SLDB-resolution under some computation rule $\mathfrak{R}$ in $n$ steps.

   The next statement to prove is

   For any computation rule $\mathfrak{R}$, for any AND-tree $T$ for the goal $\varphi$ and for any natural number $n$ either

   (a) there exist a goal $G$ in the SLDB-tree for $\varphi$ under $\mathfrak{R}$ on depth $n$, which is also an $n$-thread in $T$

   or

   (b) there is a leaf $G$ in the SLDB-tree for $\varphi$ on depth $m \leq n$, which is also an $m$-thread in $T$.

   This statement is proved by induction on $n$. For $n = 0$ it is trivial. For $n > 0$, consider the $n-1$-thread, satisfying the condition. Let it be the goal $\varphi_1, \ldots, \varphi_n$. Let $\mathfrak{R}(\varphi_1, \ldots, \varphi_n) = \varphi_i$. If $\varphi_i$ is a failed leaf in $T$, then the second statement is trivially satisfied. If not, then let $\psi_1, \ldots, \psi_k$ be all the immediate successors of $\varphi_i$ in $T$. Then $\varphi_1, \ldots, \varphi_{i-1}, \psi_1, \ldots, \psi_k, \varphi_{i+1}, \ldots, \varphi_n$ can be taken as the $G$.

   Now we can prove one part of the statement. Let $\varphi$ finitely fail. By Lemma 6.1 there is a finite SLDB-tree $S$ for $\varphi$ under a computation rule $\mathfrak{R}$, whose all leaves fail. Take as $n$ the depth of $S$. Consider any AND-tree $T$. Let $G$ be the goal which is both a leaf in $S$ and an $m$-thread in $T$. Let $\psi$ be the formula chosen by $\mathfrak{R}$ in $G$. Then $\psi$ is a failed leaf in $T$ on depth $\leq n$.

   The converse will be proved by induction on $n$. Let any AND-tree for $\varphi$ has a failed leaf on depth $\leq n$. We have to find a computation rule $\mathfrak{R}$ which gives a finite SLDB-tree for $G$. Let $G_1, \ldots, G_n$ be all the possible successors of $\varphi$ under SLDB-resolution. Consider any $G_i$. Let it be $\varphi_1, \ldots, \varphi_n$. If each of the $\varphi_j$ has an AND-tree of depth $\geq n$ with no failing leaves, then there is an AND-tree for $\varphi$ of depth $\geq n+1$ with no failing leaves, which contradicts the assumption. Thus there is a $\varphi_{j_i}$, for which all AND-trees has a failing leaf on depth $< n$. Let $\mathfrak{R}_i$ be computation rules which lead to finite SLDB-trees for $\varphi_{j_i}$. Consider the computation rule $\mathfrak{R}$, which chooses $\varphi_{j_i}$ from $G_i$ and behaves as $\mathfrak{R}_i$ on all the successors of $G_i$. It is easy to see that $\mathfrak{R}$ gives a finite SLDB-tree for $\varphi$.

■

We shall use AND-trees as the main technical tool in the rest of this section.

**Definition 6.6 (Dual formula, clause and program)** *Let $\mathcal{P}$ be a totally restricted program which defines predicates $P_1, \ldots, P_n$. Let $\widehat{P}_1, \ldots, \widehat{P}_n$ be some new predicate symbols. For each formula $\varphi$ the dual formula $\widehat{\varphi}$ 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 occurrences of $\exists$ by $\forall$, and all occurrences of $\forall$ by $\exists$. For a clause $C$ of the form $R(\bar{x})$:-$\varphi(\bar{x})$ the dual clause $\widehat{C}$ is $\widehat{R(\bar{x})}$:-$\widehat{\varphi(\bar{x})}$. The dual program $\widehat{\mathcal{P}}$ to $\mathcal{P}$ consist of the clauses dual to the clauses of $\mathcal{P}$.*

For example, the dual clause to

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

is

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

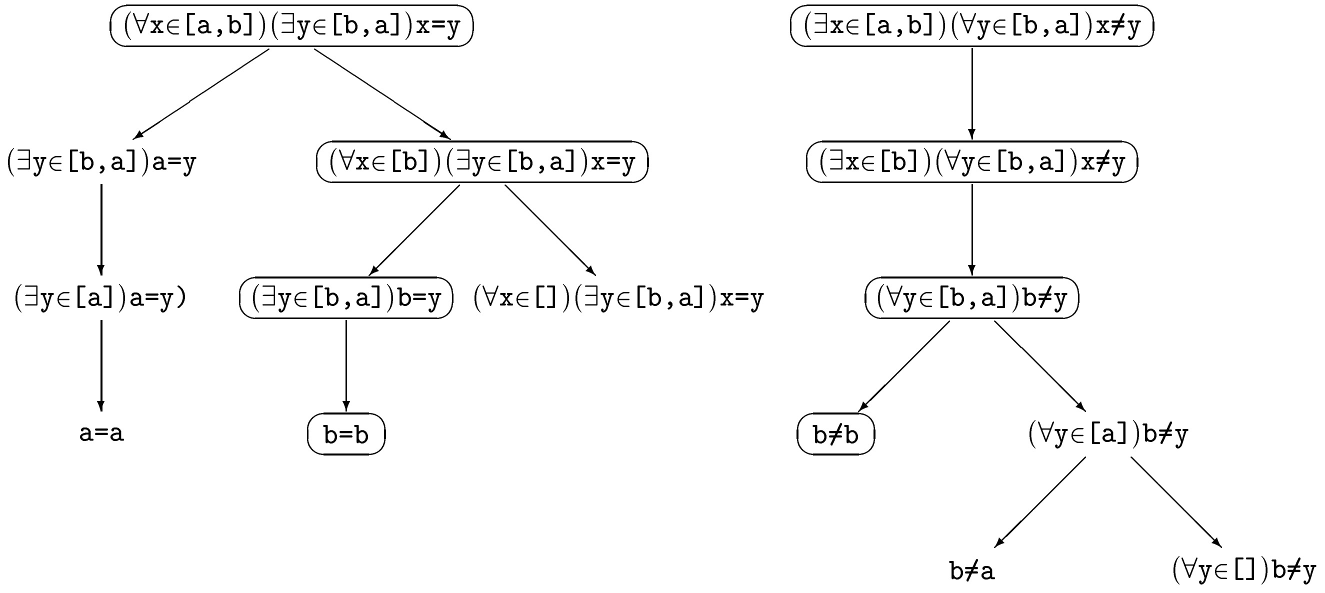
where we denoted the dual predicate symbols with the prefix `not_`. The dual program $\widehat{\mathcal{P}}$ to $\mathcal{P}$ consists of all such dual clauses together with a definition of $\neq$.

**Theorem 19** *For any predicate $P$ defined in the program $\mathcal{P}$ the predicate $\widehat{P}$ from the program $\widehat{\mathcal{P}}$ computes the finite failure set $\mathfrak{F}\mathfrak{F}_{\mathcal{P}}(P)$ for $P$ in $\mathcal{P}$.*

*Proof.* Let us introduce a few technical definitions which we use in the proof. A *path* in a tree is defined as usual. A *branch* is a path which cannot be extended to a longer path. We call a path $B_1$ in an AND-tree $T_1$ *dual* to a path $B_2$ in an AND-tree $T_2$ iff $B_1$ consists of formulas dual to the formulas from $B_2$. We prove the following statement by induction on the length of branches in SLDB-trees:

Let $T_1, T_2$ be AND-trees for the formulas $\varphi, \widehat{\varphi}$. Then there exist branches $B_1$ in $T_1$ and $B_2$ in $T_2$ such that $B_1$ is dual to $B_2$.

For example, the dual branches for two trees for formulas $(\forall$`x`$\in$`[a,b]`$)(\exists$`y`$\in$`[b,a]`$)$`x=y` and $(\exists$`x`$\in$`[a,b]`$)(\forall$`y`$\in$`[b,a]`$)$`x≠y` are shown on the following picture:




To prove this statement we consider several cases. We assume that $\varphi$ is the bottom formula of the path $B_1$, the path $B_2$ is dual to $B_1$ and prove that either $\varphi$ is a leaf or $B_1, B_2$ can be continued to longer dual paths.

1. $\varphi$ is an atomic formula $P(\bar{t})$, where $P$ is neither = nor ≠. Then $\mathcal{P}$ contains a clause $P(\bar{x})$:-$\psi(\bar{x})$ and the branch $B_1$ can be continued by $\psi(\bar{t})$. In this case the dual program $\widehat{\mathcal{P}}$ contains the dual clause $\widehat{P}(\bar{x})$:-$\widehat{\psi(\bar{x})}$, and $B_2$ can be continued by $\widehat{\psi(\bar{x})}$.
2. $\varphi$ takes the form $\varphi_1 \wedge \ldots \wedge \varphi_n$. In this case the dual formula to $\varphi$ is $\widehat{\varphi_1} \vee \ldots \widehat{\varphi_n}$. Thus any path continuing $B_2$ has one of the formulas $\widehat{\varphi_i}$ as the immediate successor of $\widehat{\varphi}$. Thus we can continue $B_1$ by $\varphi_i$.
3. $\varphi$ takes the form $(\exists$x∈[s|t]$)\psi($x$)$. In this case $B_2$ finishes at $(\forall$x∈[s|t]$)\widehat{\psi}($x$)$. In this case $B_1$ is continued either by $\psi($s$)$ or by $(\exists$x∈[t]$)\psi($x$)$. In the former case continue $B_2$ by $\widehat{\psi}($s$)$, in the latter by $(\forall$x∈[t]$)\widehat{\psi}($x$)$.

The other cases are similar. Now we can prove one part of the theorem:

> Let the formula $\varphi$ be provable in $\mathcal{P}$. Then $\widehat{\varphi}$ finitely fails in $\widehat{\mathcal{P}}$.

To prove this we use the characterization of provability and finite failure in terms of AND-trees from Lemma 6.2. Since $\varphi$ is provable, than there is an AND-tree $T_1$, whose all branches finish at leaves, which succeeds. Let $n$ be the depth of this tree (i.e. the length of a longest branch in $T_1$). Take any AND-tree $T_2$ for $\widehat{\varphi}$. Then there are branches $B_1, B_2$ in $T_1, T_2$, which are dual. Since $B_1$ finishes at a succeeding leaf, then $B_2$ finishes at a failing leaf. Since the length of $B_2$ is the same as the length of $B_1$, which is $\leq n$, then $\widehat{\varphi}$ finitely fails.

To prove the converse, it is enough to prove the following statement:

> if every AND-tree for a formula $\widehat{\varphi}$ has a failed leaf on depth $\leq n$, then $\varphi$ has an AND-tree of depth $\leq n$, whose all leaves succeed.

The proof is by induction on $n$ and case analysis.

1. Assume that $\varphi$ takes the form $\varphi_1 \wedge \ldots \wedge \varphi_n$. Then $\widehat{\varphi}$ takes the form $\widehat{\varphi_1} \vee \ldots \vee \widehat{\varphi_n}$. Then all AND-trees for $\widehat{\varphi}$ have one of the formulas $\widehat{\varphi_i}$ as the immediate successor of $\widehat{\varphi}$. All these trees finitely fail on depth $\leq n-1$. Thus for every formula $\varphi_i$ one can construct a successful AND-tree of depth $\leq n-1$. Combining these trees, we obtain the required AND-tree for $\varphi$.

2. Assume that $\varphi$ takes the form $(\exists x\in[s|t])\psi(x)$. Then $\widehat{\varphi}$ takes the form $(\forall x\in[s|t])\widehat{\psi}(x)$. Then all AND-trees for $\widehat{\varphi}$ have the formulas $\widehat{\psi}(s)$ and $(\forall x\in t)\widehat{\psi}(x)$ as the immediate successors of $\widehat{\varphi}$. All this trees finitely fail on depth $\leq n-1$. From this one can prove that either all AND-trees for $\widehat{\psi}(s)$ finitely fail on depth $\leq n-1$ or all SLDB-trees for $(\forall x\in t)\widehat{\psi}(x)$ finitely fail on depth $\leq n-1$. Consider, for instance, the former case. Then $\psi(s)$ has a successful AND-tree of the depth $\leq n-1$, and thus $\varphi$ has the required AND-tree.

The other cases are similar. As one can see from the proof, the required AND-tree for $\varphi$ can be constructed from the failing branches of the trees for dual formulas, as illustrated below for the formulas from the previous example. In this case all AND-trees take one of the following forms:

$$\begin{tikzcd}
 & & (\exists x\in[a,b])(\forall y\in[b,a])x\neq y \arrow[d] & \\
 & & (\forall y\in[b,a])a\neq y \arrow[dl] \arrow[dr] & \\
 & (\forall y\in[a])a\neq y \arrow[dl] \arrow[dr] & & \cdots \\
\cdots & & a\neq a &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (\exists x\in[a,b])(\forall y\in[b,a])x\neq y \arrow[d] & \\
 & (\exists x\in[b])(\forall y\in[b,a])x\neq y \arrow[d] & \\
 & (\forall y\in[b,a])b\neq y \arrow[dl] \arrow[dr] & \\
\cdots & & b\neq b
\end{tikzcd}$$

$$\begin{tikzcd}
(\exists x\in[a,b])(\forall y\in[b,a])x\neq y \arrow[d] \\
(\exists x\in[b])(\forall y\in[b,a])x\neq y \arrow[d] \\
(\exists x\in[\,])(\forall y\in[b,a])x\neq y
\end{tikzcd}$$

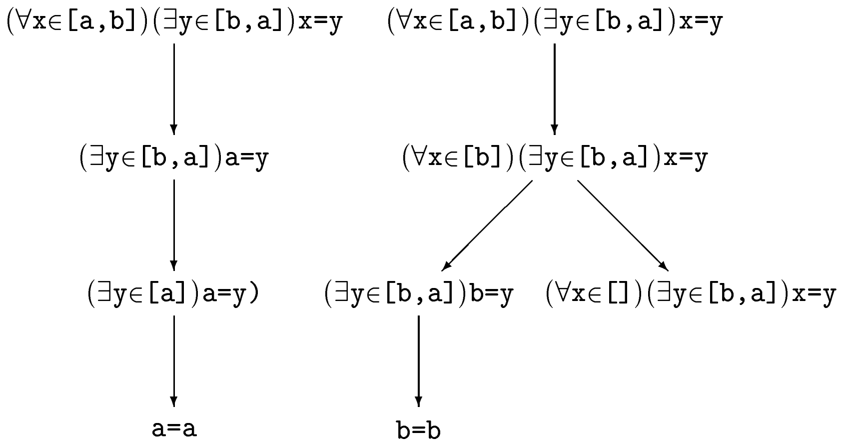
The branches dual to the failed branches are the following:

$$\begin{tikzcd}
(\forall x\in[a,b])(\exists y\in[b,a])x=y \arrow[d] \\
(\exists y\in[b,a])a=y \arrow[d] \\
(\exists y\in[a])a=y \arrow[d] \\
a=a
\end{tikzcd}$$

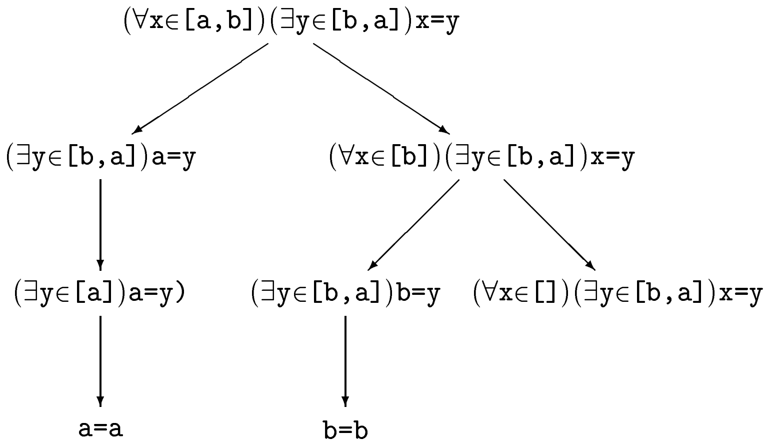
$$\begin{tikzcd}
(\forall x\in[a,b])(\exists y\in[b,a])x=y \arrow[d] \\
(\forall x\in[b])(\exists y\in[b,a])x=y \arrow[d] \\
(\exists y\in[b,a])b=y \arrow[d] \\
b=b
\end{tikzcd}$$

$$\begin{tikzcd}
(\forall x\in[a,b])(\exists y\in[b,a])x=y \arrow[d] \\
(\forall x\in[b])(\exists y\in[b,a])x=y \arrow[d] \\
(\forall x\in[\,])(\exists y\in[b,a])x=y
\end{tikzcd}$$

One can combine these branches into the successful AND-tree step by step as follows:




and




■

This theorem is applicable to the class of all totally restricted programs which is equivalent to the class of all generalized logic programs, as has been shown in Theorem 13.

## 7 Concluding remarks

There are other aspects of 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 an iterative search over finite domains. There are domains 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(x̄,l,[]) :- φ(x̄,l).
D(x̄,l,[y|z]) :- D(x̄,l,z), D(x̄,[y|l],z).
```

Bounded quantification over an integer interval `[k..n]` consisting of 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 in the style of [Voronkov 92a].

### 7.2 Constraint logic programming

The above considerations suggest that different kinds of bounded quantifiers may be used to specify constraint problems. Somewhat similar constructions have been used in the language cc(FD) for constraint logic programming over finite domains [Hentenryck 91]. Constraints using bounded quantifiers can be used 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 [Bürkert 90]. We are going to illustrate this in a future paper.

The following example specifies a generalized logic program for the N-queens problem:

**Example 7.1 (N-queens problem)** *The following example defines N-queens problem.*

```
queens(L,N) :-
   length(L,N) ∧
   admissible(L,N) ∧
   (∀l⊑L)(l=[]∨safelist(l)).

length( [], 0).
length( [_|As], N) :-
   length( As, N1),
   N is N + 1.

safelist([P|Ps]) :-
   (∀p∈Ps)safe(P,p).

safe((A,B),(C,D)) :-
   A \= C ∧
   B \= D ∧
   A+B \= C+D ∧
   A-B \= C-D.

admissible(L,N):-
   (∀x∈[1..N])(∃y∈[1..N])(∃z∈L)(z=(x,y)).
```

An interesting feature of this example is that one can modify SLDB-resolution rules for handling the bounded quantifiers to implement different strategies of solving the problem. Moreover, one can build an intelligent problem solver for the specifications with bounded quantifiers, which will encompass some known strategies of problem solving for finite domains, for example the first fail principle, which will be shown in a forthcoming paper.

### 7.3 Other applications

There are many other applications of bounded quantifiers. An obvious application of bounded quantifiers is parallel and concurrent logic programming, as was also noted in [Barklund 92]. 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 FORALL-parallelism from [Kowalski 83]. As for bounded existential quantifiers, their procedural interpretation is completely different from the interpretation of unrestricted existential quantifiers, which usually serves only for unification purposes. 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. If we allow bounded quantification and disjunctions also in the heads of clauses, then formulas containing bounded quantifiers can be used e.g. to express null values in disjunctive databases [Liu 90, Lobo 92]. For example, the query

```
?-(∃y∈x)p(y)
```

where `x` is a variable, corresponds to the query "find an indefinite (or null) value for `y` for which `p(y)` is true", which is not expressible via range-restricted clauses in deductive databases. In a forthcoming paper we intend to show how bounded quantifiers can be used to specify an indefinite information in deductive databases, which cannot be expressed in the ordinary disjunctive databases. An example of such a specification is

```
(∃x∈G)group_leader(x,G) :- group(G).
```

which express the information that each group has a group leader. Such specifications are usually processed as integrity constraints. A special proof procedure will allow to consider them also as clauses.

It has been shown elsewhere that varying the structure of bounded quantifiers, one can represent predicates or functions from different complexity classes [Buss 86, Sazonov 91].

## 8 Acknowledgments

Francois Bry and Mark Wallace made many helpful comments on a preliminary version of this paper. I also thank Michael Ratcliffe and Mike Reeve for their help in proofreading.

# Bibliography


[Ait Kaci 90] H. Ait Kaci. The WAM, a (real) tutorial. Technical Report 5, Digital Paris Research Laboratory, January 1990.

[Andréka 78] Andréka and Németi. A generalized completeness of Horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[Apt 82] K. Apt and M. van Emden. Contributions to the theory of logic programming. *Jornal of the Association for Computing Machinery*, 29(3), 1982.

[Barklund 92] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 817–824, Tokyo, 1992. ICOT.

[Barwise 75] J. Barwise. *Admissible Sets and Structures.* Springer Verlag, 1975.

[Beeri 87] 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.

[Bürkert 90] 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.

[Buss 86] S.R. Buss. *Bounded Arithmetic*, volume 3 of *Studies in Proof Theory.* Bibliopolis, Napoly, 1986.

[Chang 77] C.C. Chang and H.J. Keisler. *Model theory.* North Holland, 1977.

[Clark 78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Base*, pages 293–322. Plenum Press, New York, 1978.

[Dovier 91] 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.

[Eder 85] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computations*, 1(1):31–48, 1985.

[Falashi 89] M. Falashi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):298–318, 1989.

[Goncharov 85] S.S. Goncharov and D.I. Sviridenko. *$\Sigma$-programming (in Russian)*, volume 120 of *Vychislitelnye Systemy*, pages 3–29. Novosibirsk, 1985.

[Goncharov 86a] S.S. Goncharov and D.I. Sviridenko. Theoretical aspects of $\sigma$-programming. In *Mathematical Methods of Specification and Synthesis of Software Systems'85*, volume 215 of *Lecture Notes in Computer Science*, pages 169–179, 1986.

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

[Hagiya 84] M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2(1):59–77, 1984.

[Harrop 60] R. Harrop. Concerning formulas of the types $A \to B \land C$, $A \to (Ex)B(x)$ in intuitionistic formal system. *J. of Symb. Logic*, 17:27–32, 1960.

[Hentenryck 91] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(FD). Technical report, Brown University, December 1991.

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

[Kleene 62] S.C. Kleene. Disjunction and existence under implication in elementary intuitionistic formalism. *Journal of Symbolic Logic*, 27(1), 1962.

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

[Kuper 87] 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.

[Kuper 88] 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.

[Liu 90] Yuan Liu. Null values in definite programs. In S. Debray and M. Hermenegildo, editors, *Proc. of the 1990 North American Conference on Logic Programming*, pages 273–288. MIT Press, 1990.

[Lloyd 84] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[Lobo 92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* Logic Programming Series. MIT Press, 1992.

[Moor 81] D. Moor and B. Russel. Axiomatic data type specifications: a first order theory of linear lists. *Acta Informatica*, 15(3):193–208, 1981.

[Naish 86] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science.* Springer Verlag, 1986.

[Nicolas 83] J.-M. Nicolas and R. Demolombe. On the stability of relational queries. Technical report, ONERA-SERT, Toulouse, 1983.

[PDC 90] M. Alexander, P. Bilse, L. Jensen e.a. *PDC Prolog User's Guide.* Prolog Development Center, 1990.

[PE 91] *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM SIGPLAN Notices, v.26, no.9, September 1991.

[Rogers 67] H. Rogers. *Theory of recursive functions and effective computability.* McGraw Hill, 1967.

[Sato 84] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proc. of the Conference on Fifth Generation Computer Systems*, pages 195–201. ICOT, 1984.

[Sazonov 91] V.Yu. Sazonov. Hereditarily finite sets with attributes, databases and polynomial time computability. In *Actes Preliminaires du Symposium Franco-Sovetique INFORMATIKA-91*, pages 113–132. INRIA, 1991.

[Schwartz 86] J.T. Schwartz, R.B.K. Devar, E. Dubinski, and E. Schonberg. *Programming with sets: an Introduction to SETL*. Springer Verlag, 1986.

[Sheperdson 87] J.C. Sheperdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1987.

[Tärnlund 77] S.-Å. Tärnlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[Topor 87] R.W. Topor. Domain independent formulas and databases. *Theoretical Computer Science*, 52(3):281–306, 1987.

[Tsur 86] 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.

[Turner 86] D.A. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[Voronkov 86a] A. Voronkov. Program execution methods in $\Sigma$-programming (in Russian). In *Proc. 4th Soviet Conf. on Applications of Mathematical Logic*, pages 51–53, Tallinn, 1986.

[Voronkov 86b] A. Voronkov. Intuitionistic list theory (in Russian). In *Proc. 8th Soviet Conf. on Mathematical Logic*, page 32, Moscow, 1986.

[Voronkov 86c] A. Voronkov. Logic programs and their synthesis (in Russian). Technical Report 23, Institute of Mathematics, Novosibirsk, 1986.

[Voronkov 86d] A. Voronkov. Synthesis of logic programs (in Russian). Technical Report 24, Institute of Mathematics, Novosibirsk, 1986.

[Voronkov 87] A. Voronkov. *A natural calculus for $\Sigma$-programs (in Russian)*, volume 120 of *Vychislitelnye Sistemy*, pages 14–23. Novosibirsk, 1987.

[Voronkov 89] A. Voronkov. Logic programming and $\Sigma$-programming (in Russian). *Kibernetika*, (1):67–72, 1989.

[Voronkov 91] A. Voronkov. N-realizability: one more constructive semantics. Technical Report 71, Monash University, Department of Computer Science, Clayton, Australia, 1991.

[Voronkov 92a] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 486–514. Springer Verlag, 1992.

[Voronkov 92b] A. Voronkov. On computability by logic programs. Technical Report ECRC-92-8, European Computer Industry Research Center, 1992.

[Walther 90] C. Walter. Many-sorted inferences in theorem proving. In K.H. Blasius, U. Hedstuck, and C.R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *Lecture Notes in Artificial Intelligence*, pages 18–48, 1990.

[Wiggins 91] G.A. Wiggins, A. Bundy, H.C. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K.K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, Workshops in Computing. Springer Verlag, 1990.