In this paper we propose a concept of type and type sorts for logic programming and we discuss computing with types and type sorts. Instead of defining "well-sorted" terms and substitutions, type constraints and type sort constraints are defined. We assume that the universe of discourse can be represented by ground data terms. Data terms (also called constructor terms) are generated by contructor symbols and variables. Constructor symbols are function symbols which are interpreted by themselves and do not have any operational semantics. Ground data terms are data terms without any variables. A data term with variables can be regarded as a representation of a specific subset of the universe of discourse. However, generally, not every ground data term can be assigned to an element of the universe of discourse. The construction of data terms can be restricted if typed variables are used instead of general variables, where a type characterizes a specific set of data terms. For the representation of types data terms are used. Thus we have to distinguish between two kinds of constructor symbols: constructor symbols for the representation of the basic objects of the universe and constructor symbols for the representation of types. In the sequel we use the names "*object constructor*" and "*type constructor*" for these kinds of constructor symbols. Furthermore, data terms generated only by object constructors and variables are called object terms, and data terms for the representation of types are called type terms.

Types can be defined by type rules of the form $s :\Rightarrow t \Leftarrow C$, where s is a type term, t is an object term and C is a set of type constraints. For example, the type of natural numbers can be defined by the following type rules: $nat :\Rightarrow 0$ and $nat :\Rightarrow suc(x) \Leftarrow x{:}nat$. This syntax emphasizes the concept that types are specified by means of the elements belonging to these types. Thus type rules can also be regarded as rules for the generation of the elements belonging to the corresponding type. Note that types are interpreted as subsets of the universe.

We introduce type sorts so that the set of types itself can be structured. A type sort is interpreted as a set of types. For example, type sorts may be introduced for types representing subsets of numbers, for types representing sets of lists, and for types representing specific object terms. A type sort does not contain any parameter, but a subset relation can exist between some type sorts. The introduction of a type sort for a set of types has more expressive power than a type representing the union of this set. There are better possibilities of structuring the universe by the introduction of type sorts.

We show that the definitions of types and type sorts can be transformed into definite clauses. Therefore, the semantics of the types and type sorts can be based on the least Herbrand model of the constraint theory consisting of these definite clauses. An important operation with types is the generation of the intersection of two types. If computing with types is regarded as constraint processing, then it is sufficient to generate an element which belongs to the intersection (or to prove that such an element does not exist).

Type constraint processing is based on SLD-resolution with a specific computation rule. Type checking, i.e. checking if a type constraint without any variable is satisfiable, is shown to be decidable. Provided that the specifications of types and type sorts satisfy certain conditions, some interesting properties of type constraint

processing are discussed. Furthermore, a strategy for type constraint processing is proposed.

The type system discussed in this paper differs mainly from other approaches of type systems by the following points:

- Types are strictly regarded as constraints and computation with types is regarded as contraint processing.
- Besides type variables, object terms and object variables can be parameters of types.
- The set of types is structured by type sorts (note that the introduction of type sorts is mentioned in [11]).

Usually, the definitions of types have to satisfy certain conditions in order to guarantee good properties of computation. In the presented approach these conditions are syntactical conditions which can be easily checked.

We assume the reader to be familiar with the usual notations and basic results of the foundations of logic programming and constraint logic programming (see e.g. [17], [13, 14]).

2 Basic Definitions and Notations

The set of function symbols $\mathbb{F}$ is subdivided into symbols for object constructors $\mathbb{F}_{obj}$, symbols for type constructors $\mathbb{F}_{type}$, and 0-ary symbols for type sorts $\mathbb{F}_{sort}$. These subsets are supposed to be pairwise disjoint. We use the notation f/n if we want to express that a function symbol f is associated with the arity n. In the following we suppose that $\mathbb{F}$ is given and finite, $types \in \mathbb{F}_{sort}$, and that both $\mathbb{F}_{obj}$ and $\mathbb{F}_{type}$ contain at least one function symbol of arity 0. Two sorts of variables are introduced: a denumerable set $\mathbb{V}_{obj} = \{x, y, z, x_1, y_1, z_1, x_2, \ldots\}$ of object variables and a denumerable set $\mathbb{V}_{type} = \{\alpha, \beta, \alpha_1, \beta_1, \alpha_2, \ldots\}$ of type variables (throughout this paper we use these notations for variables). The union of these sets of variables is denoted by $\mathbb{V}$ and we use the notation of an object variable, if we want to mark a variable of any sort. If Ex is a syntactical expression, the set of (object and type) variables which are contained in it is denoted by $var(Ex)$ and the set of type variables by $var_{type}(Ex)$.

If a set $V \subseteq \mathbb{V}$ of variables and a set $F \subseteq \mathbb{F}$ of function symbols are given, the set of first order terms $Tm(V, F)$ is defined as usual. Elements of $Tm(\mathbb{V}_{obj}, \mathbb{F}_{obj})$ are called *object terms* (data terms) and denoted by $\mathcal{O}$. The set $\mathcal{T}$ of *type terms* is the set of terms $h(t_1, \ldots, t_n)$ satisfying the following conditions: $h/n \in \mathbb{F}_{type}$ and $t_1, \ldots, t_n \in \mathbb{V}_{type} \cup \mathcal{T} \cup \mathcal{O}$. We say a term is ground if there is not any variable in it. The set of ground object terms is denoted by $\mathcal{O}_0$ and the set of ground type terms by $\mathcal{T}_0$. A *type constraint* is a syntactical expression of the form $t{:}s$, where one of the following conditions must be satisfied: $t \in \mathcal{O}$ and $s \in \mathbb{V}_{type} \cup \mathcal{T}$ or $t \in \mathbb{V}_{type} \cup \mathcal{T}$ and $s \in \mathbb{F}_{sort}$. A *variable constraint* is a type constraint $x{:}s$ with $x \in \mathbb{V}_{obj} \cup \mathbb{V}_{type}$ and $x \notin var(s)$. We use a tilde to denote a sequence of terms. For example, $\tilde{t}$ denotes the terms $t_1, \ldots, t_n$. $\exists\varphi$ denotes the exential closure of a formula φ.

A *substitution* σ is a mapping from $\mathbb{V}$ to the set $Tm(\mathbb{V}, \mathbb{F})$ such that the set

$\{x \in \mathbb{V} \mid \sigma(x) \neq x\}$ is finite, $x \in \mathbb{V}_{obj}$ implies $\sigma(x) \in \mathcal{O}$, and $\alpha \in \mathbb{V}_{type}$ implies $\sigma(\alpha) \in \mathcal{T}$. We use the notation $\sigma(t)$ to represent the term obtained by replacing the variables of t by their images under σ. A substitution σ is determined by a set $\{x_1/t_1, \ldots, x_n/t_n\}$. The composition $\sigma o \tau$ of two substitutions σ and τ is defined by $\sigma o \tau(x) = \tau(\sigma(x))$. A *unifier* of an equation $t = u$ is a substitution σ such that $\sigma(t)$ and $\sigma(u)$ are syntactically identical. A unifier σ of an equation $t = u$ is called a most general unifier of this equation, if for each unifier ϑ there is a substitution τ such that $\vartheta = \sigma o \tau$. A unifier of a set of equations $\{t_1 = u_1, \ldots, t_n = u_n\}$ is a substitution σ such that σ is a unifier of each equation occuring in this set. Note that a unifier takes into account the two sorts of variables.

3 Type Sorts and Types

We introduce type sorts so that the set of types itself can be structured. A type sort does not contain any parameter and is interpreted as a set of types. A subset relation can exist between some type sorts. We assume that there is a type sort *types* which is interpreted as the set of all types. The introduction of a type sort for a set of types has more expressive power than a type representing the union of this set. There are better possibilities of structuring the universe by the introduction of type sorts.

Definition 1 (specification of type sorts) *A specification of type sorts consists of a set* $\mathbb{F}^b_{sort} \subseteq \mathbb{F}_{sort}$ *and a set of definitions* $a := b_1 \; ++ \; \ldots \; ++ \; b_k$ *with* $a \in \mathbb{F}_{sort} \setminus (\mathbb{F}^b_{sort} \cup \{types\})$ *and* $b_1, \ldots, b_k \in \mathbb{F}_{sort}$. *If* $\mathbb{F}_{sort} \setminus \{types\} \neq \emptyset$, *then* $types \notin \mathbb{F}^b_{sort}$. *The elements of* $\mathbb{F}^b_{sort}$ *are called basic type sorts.*

Example 1 A specification of type sorts is:
$\mathbb{F}^b_{sort} = \{number\_types, list\_types, vehicle\_types, person\_types, country\_types\}$,
$basic\_object\_types := vehicle\_types \; ++ \; person\_types \; ++ \; country\_types$. ◁

A definition of a type consists of a declaration of this type and a set of type rules (the definition rules). The declaration of a type determines the types and type sorts, respectively, of the arguments and specifies the type sort to which the type belongs. Type rules can be regarded as rules for the generation of the elements belonging to the corresponding type.

Definition 2 (declaration of a type) *A declaration of a type* $h/n \in \mathbb{F}_{type}$ *is an expression* **type** $h: s_1 \times \ldots \times s_n \rightarrow a$, *where* $s_1, \ldots, s_n \in \mathbb{F}_{sort} \cup \mathcal{T}_0$ *and* $a \in \mathbb{F}_{sort}$.

Definition 3 (type rule) *A type rule of a type is an expression* $s :=> u \leftarrow C$ *such that the following conditions are satisfied (*x *denotes any variable):*

(1) $s = h(t_1, \ldots, t_n)$ *with* $h \in \mathbb{F}_{type}$, $t_1, \ldots, t_n \in \mathcal{O} \cup \mathcal{T}_0 \cup \mathbb{V}_{type}$, *and* $u \in \mathcal{O}$;

(2) C *is a set of variable constraints with* $var(C) \subseteq var(u) \cup var(s)$;

(3) if $x:r \in C$, *then* $x \in var(u) \setminus var(s)$, $var(r) \subseteq var(s)$, *and there is not any type variable which occurs more than once in* r *(if* $var(u) \subseteq var(s)$, *then* $C = \emptyset$*);*

(4) if $u \in \mathbb{V}$ *and* $u:r \in C$, *then* $r \notin \mathbb{V}_{type}$, *and the type constructor* h *does not occur in* r;

(5) if $x:r \in C$ *is a variable constraint which contains the type constructor* h, *then* $r = h(r_1, \ldots, r_k)$, h *does not occur in any argument* r_i *of* r, *and for each* $i \leq k$ *holds* $r_i \in \mathbb{V}_{type}$ *or* $var_{type}(r_i) = \emptyset$.

Example 2 The definitions of the type of finite natural numbers $\{0, suc(0), \ldots, x\}$ and the type of lists with a certain number of elements are given (the definition of $nat/0$ is given in Section 1).

type $f\_nat: nat \rightarrow number\_types$
$f\_nat(x) :=> x$
$f\_nat(suc(x)) :=> y \leftarrow y:f\_nat(x)$

type $list: nat \times types \rightarrow list\_types$
$list(0, \alpha) :=> []$
$list(suc(x), \alpha) :=> [y \mid z] \leftarrow y:\alpha, \; z:list(x, \alpha)$ ◁

Let $h(t_1, \ldots, t_n) :=> u \leftarrow C$ be a type rule. If the argument t_i of $h(t_1, \ldots, t_n)$ is a type parameter, then t_i is a type variable or a ground type term (condition (1) of Definition 3). The conditional part C of this type rule cannot introduce any new variable (condition (2)). The conditional part C contains variable constraints only for such variables which belong to u and do not belong to $h(t_1, \ldots, t_n)$ (condition (3)). Condition (5) of Definition 3 restricts the possibilities of recursive definitions. These restrictions of the definitions of types and the assumption that type sorts and types are hierachically defined guarantee good properties of computation. Especially, these conditions are needed in the proof of Theorem 1.

Note that a type rule $s :=> x \leftarrow x:r$ states that the interpretation of r is a subset of the interpretation of s. Furthermore, a type rule of the form $s :=> x \leftarrow x:s_1, x:s_2$ states that the interpretation of s is the intersection of the interpretations of r_1 and r_2.

A type can also be defined by means of a set definition ($s := \{t_1, \ldots, t_m\}$, where $s \in \mathbb{F}_{type}$ and $t_1, \ldots, t_m$ are 0-ary function symbols of $\mathbb{F}_{obj}$) or by means of a union of types ($s := s_1 \; ++ \; \ldots ++ \; s_m$ with $s_1, \ldots, s_m \in \mathcal{T}$ and $var(\{s_1, \ldots, s_m\}) \subseteq var(s)$). However, such a definition can be transformed into a set of type rules. A definition of a type by means of a set definition $s := \{u_1, \ldots, u_m\}$ can be transformed into the type rules $s :=> u_1$, ..., $s :=> u_m$. A definition of a type by means of a definition $s := s_1 ++ \ldots ++ s_m$ can be transformed into the type rules $s :=> x \leftarrow x:s_1$, ..., $s :=> x \leftarrow x:s_m$, where x is a new variable (x does not occur in $s, s_1, \ldots, s_m$). Therefore, in the following, we can suppose that all types are defined by means of type rules.

We assume that type sorts and types are hierarchically defined (i.e. a type can be only used in a type definition, if this type is already defined), where a direct recursive definition of types is allowed. This assumption can be formally defined by a function (level mapping) η of $\mathbb{F}$ into the natural numbers such that the following conditions are satisfied:

(C0) If $f \in \mathbb{F}_{obj}$, $h \in \mathbb{F}_{type}$, and $a \in \mathbb{F}_{sort} \setminus \{types\}$, then
$\eta(f) < \eta(h) < \eta(a) < \eta(types)$.

(C1) If $a := b_1 \; ++ \; \ldots \; ++ \; b_k$ is a definition of type sorts, then $\eta(b_i) < \eta(a)$ for each $i \leq k$.

(C2) If **type** $h : s_1 \times \ldots \times s_n \rightarrow a$ is a declaration of some type, then $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_{type}$ occuring in $s_1, \ldots, s_n$.

(C3) If $h(t_1, \ldots, t_n) :=> u \leftarrow \mathcal{C}$ is a type rule, then
- $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_{type}$ contained in $t_1, \ldots, t_n$;
- $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_{type}$ which occurs in $\mathcal{C}$ and $g \neq h$.

Because of technical reasons we introduce further notations. The expression $x:(s_1, \ldots, s_n)$ abbreviates the set $\{ x:s_1, \ldots, x:s_n \}$. We use this notation in the case $n = 1$, too. The notation $x:(s_1, \ldots, s_n) \uplus \mathcal{C}$ means that there is not any r such that $x : r \in \mathcal{C}$. $\mathbb{F}^*_{type}$ denotes the set of finite tuples of elements of $\mathbb{F}_{type}$. Analogously, $\mathcal{T}_0^*$ denotes the set of finite tuples of elements of $\mathcal{T}_0$.

4 Type Clauses

Specifications of type sorts and types can be transformed into definite clauses called *type clauses*.

If $a := b_1 \; ++ \; \ldots \; ++ \; b_k$ is a definition of some type sort, then the following clauses are generated: $\alpha:a \leftarrow \alpha:b_1, \ldots, \alpha:a \leftarrow \alpha:b_k$. Moreover the clause $\alpha:types \leftarrow \alpha:a$ is added for each type sort a which does not occur in the left hand side of some definition of a type sort. If $h(t_1, \ldots, t_n) :=> u \leftarrow y_1:r_1, \ldots, y_k:r_k$ is a type rule of a type definition and **type** $h : s_1 \times \ldots \times s_n \rightarrow a$ is the declaration belonging to this definition, then the following definite clause is generated: $u : h(t_1, \ldots, t_n) \leftarrow y_1:r_1, \ldots, y_k:r_k, t_1:s_1, \ldots, t_n:s_n$. In addition the clause $h(x_1, \ldots, x_n) : a \leftarrow x_1:s_1, \ldots, x_n:s_n$ is generated for the declaration.

Example 3 The following clauses are generated by the definition of *list*/2 [1]:

$list(x, \alpha) : list\_types \leftarrow x:nat, \alpha:types$
$[] : list(0, \alpha) \leftarrow 0:nat \; \alpha:types$
$[y \mid z] : list(s(x), \alpha) \leftarrow y:\alpha, z:list(x, \alpha), s(x):nat, \alpha:types$ ◁

A type clause $u:h(\tilde{t}) :=> u \leftarrow \mathcal{C}$ is said to be recursive, if the type constructor h occurs in $\mathcal{C}$. In the other case this clause is called nonrecursive type clause.

The described transformation can be improved for recursive type clauses such that type checking of the arguments of a recursive type is only carried out in the first call and not in a recursive call. If the definition of $h/n \in \mathbb{F}_{type}$ is recursive and **type** $h : s_1 \times \ldots \times s_n \rightarrow a$ is the corresponding declaration, then $\mathbb{F}_{type}$ is extended by a new symbol h_{rec}/n of a type constructor, the definite clause $x:h(x_1, \ldots, x_n) \leftarrow x:h_{rec}(x_1, \ldots, x_n), x_1:s_1, \ldots, x_n:s_n$ is added, and for each type rule $h(\tilde{t}) :=> u \leftarrow y_1:r_1, \ldots, y_k:r_k$ the type clause $u:h_{rec}(\tilde{t}) \leftarrow y_1:\delta(r_1), \ldots, y_k:\delta(r_k)$ is generated, where $\delta(r_i)$ is generated from

[1] see Example 2

r_i by the replacement of h by h_{rec} (i.e. the types of the arguments $\tilde{t}$ are not checked). Note that the type function h_{rec}/n may occur in these type clauses, only.

For each object variable x (and type variable α, respectively) occuring in some type clause, there is a type constraint $t : s$ in the body of this clause such that $x \in var(t)$ and $s \in \mathbb{V}_{type} \cup \mathcal{T}$ (and $\alpha \equiv t$, $s \in \mathbb{F}_{sort}$, respectively). Moreover, there is not any type constraint $x:a$ of some clause such that $x \in \mathbb{V}_0$ and $a \in \mathbb{F}_{sort}$. Thus different notations (and sorts) of both kinds of variables are not necessary. Whether a variable occuring in some type clause is an object variable or a type variable can be deduced from syntactical information on this type rule. It is easy to define a binary relation $=$ by means of definite clauses such that, for the least Herbrand model $\mathcal{H}^=$ of this set of clauses, $t = u$ is valid in $\mathcal{H}^=$ if, and only if $t \in \mathcal{O} \cup \mathcal{T} \cup \mathbb{F}_{sort}$ and u is identical to t. Consequently, we can consider one-sorted logic. However, substitutions are defined in a restricted way (see Section 2) and take into account the deduced sorts of variables.

The semantics of the types and type sorts are determined by the least Herbrand model $\mathcal{H}^t$ of the generated clauses. The set $\{ t \in \mathcal{O}_0 \mid t:s \in \mathcal{H}^t \}$ can be regarded as the interpretation of some type $s \in \mathcal{T}_0$. Analogously, the set $\{ s \in \mathcal{T}_0 \mid s:a \in \mathcal{H}^t \}$ can be regarded as the interpretation of a type sort $a \in \mathbb{F}_{sort}$.

The union of the set of type clauses generated from specifications of type sorts and types and from the set of clauses for defining the relation $=$ is called constraint theory. The least Herbrand model of the constraint theory is the union of the Herbrand models $\mathcal{H}^t$ and $\mathcal{H}^=$.

5 Logic Programming with Type Constraints

A logic program with type constraints consists of definite clauses with type declarations for each relation and of specifications of type sorts and types, where the relations ":" and "=" must not occur in the head of these clauses. The definite clauses and the corresponding type declarations for the relations can be transformed into clauses with type constraints each being of the form $H \leftarrow \mathcal{C} \diamond B_1, \ldots, B_n$, where H (called head) and $B_1, \ldots, B_n$ (called body) are atomic formulas, and $\mathcal{C}$ is a set of type constraints and equations. The type constraints belonging to $\mathcal{C}$ are generated from the type declarations of the predicates occuring in the body, where we assume that a predicate can only be called, if the arguments satisfy the types given in the type declaration of this predicate. A goal is a clause without head. Thus we can assume that a set of clauses with type constraints and a constraint theory consisting of type clauses and equations are given. Using type inference the number of type constraints can be reduced (in many cases drastically).

Since a logic program with type constraints consists of definite clauses, the least Herbrand model can be used for the semantics. This Herbrand model is an extension of the least Herbrand model of the constraint theory. A ground type term s is interpreted as the set of ground object terms t such that $t:s$ belongs to the least Herbrand model. Analogously, a type sort is interpreted.

Let $\mathcal{H}^c$ be the least Herbrand model of the given constraint theory. A deriva-

tion step takes a goal $\leftarrow C_A \diamond A_1, \ldots, A_i, \ldots, A_n$, takes a variant of a clause $H \leftarrow C_B \diamond B_1, \ldots, B_m$ of the given program, and returns a new goal $\leftarrow C_A \cup C_B \cup \{H = A_i\} \diamond A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n$, if the constraints $C_A \cup C_B \cup \{H = A_i\}$ are satisfiable in $\mathcal{H}^c$. A derivation sequence is a possibly infinite sequence of goals connected by derivation steps. A derivation sequence is called succesful if it is finite and its last goal contains only constraints. This inference rule is a special case of the resolution with restricted quantifiers described in [4]. Thus soundness and completeness of our system follow from the results presented in this book.

The strategy of deduction can be modified for reasons of efficiency. In every derivation step the constraints C can be replaced by simplified constraints C_s, if $\mathcal{H} \models C \leftrightarrow C_s$. The complete test for satisfiability of the constraints can be replaced by a checkup of necessary conditions of satisfiability, if the satisfiability of the relevant constraints is proved in the last derivation step and the satisfiability of the other sets of constraints (belonging to the derivation sequence) follows from this proof.

6 Type Constraint Processing

For constraint processing we need rules for the test of satisfiability and rules for the simplification of a set of constraints. Since the constraint theory is a set of definite clauses, computing with constraints can be based on SLD-resolution. Because of the assumptions with respect to the type rules, special computation rules can be applied for the SLD-resolution.

The subset of equations contained in a set of constraints can be deleted, if there is a most general unifier of these equations (in the other case the constraints cannot be satisfied) and this unifier is applied to the set of constraints. Therefore, we consider only sets of type constraints in this section. Futhermore, we assume that the type terms occuring in a set of type constraints are ground terms. This means that the type parameters of a logic program with type constraints are assumed to be input variables (or that these variables can be bound by abstract interpretation). If a SLD-resolution step is applied to a type constraint which does not contain any type term with variables, then the new type constraints cannot contain type terms with variables either.

In the following let $\mathcal{R}^t$ be a finite set of type clauses and let $\mathcal{H}^t$ be the least Herbrand model of $\mathcal{R}^t$. The following theorem is devoted to an important property of type checking (i.e. checking if a type constraint without any variable holds).

Theorem 1 *If $t:s$ is a type constraint with $var(t:s) = \emptyset$, then it is decidable whether $t:s$ follows from $\mathcal{R}^t$ or not.*

Proof (*Sketch*). Let $\mathcal{N}_f$ be the set of ordered tuples of natural numbers, where the first element of a tuple is the greatest number, i.e. $n_i \geq n_j$ for each tuple $(n_1, \ldots, n_k)$ of $\mathcal{N}_f$, if $i \leq j$. An ordering between the elements of $\mathcal{N}_f$ is defined by: $(n_1, \ldots, n_{k_n}) > (m_1, \ldots, m_{k_m})$ iff one of the following conditions is satisfied

- there is an i with $1 \leq i \leq k_n$ and $i \leq k_m$ such that $n_i > m_i$ and $n_j = m_j$ for all j with $1 \leq j < i$ or
- $k_n > k_m$ and $n_j = m_j$ for all j with $1 \leq j \leq k_m$.

This binary relation is a linear ordering in $\mathcal{N}_f$ and is well-founded in $\mathcal{N}_f$, i.e. there is no infinite sequence $\{\vec{n}_i\}$ of elements of $\mathcal{N}_f$ such that $\vec{n}_i > \vec{n}_{i+1}$ for all $i \geq 0$. An ordering for a set of pairs $\{(\vec{n}, \vec{m}) \mid \vec{n}, \vec{m} \in \mathcal{N}_f\}$ is defined by: $(\vec{n}_1, \vec{n}_2) > (\vec{m}_1, \vec{m}_2)$ iff $\vec{n}_1 > \vec{m}_1$ or $\vec{n}_1 = \vec{m}_1$ and $\vec{n}_2 > \vec{m}_2$.

Each type constraint is mapped by a function Ψ to a pair of tuples of natural numbers. This function Ψ and the functions $\Phi_t : \mathcal{T}_0 \rightarrow \mathcal{N}_f$, $\Phi_c : \mathcal{O}_0 \rightarrow \mathcal{N}_f$ are defined by:

- For each $s \in \mathcal{T}_0$, $\Phi_t(s)$ is an ordered tuple $(n_1, \ldots, n_k) \in \mathcal{N}_f$ such that there is a one-to-one mapping beetween the symbols of type functions occuring in s and the natural numbers of the tuple, where each $h \in \mathbb{F}_{type}$ occuring in s is mapped to a natural number n_i with $n_i = \eta(h)$.
- For each $t \in \mathcal{O}_0$, $\Phi_c(t)$ is an ordered tuple $(m_1, \ldots, m_k) \in \mathcal{N}_f$ such that there is a one-to-one mapping beetween the object constructors occuring in t and the natural numbers of the tuple, where each $g \in \mathbb{F}_{obj}$ occuring in t is mapped to a natural number m_i with $m_i = \eta(g)$.
- For each $t \in \mathcal{O}_0$ and $s \in \mathcal{T}_0$, $\Psi(t:s) = (\Phi_t(s), \Phi_c(t))$.
- For each $a \in \mathbb{F}_{sort}$ and $s \in \mathcal{T}_0$, $\Psi(s:a) = (\Phi_t(s), (\eta(a)))$.

Then Theorem 1 follows from the proposition: *If $t:s \leftarrow u_1:r_1, \ldots, u_k:r_k$ is a ground instance of a type clause of $\mathcal{R}^t$, then holds: $\Psi(t:s) > \Psi(u_i:r_i)$ for all i with $1 \leq i \leq k$.* ◁

Thus a type constraint without any variable can be reduced to "`true`" or "`false`" and it is sufficient to consider type constraints $t:s$ with $s \in \mathcal{T}_0$ and $var(t) \neq \emptyset$. Interesting cases are: "Is there a computed answer for $t:s$" and "Is there a computed answer for $x:(s_1, \ldots, s_n)$?" In general these problems are not decidable. However if there is not any recursive type clause in $\mathcal{R}^t$, then these problems are decidable.

In support of type constraint processing a function Δ of $\mathcal{O} \cup \mathcal{T} \cup \mathcal{T}_0^* \cup \mathbb{F}_{sort}$ to $Pow(\mathbb{F}_{obj}) \cup Pow(\mathbb{F}_{type})$ is defined ($Pow(M)$ denotes the power set of a set M). This function is based on a dependency graph (see also [21]). Before the unary function Δ can be defined we need the definitions of two other unary functions. Note that these definitions are correct since we suppose that there is a level mapping satisfying the conditions (C0) - (C3).

Definition 4 *ζ is a function of $\mathbb{F}_{type} \cup \mathbb{F}^*_{type}$ to $Pow(\mathbb{F}_{obj})$ which is defined in the following way:*

- *For each $h \in \mathbb{F}_{type}$, $\zeta(h)$ is the set of all $f \in \mathbb{F}_{obj}$ satisfying one of the following conditions:*
  - *there is a type clause $f(\tilde{t}):h(\tilde{u}) \leftarrow C$ in $\mathcal{R}^t$;*
  - *there is a type clause $x : h(\tilde{u}) \leftarrow x : (g_1(\tilde{u}_1), \ldots, g_k(\tilde{u}_k)) \uplus C$ in $\mathcal{R}^t$, $x \in \mathbb{V}_{obj}$, and $f/m \in \zeta((g_1, \ldots, g_k))$.*

- *For each* $(g_1,\ldots,g_k) \in \mathbb{F}^*_{type}$, $\zeta((g_1,\ldots,g_k)) = \zeta(g_1) \cap \ldots \cap \zeta(g_k)$.

Definition 5 *ξ is a function of $\mathbb{F}_{sort}$ to $Pow(\mathbb{F}_{type})$ which is defined in the following way: For each $a \in \mathbb{F}_{sort}$, $\xi(a)$ is the set of all $h \in \mathbb{F}_{type}$ satisfying one of the following conditions:*

- *there is a type clause $h(\tilde{x}):a \leftarrow C$ in $\mathcal{R}^t$;*
- *there is a type clause $x:a \leftarrow x:b$ in $\mathcal{R}^t$, $x \in \mathbb{V}_{type}$, and $h \in \xi(b)$.*

Definition 6 *For each $t \in \mathcal{O} \cup \mathcal{T} \cup \mathcal{T}_0^* \cup \mathbb{F}_{sort}$, $\Delta(t)$ is defined by:*

- *if $t \in \mathbb{V}_{obj}$, then $\Delta(t) = \mathbb{F}_{obj}$;*
- *if $t \in \mathbb{V}_{type}$, then $\Delta(t) = \mathbb{F}_{type}$;*
- *if $t \in \mathcal{O}$ and $t \equiv f(t_1,\ldots,t_n)$, then $\Delta(t) = \{f\}$;*
- *if $t \in \mathcal{T}$ and $t \equiv h(u_1,\ldots,u_m)$, then $\Delta(t) = \zeta(h)$;*
- *if $t \in \mathcal{T}_0^*$ and $t \equiv (g_1(\tilde{u}_1),\ldots,g_k(\tilde{u}_k))$, then $\Delta(t) = \zeta(g_1) \cap \ldots \cap \zeta(g_k)$;*
- *if $t \in \mathbb{F}_{sort}$, then $\Delta(t) = \xi(t)$.*

The function Δ depends only on the specifications of the types and type sorts. Therefore, this function can be generated at compile time. The following lemma shows the advantages of the function Δ. The proof of this lemma is straightforward.

Lemma 1 *Let $a \in \mathbb{F}_{sort}$, $h(\tilde{u}), s, s_1, \ldots, s_m \in \mathcal{T}_0$, and $f(\tilde{t}) \in \mathcal{O}$.*

- *If $h \notin \Delta(a)$, then $\mathcal{H}^t \not\models h(\tilde{u}):a$.*
- *If $f \notin \Delta(s)$, then $\mathcal{H}^t \not\models \exists f(\tilde{t}):s$.*
- *If $\Delta((s_1,\ldots,s_m)) = \emptyset$, then $\mathcal{H}^t \not\models \exists x(x:(s_1,\ldots,s_m))$.*

Necessary conditions for satisfiability of type constraints follow from this lemma. Furthermore, the number of type clauses which are applicable to a type constraint can be reduced, if this lemma is used. The following inference rules take into account the statements of Lemma 1.

Let C be a set of type constraints with two kinds of constraints: $t:s$ with $t \in \mathcal{O} \setminus \mathbb{V}_{obj}$, $var(s) = \emptyset$, and $x:(s_1,\ldots,s_n)$ with $x \in \mathbb{V}_{obj}$, $(s_1,\ldots,s_n) \in \mathcal{T}_0^*$. For these two kinds of constraints, we formulate different inference rules. Furthermore, we formulate different inference rules for the two kinds of clauses: $u:r \leftarrow C$ with $u \notin \mathbb{V}$, and $x:r \leftarrow C$ with $x \in \mathbb{V}_{obj}$. There is one inference rule for each of these four cases (x and y denote object variables):

$$(1)\quad \frac{\{t:s\} \cup C}{u:r \leftarrow C_r \;\;\; \sigma(C_r) \cup \sigma(C)}$$

if t and u are not variables, and σ is a most general unifier of $u:r$ and $t:s$

$$(2)\quad \frac{\{t:s\} \cup C}{y:r \leftarrow y:(r_1,\ldots,r_k) \uplus C_r \;\;\; \{t:\sigma(r_1),\ldots,t:\sigma(r_k)\} \cup \sigma(C_r) \cup C}$$

if t is not any variable, σ is a match substitution such that $\sigma(r) = s$, and $\Delta(t) \cap \Delta((r_1,\ldots,r_k)) \neq \emptyset$

$$(3)\quad \frac{x:(s_1,\ldots,s_n) \uplus C \quad y:r \leftarrow y:(r_1,\ldots,r_k) \uplus C_r}{x:(s_1,\ldots,s_{i-1},\sigma(r_1),\ldots,\sigma(r_k),s_{i+1},\ldots,s_n) \uplus \sigma(C_r) \cup C}$$

if σ is a match substitution such that $\sigma(r) = s_i$ and $\Delta((s_1,\ldots,s_n)) \cap \Delta((r_1,\ldots,r_k)) \neq \emptyset$

$$(4)\quad \frac{x:(s_1,\ldots,s_n) \uplus C \quad f(\tilde{u}_1):r_1 \leftarrow C_1 \quad \vdots \quad f(\tilde{u}_n):r_n \leftarrow C_n}{\sigma(C_1) \cup \ldots \cup \sigma(C_n) \cup \sigma(C)}$$

if σ is a most general unifier of $\{s_1 = r_1,\ldots,s_n = r_n, x = f(\tilde{u}_1),\ldots,x = f(\tilde{u}_n)\}$

Inference rule (1) is the SLD-resolution rule for the considered case. With respect to the considered cases, the inference rules (2) and (3) correspond to the SLD-resolution rule with an additional condition for the choice of type clauses. Inference rule (4) can be regarded as a sequence of SLD-resolution steps with a special selection rule.

Using these inference rules a *derivation* of a finite set of type constraints is usually defined. Such a derivation is called *succesful* if it is finite and the last set of type constraints is empty. Obviously, each derivation of a set of type constraints is an SLD-derivation. By contrast with SLD-derivation the additional conditions of the inference rules (2) and (3) do not result in a restriction of succesful derivations (see Lemma 1). The following theorem is a consequence of the results on SLD-resolution (see e.g. [17]) and Lemma 1.

Theorem 2 *Let C be a finite set of type constraints which is satisfiable in $\mathcal{H}^t$.*

- *If C_1 is generated from C by applying an inference rule, then C_1 is satisfiable in $\mathcal{H}^t$.*
- *There is a succesful derivation of C.*

Note that $\sigma(C_r)$ does not contain any variables in the inference rules (2) and (3). Since type checking is decidable (Theorem 1) the condition $\mathcal{H}^t \models \sigma(C_r)$ can be used for the choice of a type rule. Thus inference rule (2) can be replaced by a more complex inference rule (2'):

$$(2')\quad \frac{\{t:s\} \cup C}{y:r \leftarrow y:(r_1,\ldots,r_k) \uplus C_r \;\;\; \{t:\sigma(r_1),\ldots,t:\sigma(r_k)\} \cup C}$$

if t is not any variable, σ is a match substitution such that $\sigma(r) = s$, $\mathcal{H}^t \models \sigma(C_r)$, and $\Delta(t) \cap \Delta((r_1,\ldots,r_k)) \neq \emptyset$

Inference rule (3') can be analogously generated from inference rule (3).

We assume that a finite set C of type constraints is given and that $t:s \in C$ implies $var(s) = \emptyset$. The following operations are proposed for type constraint processing:

1. Checking necessary conditions
   - If there is a type constraint $t:s \in C$ such that $\Delta(t) \cap \Delta(s) = \emptyset$ or if there is a set of variable constraints $x:(s_1,\ldots,s_m) \subseteq C$ such that $\Delta((s_1,\ldots,s_m)) = \emptyset$, then C cannot be satisfiable.

- If there is a type constraint $t:s \in \mathcal{C}$ or a set of variable constraints $x:(s_1,\ldots,s_m) \subseteq \mathcal{C}$ such that there is not any inference rule which can be applied to this constraint or to this set of constraints, then $\mathcal{C}$ cannot be satisfiable.

2. Type checking and simplification
   - Let $t:s \in \mathcal{C}$ be a type constraint with $var(t:s) = \emptyset$. If $\mathcal{H}^t \models t:s$, then this constraint can be deleted from $\mathcal{C}$. If $\mathcal{H}^t \not\models t:s$, then $\mathcal{C}$ cannot be satisfiable. For type checking inference rules (1) and (2) can be applied.
   - Inference rules (1), (2'), (3'), or (4) are applied with respect to a type constraint, if there is only one type clause which can be used.

3. Checking satisfiability
   - The inference rules are applied to generate a succesful derivation of $\mathcal{C}$.

Furthermore, we suggest that the number of applications of a recursive type clause is restricted by a computation rule. In such a case, the complete test of satisfiability is delayed.

Using the operations "type checking and simplification", the given set $\mathcal{C}$ of type constraints is replaced by a simplified set $\mathcal{C}_s$ of type constraints (if type checking is succesful; the other case is trivial). Because of completeness of SLD-resolution $\mathcal{H}^t \models \mathcal{C} \leftrightarrow \mathcal{C}_s$ holds.

If the type clauses satisfy certain conditions, some interesting properties can be deduced. For this, results on terminating logic programs can be also used (see e.g. [21]). For instance, if $x:s$ is a type constraint with $var(s) = \emptyset$ and the type clauses which are needed for the construction of the SLD-tree of $x:s$ are nonrecursive, then this SLD-tree is finite and the elements belonging to s can be generated by a finite number of derivation steps. In the following theorem, sufficient conditions of recursive type clauses are described in order to guarantee that the mentioned problems $t:s$ and $x:(s_1,\ldots,s_n)$ are decidable.

Theorem 3 *Supposed that the following conditions are satisfied for each type function $h \in \mathbb{F}_{type}$ having a recursive type clause in $\mathcal{R}^t$:*

- *If $f(\tilde{t_1}):h(\tilde{u}) \leftarrow \mathcal{C}_h$ is a recursive type clause of $\mathcal{R}^t$, then there is exactly one recursive type constraint $x:h(\tilde{v}) \in \mathcal{C}_h$.*
- *One of the following conditions holds for all recursive type clauses $f(\tilde{t_1}):h(\tilde{u}) \leftarrow \mathcal{C}_h$ of the type constructor h:*
  - *(a) if $x:h(\tilde{v}) \in \mathcal{C}_h$, then there is an i such that $v_i \in \tilde{v}$ is a proper subterm of $u_i \in \tilde{u}$;*
  - *(b) if $x:h(\tilde{v}) \in \mathcal{C}_h$, then $h(\tilde{u}) \equiv h(\tilde{v})$.*

Then the satisfiability of a finite set of type constraints[2] is decidable.

[2]Note that we suppose that the type terms occuring in this set do not contain any variables

For better control of type constraint processing the following conditions are very useful:

(1) Each type constructor belongs to a basic type sort (elements of $\mathbb{F}^b_{sort}$) and the interpretations of type terms $h_1(\tilde{u}_1)$ and $h_2(\tilde{u}_2)$ are disjoint, if h_1 and h_2 belong to different basic type sorts.

(2) Two different types are disjoint, if these types are defined by means of set definitions.

If condition (1) is satisfied and the type constructors h_1, h_2 belong to different basic type sorts, then $x:(h_1(\tilde{u}_1),h_2(\tilde{u}_2))$ cannot be satisfiable. Thus, condition (1) implies a necessary condition of satisfiability which can be easily checked. If condition (2) is satisfied, then the definition of the function Δ can be modified. Let s be a type defined by $s := \{t_1,\ldots,t_n\}$ where $s,t_1,\ldots,t_n$ are constants. Instead of $\Delta(s) = \{t_1,\ldots,t_n\}$ it is sufficient to define $\Delta(s) = s$ in such a case.

7 Related Work

The benefit of using type information is widely recognized within the field of knowledge representation and automated theorem proving. Mycroft and O'Keefe ([19]) have proposed a polymorphic type system for Prolog based on many-sorted logic. A formal semantics of this approach is discussed in [16]. A typed functional extension of logic programming is defined in [22]. OBJ is a logic programming language based on order-sorted equational logic ([15]). Logic programming with polymorphically order-sorted types is proposed in [23]. Other approaches of type concepts for logic programming are described in [20]. A framework for integrating logical deduction and sortal deduction to form deductive systems for logics with sorted variables is discussed in [5].

In [1] another direction of an approach for types in logic programming is given. Ordinary first-order terms are replaced by record structures and a special type unification is used. A further direction is proposed in [6] and [7]. Types are defined by functions on terms (by means of equations). Equations are also used for the type system in [11], but there is a clear distinction between types and functions (and predicates).

The general scheme of constraint logic programming was first defined in [13]. Since 1987, a great number of publications related to CLP have appeared, devoted to questions of semantics, new generalizations, implementation of concrete systems, applications, and other aspects. The CLP scheme was generalized in [12] where the process of defining a logic language is regarded as a hierarchical construction of constraint systems and the conditions imposed on constraint structures are relaxed. Several kinds of semantics for CLP are discussed in [8]. A survey on constraint logic programming is given in [14] and interesting papers of constraint logic programming are contained in [3].

A predicate logic with restricted quantifiers is discussed in [4]. Universal quantifiers with restrictions can be regarded as constraints for the corresponding variables.

Furthermore, a resolution principle for clauses with constraints was proposed in [4], where unification is replaced by testing constraints for satisfiability over a restriction theory. In [4] a sort theory is also mentioned as an example of such a restriction theory. Further examples of papers dealing with constraints and types are [2], [18], and [24]. A constraint system for logic programming with feature terms is presented in [2]. An abstract constraint system is introduced in [18] and a simple type system is discussed as an example of this constraint system. In [24] a class of monadic constraints is defined where variables range over sets of ground terms described by systems of set constraints and a unification algorithm for the corresponding constraint terms is given.

8 Concluding Remarks

Our future research will include extensions of the presented approach of a type concept in several directions. The conditional parts of type rules may be extended by constraints which are not type constraints, e.g. $s(x) :=> y <\!\!- p(x,y)$ (where $p/2$ is a predicate defined by definite clauses). Such constraints should be decidable and constructive, and the types of the type declaration of such a constraint should not depend on the type which is defined by means of this constraint. The presented type concept can be integrated into constraint logic programming. In such a case it is useful if the conditional part of a type rule can contain constraints of the corresponding domain, e.g. $interval(x,y) :=> z <\!\!- z{:}nat, x < z, z < y$ (the computation with types has to be extended). Another interesting extension is the use of predicates connected to an external database in the conditional part of type rules. For instance, the type *"student"* may be defined by $student :=> x <\!\!- p\_student(x)$, where we assume that the predicate $p\_student/1$ is connected to an external database and that this predicate can generate the elements belonging to the corresponding relation of the database or can check if an element belongs to this relation. Besides of the mentioned extensions, our future research will include the development of improved methods for computing with type constraints. Note that type rules can also be regarded as special rewrite rules. This is discussed in [9].

References

[1] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *J. Logic Programming*, 3:185–215, 1986.

[2] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *Proc. Int. Conf. Fifth Generation Computer Systems 1992*, pages 1012–1021, Tokyo, 1992. ICOT, IOS Press.

[3] F. Benhamou and A. Colmerauer, editors. *Constraint Logic Programming (Selected Research)*. MIT Press, Cambridge, London, 1993.

[4] H.-J. Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*, volume 568 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1991.

[5] A.M. Frisch. The substitutional framework for sorted deduction: Fundamental result on hybrid reasoning. *Artificial Intelligence*, 49:161–198, 1991.

[6] U. Furbach. *Logische und Funktionale Programmierung: Grundlagen einer Kombination*. Vieweg Verlag, Braunschweig, 1991.

[7] U. Furbach and S. Hölldobler. Equations, order-sortedness and inheritance in logic programming. FKI-110-89, TU-München, 1989.

[8] M. Gabbrielli and G. Levi. Modeling answer constraints in constraint logic programming. In Koichi Furukawa, editor, *Proc. 8th Int. Conf. Logic Programming*, pages 238–25 MIT Press, Cambridge (Mass.), London, 1991.

[9] H.-J. Goltz. A constructive type system based on data terms. In D. Pearce and G. Wa ner, editors, *Logics in AI - JELIA '92*, volume 633 of *Lecture Notes in Artificial Intel gence*, pages 279–303. Springer-Verlag, Berlin, Heidelberg, 1992.

[10] J. Grabowski, P. Lescanne, and W. Wechler, editors. *Algebraic and Logic Programmin* volume 49 of *Mathematical Research*. Akademie-Verlag, Berlin, 1988. Also as LNC 343, Springer-Verlag, 1988.

[11] M. Hanus. Logic programming with type specifications. In *[20]*, pages 91–140, 1992.

[12] M. Höhfeld and G. Smolka. Definite relations over constraint language. LILOG-Repo 53, IBM-Deutschland,, Germany, 1988.

[13] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th Principles Programming Languages*, pages 111–119, Munich, 1987.

[14] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Logic Progra ming*, 1994. to appear.

[15] J.-P. Jouannaud, C. Kirchner, H. Kirchner, and A. Megrelis. OBJ: Programming w equalities, subsorts, overloading and parameterization. In *[10]*, pages 41–52, 1988.

[16] T.K. Lakshman and U.S. Reddy. Typed prolog: A semantic reconstruction of Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Logic Programmi Proc. Int. Symposium 1991*, pages 201–217. MIT Press, 1991.

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

[18] M. Mamede and L. Monteiro. A constraint logic programming scheme for taxono reasoning. In K. Apt, editor, *Logic Programming, Proceedings of the Joint Internatio Conference and Symposium on Logic Programming*, pages 255–269, Cambridge, Lond 1992. MIT Press.

[19] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial In ligence*, 23:295–307, 1984.

[20] F. Pfenning, editor. *Types in Logic Programming*. Logic Programming Series. The M Press, Cambridge (Mass.), London, 1992.

[21] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Note Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1990.

[22] D.W. Shin, J.H. Nang, S.R. Maeng, and J.W. Cho. A typed functional extensio logic programming. *Journal New Generation Computing*, 10:197–221, 1992.

[23] G. Smolka. Logic programming with polymorphicaly order-sorted types. In *[10]*, p 53–70, 1988.

[24] T.E. Uribe. Sorted unification using set constraints. In D. Kapur, editor, *Autom Deduction – CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, p 163–177. Springer-Verlag, Berlin, Heidelberg, 1992.

Proving Termination of Prolog Programs

Paolo Mascellani and Dino Pedreschi

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy
e_mail: pedre@di.unipi.it

Abstract

We provide a method for conducting termination proofs for pure Prolog programs, based on the notion of cyclically and acyclically acceptable programs. The proposed method improves on that of Apt and Pedreschi [AP90] (later refined in [AP94]) in two directions: mutual recursion, and modularity. On one side, a finer tuned treatment of mutual recursion allows us to construct simpler proofs, which adopt more natural level mappings (i.e., termination functions.) On the other side, the proposed method exhibits a higher degree of modularity in conducting termination proofs. As a consequence, the proposed method is more amenable to be (partly) supported by automatic tools. The method is illustrated on a number of realistic Prolog programs.

1 Introduction

1.1 Motivation

The study of termination of logic and Prolog programs is an active research area, as witnessed by De Schreye and Decorte in their recent survey [DSD93]. A stream of methods for proving termination was started by Bezem for logic programs [Bez89], and by Apt and Pedreschi for pure Prolog programs [AP90], and later refined in [AP93, AP94, DSVB92]. These methods are based on the concept of a *level mapping*, i.e., a mapping of the Herbrand base of a program into natural numbers, which plays the role of a termination function in the proofs.

Unfortunately, the level mapping required in a termination proof is sometimes unnatural, and therefore difficult to be guessed. Also, the structure of the proof often lacks modularity. The revised method presented by Apt and Pedreschi in [AP94] partly fixes these deficiencies. However, as the realistic examples presented in this paper will show, the method in [AP94] still requires unnatural level mappings in the case of *mutually recursive* predicates, and support a degree of modularity which is unable to properly deal with natural programs.

In the present paper, we provide a refined proof method for termination of pure Prolog programs, building on that of Apt and Pedreschi. The new method is based on

the notion of *cyclically* and *acyclically acceptable* programs, and improves on that of Apt and Pedreschi in two directions: mutual recursion, and modularity. On one side, a finer tuned treatment of mutual recursion allows us to construct simpler proofs, which adopt more natural level mappings. On the other side, the proposed method exhibits a higher degree of modularity in conducting termination proofs.

1.2 Plan of the Paper

This paper is organized as follows. In Section 2 we recall the proof method of acceptability from [AP90, AP93], and provide some examples, which point out the deficiencies of the method. In Section 3 we recall the proof method of semi-acceptability from [AP94], and apply it to the same examples, thus pointing out the solved deficiencies and those which are not properly dealt with. In Section 4 we introduce the new proof method of cyclic and acyclic acceptability, and apply it to the same examples, thus pointing out how the cited deficiencies are dealt with. Finally, we provide some concluding remarks.

1.3 Preliminaries

Throughout this paper we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. In particular, for a logic program P (or simply a *program*) we denote the Herbrand Base of P by B_P, the least Herbrand model of P by M_P and the immediate consequence operator by T_P. Also, we use Prolog's convention identifying in the context of a program each string starting with a capital letter with a variable, reserving other strings for the names of constants, terms or relations. So, for example Xs stands for a variable whereas xs stands for a term.

In the programs we use Prolog's list notation. The constant [ ] denotes the empty list and [. | .] is a binary function which given a term x and a list xs produces a new list [x | xs] with head x and tail xs. By convention, identifiers ending with "s", like xs, will range over lists. The standard notation [$x_1, \ldots, x_n$], for $n \geq 0$, is used as an abbreviation of [x_1 | [$\cdots$ [x_n | [ ]] $\cdots$]]. $len(xs)$ denotes the length of the list xs.

Throughout the paper we consider LD-resolution. It is obtained from SLD-resolution by using Prolog first-left selection rule. The concepts of LD-derivation, LD-refutation, LD-tree, etc. are defined in the usual way. By "pure Prolog" we mean in this paper the LD-resolution combined with the depth first search in the LD-trees.

Finally, in this paper we concentrate on *left terminating* program, according to the following definition from Apt and Pedreschi [AP90]:

Definition 1.1 A program is *left terminating* iff all its LD-derivations starting with a ground goal are finite. □

All proof method considered in this paper are sound and complete characterizations of the class of left terminating programs. From empirical investigations, we found that most (if not all) practical pure Prolog programs are left terminating (e.g., most pure programs in Sterling and Shapiro's book [SS86].)

2 Acceptable Programs

The notion of *acceptability* provides a declarative characterization of terminating Prolog programs [AP90, AP93]. We begin by recalling some useful definitions.

Definition 2.1 A *level mapping* for a program P is a function $|.| : B_P \rightarrow \omega$ mapping ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the *level* of A. □

Definition 2.2 Let P be a program, $|.|$ a level mapping for P and I_P a (not necessarily Herbrand) interpretation of P.

- A clause of P is called *acceptable* with respect to $|.|$ and I_P, if I_P is a model of the clause and for all its ground instances $A \leftarrow As, B, Bs$: $I_P \models As \Rightarrow |A| > |B|$.
- A program P is called *acceptable* with respect to given level mapping and interpretation, if all its clauses are. A program is called *acceptable* if it is acceptable with respect to some level mapping and interpretation. □

Intuitively the use of the premise $I_P \models As$ expresses the fact that when, in the evaluation of the goal $\leftarrow As, B, Bs$ using the leftmost selection rule, the atom B is reached, the atoms As are already refuted. Consequently, by the soundness of the LD-resolution, these atoms are true in I_P. Next, the notion of *boundedness* is the tool for reasoning about termination of non-ground goals.

Definition 2.3 Let P be a program, A be an atom, G a goal, $|.|$ a level mapping for P and I_P a model of P:

- A is *bounded* with respect to $|.|$ ($|A| < \infty$), iff:

$$\exists\, k \in \omega : \forall\, A' \in ground(A) : |A'| \leq k.$$

  Level mappings can be extended to non-ground bounded atoms:

$$|A| = \min\{k \mid \forall\, A' \in ground(A) : |A'| \leq k\}$$

- G is *bounded* with respect to I_P and $|.|$ ($|G| < \infty$), iff:

$$\forall \leftarrow As, A, Bs \in ground(G) : I_P \models As \Rightarrow |A| < \infty.$$

□

We can now relate the notions of acceptable and left terminating programs.

Theorem 2.4 *Let P be a left terminating program. Then for some level mapping $|.|$ and interpretation I_P of P:*

- *P is acceptable w.r.t. $|.|$ and I_P;*

- *for every goal G, G is bounded w.r.t. $|.|$ and I_P iff all LD-derivations of $P \cup \{G\}$ are finite.*

Proof. See [AP93]. □

As a consequence, acceptability provides a sound and complete proof method for left termination:

Theorem 2.5 *A program is acceptable iff it is left terminating.* □

2.1 Examples

Maximum of a list of numbers

Consider the following program `LISTMAX`, which computes the maximum of a list of natural numbers:

```
listmax([],0) ←
listmax([X|Xs],Xmax) ←listmax(Xs,Xmax1), selmax(X,Xmax1,Xmax)
selmax(X,Y,X) ←gte(X,Y)
selmax(X,Y,Y) ←gte(Y,X)
gte(X,Y) ←sum(Z,Y,X)
sum(X,0,X) ←
sum(X,s(Y),s(Z)) ←sum(X,Y,Z)
```

In order to prove that `LISTMAX` is left terminating, we introduce the Herbrand interpretation $I_{LISTMAX}$, defined by the following relations:

$$\begin{array}{lll} I_{LISTMAX} \models \texttt{listmax(xs,xmax)} & \Leftrightarrow & \texttt{xmax} \in \texttt{xs} \vee \texttt{xmax} = \texttt{0} \\ I_{LISTMAX} \models \texttt{selmax(x,y,max)} & \Leftrightarrow & \texttt{max} = \texttt{x} \vee \texttt{max} = \texttt{y} \\ I_{LISTMAX} \models \texttt{gte(x,y)} & & \\ I_{LISTMAX} \models \texttt{sum(x,y,z)} & & \end{array}$$

and the level mapping $|.|$:

$$\begin{array}{lcl} |\texttt{sum(x,y,z)}| & = & val(\texttt{z}) \\ |\texttt{gte(x,y)}| & = & val(\texttt{x}) + 1 \\ |\texttt{selmax(x,y,z)}| & = & max([\texttt{x},\texttt{y}]) + 2 \\ |\texttt{listmax(xs,xmax)}| & = & len(\texttt{xs}) + max(\texttt{xs}) + 3 \end{array}$$

where:

$$\begin{array}{lcl} val(\texttt{0}) = val([]) = val([\texttt{x}|\texttt{xs}]) & = & 0 \\ val(\texttt{s(x)}) & = & val(\texttt{x}) + 1 \end{array}$$

and:

$$max([]) = 0$$

$$max([\texttt{x}|\texttt{xs}]) = \begin{cases} val(\texttt{x}) & \text{if } val(\texttt{x}) > max(\texttt{xs}) \\ max(\texttt{xs}) & \text{otherwise} \end{cases}$$

The interpretation $I_{LISTMAX}$ is clearly a model of LISTMAX. Moreover, the program LISTMAX is acceptable w.r.t. $I_{LISTMAX}$ and $|.|$; in fact:

$$|\texttt{sum(x,s(y),s(z))}| = val(\texttt{z}) + 1 > val(\texttt{z}) = |\texttt{sum(x,y,z)}|$$
$$|\texttt{gte(x,y)}| = val(\texttt{x}) + 1 > val(\texttt{x}) = |\texttt{sum(z,y,x)}|$$
$$|\texttt{selmax(x,y,x)}| = max(val(\texttt{x}), val(\texttt{y})) + 2 > val(\texttt{x}) + 1 = |\texttt{gte(x,y)}|$$
$$|\texttt{selmax(x,y,y)}| = max(val(\texttt{x}), val(\texttt{y})) + 2 > val(\texttt{y}) + 1 = |\texttt{gte(y,x)}|$$
$$|\texttt{listmax([x|xs],xmax)}| = len(\texttt{[x|xs]}) + max(\texttt{[x|xs]}) + 3 >$$
$$> len(\texttt{xs}) + max(\texttt{xs}) + 3 = |\texttt{listmax(xs,xmax1)}|$$
$$I_{LISTMAX} \models \texttt{listmax(xs,xmax1)} \Rightarrow$$
$$|\texttt{listmax([x|xs],xmax)}| = len(\texttt{[x|xs]}) + max(\texttt{[x|xs]}) + 3 >$$
$$> max(\texttt{x},\texttt{xmax1}) + 2 = |\texttt{selmax(x,xmax1,xmax)}|$$

Therefore, we can conclude that the program LISTMAX is left terminating. Moreover, any goal $\leftarrow$ listmax(xs,Xmax) is bounded, and hence every LD-derivation for such goal is finite.

Powers

Consider now the following program POWER, which computes x^y, where x and y are natural numbers:

```
power(X,0,s(0)) ←
power(X,s(Y),Z) ← power(X,Y,W),times(X,W,Z)
times(X,0,0) ←
times(X,s(Y),Z) ← times(X,Y,W),sum(X,W,Z)
sum(X,0,X) ←
sum(X,s(Y),s(Z)) ← sum(X,Y,Z)
```

Consider the Herbrand interpretation I_{POWER} defined as follows:

$$I_{POWER} \models \texttt{power(x,y,z)} \Leftrightarrow z = x^y \ (with\ 0^0 = 1)$$
$$I_{POWER} \models \texttt{times(x,y,z)} \Leftrightarrow z = xy$$
$$I_{POWER} \models \texttt{sum(x,y,z)} \Leftrightarrow z = x + y$$

I_{POWER} is clearly a model of POWER. Consider next the level mapping $|.|$:

$$|\texttt{power(x,y,z)}| = (x+2)^y$$
$$|\texttt{times(x,y,z)}| = (x+1)y$$
$$|\texttt{sum(x,y,z)}| = y$$

POWER is acceptable w.r.t. I_{POWER} and $|.|$; in fact:

$$|\texttt{power(x,s(y),z)}| = (x+2)^{y+1} > (x+2)^y = |\texttt{power(x,y,z)}|$$

$$I_{POWER} \models \texttt{power(x,y,w)} \Rightarrow$$
$$|\texttt{power(x,s(y),z)}| = (x+2)^{y+1} > (x+1)x^y = |\texttt{times(x,w,z)}|$$
$$|\texttt{times(x,s(y),z)}| = (x+1)(y+1) > (x+1)y = |\texttt{times(x,y,z)}|$$
$$I_{POWER} \models \texttt{times(x,y,w)} \Rightarrow$$
$$|\texttt{times(x,s(y),z)}| = (x+1)(y+1) > xy = |\texttt{sum(x,w,z)}|$$
$$|\texttt{sum(x,s(y),s(z))}| = y + 1 > y = |\texttt{sum(x,y,z)}|$$

Moreover, any goal $\leftarrow$ power(x,y,Z) is bounded, and hence every LD-derivation for such goal is finite.

Binary search

Consider the following program SEARCH, implementing the binary (or dichotomic) search on a list of pairs $(Key, Value)$ ordered with respect to Key:

```
search(N,Xs,M) ← divide(Xs,Xs1,X,Y,Xs2), switch(N,X,Y,Xs1,Xs2,M)
switch(N,N,M,Xs1,Xs2,M) ←
switch(N,X,Y,Xs1,Xs2,M) ← N>X, search(N,Xs2,M)
switch(N,X,Y,Xs1,Xs2,M) ← N<X, search(N,Xs1,M)
```

augmented by a DIVIDE program defining relation divide, which we omit.

Consider the Herbrand interpretation I_{SEARCH}:

$$I_{SEARCH} \models \texttt{search(n,xs,m)}$$
$$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Leftrightarrow len(\texttt{xs}) = 1 + len(\texttt{xs1}) + len(\texttt{xs2})$$
$$I_{SEARCH} \models \texttt{switch(n,x,y,xs1,xs2,m)}$$
$$I_{SEARCH} \models \texttt{x} > \texttt{y}$$
$$I_{SEARCH} \models \texttt{x} < \texttt{y}$$

I_{SEARCH} is clearly a model of SEARCH if it is a model of DIVIDE. Next, consider the level mapping $|.|$:

$$|\texttt{search(x,xs,m)}| = 2\ len(\texttt{xs}) + 1$$
$$|\texttt{divide(xs,xs1,x,y,xs2)}| = len(\texttt{xs})$$
$$|\texttt{switch(x,x,y,xs1,xs2,m)}| = 2\ (len(\texttt{xs1}) + len(\texttt{xs2}) + 1)$$
$$|\texttt{x} < \texttt{y}| = |\texttt{x} > \texttt{y}| = 0$$

SEARCH is acceptable w.r.t. $|.|$ and I_{SEARCH} if DIVIDE is; in fact:

$$|\texttt{search(n,xs,m)}| = 2\ len(\texttt{xs}) + 1 > len(\texttt{xs}) = |\texttt{divide(xs,xs1,x,y,xs2)}|$$
$$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Rightarrow$$
$$|\texttt{search(n,xs,m)}| = 2\ len(\texttt{xs}) + 1 > 2\ (len(\texttt{xs1}) + len(\texttt{xs2}) + 1) =$$
$$= |\texttt{switch(n,x,y,xs1,xs2,m)}|$$
$$|\texttt{switch(n,x,y,xs1,xs2,m)}| = 2\ (len(\texttt{xs1}) + len(\texttt{xs2}) + 1) > 0 =$$

$$= |\mathtt{n} > \mathtt{x}| = |\mathtt{n} < \mathtt{x}|$$

$$|\mathtt{switch}(\mathtt{n},\mathtt{x},\mathtt{y},\mathtt{xs1},\mathtt{xs2},\mathtt{m})| = 2\ (len(\mathtt{xs1}) + len(\mathtt{xs2}) + 1) > 2\ len(\mathtt{xs2}) + 1 =$$
$$= |\mathtt{search}(\mathtt{n},\mathtt{xs2},\mathtt{m})|$$

$$|\mathtt{switch}(\mathtt{n},\mathtt{x},\mathtt{y},\mathtt{xs1},\mathtt{xs2},\mathtt{m})| = 2\ (len(\mathtt{xs1}) + len(\mathtt{xs2}) + 1) > 2\ len(\mathtt{xs1}) + 1 =$$
$$= |\mathtt{search}(\mathtt{n},\mathtt{xs1},\mathtt{m})|.$$

Moreover, any goal $\leftarrow \mathtt{search}(\mathtt{n},\mathtt{xs},\mathtt{M})$ is bounded, and hence every LD-derivation starting from such goal is finite.

3 Semi-acceptable programs

The proof method for left termination presented in the previous section suffers from two drawbacks.

- The level mapping used in the proof of acceptability is sometimes different from the expected natural candidate. Consider for instance the program SEARCH. The relation **search** is defined by induction on the length of its second argument, which is a list, and therefore a natural candidate for $|\mathtt{search(n,xs,m)}|$ is $len(\mathtt{xs})$. Nevertheless, it is needed to multiply by 2 and add 1 to such a value in order to enforce a strict decreasing from the relation **search** to the relations **divide** and **switch**, as required by the definition of acceptability.
- The proposed proof method does not provide means for constructing modular proofs, hence no straightforward technique is available to combine proofs for separate programs in order to obtain proofs of combined programs. In the SEARCH example, abstracting over the termination proof for the program DIVIDE is, strictly speaking, improper, and it is not a trivial task to find a program which is acceptable w.r.t. the given level mapping.

As the module hierarchy of a program becomes more complex, the lack in modularity and the needed adjustments of natural level mappings become more artificial and consequently more difficult to discover and to handle. These drawbacks are (partially) overcome by the concept of semi-acceptability. We first introduce some useful definitions from [AP94]:

Definition 3.1 Let P be a program, Π_P the set of relations defined by P, p and q relations in Π_P:

- we say that p *refers to* q in P if there is a clause in P that uses p in its head and q in its body;
- se say that p *depends on* q in P, and write $p \sqsupseteq q$, if (p,q) is in the reflexive, transitive closure of the relation *refers to*;
- we write $p \simeq q$ iff $p \sqsupseteq q \ \wedge\ q \sqsupseteq p$;

- we write $p \sqsupset q$ iff $p \sqsupseteq q \ \wedge\ q \not\sqsupseteq p$. □

The relation $\sqsupset$ allows us to distinguish non-recursive calls from recursive ones. The notion of acceptability can be relaxed by requiring a strict decreasing of the level mapping only on recursive calls.

Definition 3.2 Let P be a program, $|.|$ a level mapping and I_P an interpretation of P; then:

- A clause is called *semi-acceptable* with respect to $|.|$ and I_P, iff I_P is its model and for every its ground instance $A \leftarrow As, B, Bs$:

$$I_P \models As \Rightarrow \begin{cases} |A| > |B| & \text{if } rel(A) \simeq rel(B) \\ |A| \geq |B| & \text{if } rel(A) \sqsupset rel(B) \end{cases}$$

- A program is called *semi-acceptable* with respect to a level mapping and an interpretation iff every its clause is. A program is called *semi-acceptable* if it is semi-acceptable with respect to some level mapping and interpretation. □

Indeed, the notions of acceptability and semi-acceptability are equivalent.

Theorem 3.3 *Let P be a program, $|.|$ a level mapping, I_P an interpretation of P and G a goal; thus:*

- *P is semi-acceptable iff it is acceptable;*
- *if P is semi-acceptable and G is bounded w.r.t. $|.|$ and I_P, then P is acceptable and G bounded w.r.t. a level mapping $\|.\|$ and I_P.*

Proof. See [AP94]. □

The new definitions improve the modularity of the method; to this purpose we introduce the following definition:

Definition 3.4 Let P and Q be programs:

- A relation is *defined* in a program P if it occurs in the head of a clause of P;
- *P extends Q* if no relation defined in P occurs in Q. □

Basically, P extends Q means that P uses Q as a subprogram. The following result provides a method for conducting modular termination proofs.

Theorem 3.5 *Let P and Q be programs such that P extends Q. Suppose that:*

- *Q is semi-acceptable w.r.t. $|.|_Q$ and $I_P \cap B_Q$;*
- *P is semi-acceptable w.r.t. $|.|_P$ and I_P;*

- *there exists a level mapping* $\|.\|_P$ *such that for every ground instance* $A \leftarrow As, B, Bs$ *of every clause of* P*:*

$$I_P \models As \Rightarrow \begin{cases} \|A\|_P \geq \|B\|_P & \text{if } rel(B) \text{ is defined in } P \\ \|A\|_P \geq |B|_Q & \text{if } rel(B) \text{ is defined in } Q \end{cases}$$

then $P \cup Q$ *is semi-acceptable w.r.t.* $|.|$ *and* I_P*, where* $|.|$ *is defined as follows:*

$$|A| = \begin{cases} |A|_P + \|A\|_P & \text{if } rel(A) \text{ is defined in } P \\ |A|_Q & \text{if } rel(A) \text{ is defined in } Q \end{cases}$$

Proof. See [AP94]. □

3.1 Examples

Maximum of a list of numbers

Consider again the program LISTMAX and the following level mapping:

$$\begin{array}{lcl} |\mathtt{sum(x,y,z)}| & = & val(\mathtt{z}) \\ |\mathtt{gte(x,y)}| & = & val(\mathtt{x}) \\ |\mathtt{selmax(x,y,z)}| & = & max([\mathtt{x},\mathtt{y}])) \\ |\mathtt{listmax(xs,xmax)}| & = & len(\mathtt{xs}) + max(\mathtt{xs}) \end{array}$$

LISTMAX is semi-acceptable w.r.t. the interpretation $I_{LISTMAX}$ as in Section 2.1 and $|.|$; in fact:

$$\begin{array}{l} |\mathtt{sum(x,s(y),s(z))}| = val(\mathtt{z}) + 1 > val(\mathtt{z}) = |\mathtt{sum(x,y,z)}| \\ |\mathtt{gte(x,y)}| = val(\mathtt{x}) = |\mathtt{sum(z,y,x)}| \\ |\mathtt{selmax(x,y,x)}| = max(val(\mathtt{x}), val(\mathtt{y})) \geq val(\mathtt{x}) = |\mathtt{gte(x,y)}| \\ |\mathtt{selmax(x,y,y)}| = max(val(\mathtt{x}), val(\mathtt{y})) \geq val(\mathtt{y}) = |\mathtt{gte(y,x)}| \\ |\mathtt{listmax([x|xs],xmax)}| = len([\mathtt{x}|\mathtt{xs}]) + max([\mathtt{x}|\mathtt{xs}]) > \\ > len(\mathtt{xs}) + max(\mathtt{xs}) = |\mathtt{listmax(xs,xmax1)}| \\ I_{LISTMAX} \models \mathtt{listmax(xs,xmax1)} \Rightarrow \\ |\mathtt{listmax([x|xs],xmax)}| = len([\mathtt{x}|\mathtt{xs}]) + max([\mathtt{x}|\mathtt{xs}]) \geq \\ \geq max(\mathtt{x},\mathtt{xmax1}) = |\mathtt{selmax(x,xmax1,xmax)}|. \end{array}$$

Observe that the proof method based on semi-acceptability allows us to construct a simpler termination proof for LISTMAX, which uses more natural level mappings.

Powers

Consider again the program POWER. We define a level mapping $|.|_{POWER}$ as follows:

$$\begin{array}{lcl} |\mathtt{power(x,y,z)}|_{POWER} & = & y \\ |\mathtt{times(x,y,z)}|_{POWER} & = & 0. \end{array}$$

The clauses defining the predicate power are semi-acceptable w.r.t. the Herbrand interpretation I_{POWER} as in Section 2.1 and $|.|_{POWER}$:

$$\begin{array}{l} |\mathtt{power(x,s(y),z}|_{POWER} = y + 1 > y = |\mathtt{power(x,y,z}|_{POWER} \\ |\mathtt{power(x,s(y),z}|_{POWER} = y + 1 > 0 = |\mathtt{times(x,w,z}|_{POWER} \end{array}$$

We define a level mapping $|.|_{TIMES}$ as follows:

$$\begin{array}{lcl} |\mathtt{times(x,y,z)}|_{TIMES} & = & y \\ |\mathtt{sum(x,y,z)}|_{TIMES} & = & 0 \end{array}$$

The clauses defining the predicate times are semi-acceptable w.r.t. I_{POWER} and $|.|_{TIMES}$:

$$\begin{array}{l} |\mathtt{times(x,s(y),z)}|_{TIMES} = y + 1 > y = |\mathtt{times(x,y,z)}|_{TIMES} \\ |\mathtt{times(x,s(y),z)}|_{TIMES} = y + 1 > 0 = |\mathtt{sum(x,w,z)}|_{TIMES}. \end{array}$$

Finally, we define a level mapping $|.|_{SUM}$ as follows:

$$|\mathtt{sum(x,y,z)}|_{SUM} = y$$

The clauses defining the predicate sum are semi-acceptable w.r.t. I_{POWER} and $|.|_{SUM}$, as in Section 2.1.

In order to prove the left termination of POWER we now use (two times) Theorem 3.5. As first step we define the level mapping $\|.\|_{TIMES}$ as follows:

$$\|\mathtt{times(x,y,z)}\|_{TIMES} = xy$$

$\|.\|_{TIMES}$ fulfills the requirements of Theorem 3.5; in fact:

$$\begin{array}{l} \|\mathtt{times(x,s(y),z)}\|_{TIMES} = x(y+1) \geq xy = \|\mathtt{times(x,y,w)}\|_{TIMES} \\ I_{POWER} \models \mathtt{times(x,y,w)} \Rightarrow \\ \|\mathtt{times(x,s(y),z)}\|_{TIMES} = x(y+1) \geq xy = |\mathtt{sum(x,w,z)}|_{SUM}. \end{array}$$

Therefore, the program consisting of the clauses defining the predicates times and sum is semi-acceptable w.r.t. I_{POWER} and $|.|^*_{TIMES}$ defined as follows:

$$\begin{array}{lcl} |\mathtt{times(x,y,z)}|^*_{TIMES} & = & y + xy = (x+1)y \\ |\mathtt{sum(x,y,z)}|^*_{TIMES} & = & y. \end{array}$$

As a second step, we define a level mapping $\|.\|_{POWER}$ as follows:

$$\|\mathtt{times(x,y,z)}\|_{TIMES} = (x+1)^y$$

$\|.\|_{POWER}$ fulfills the requirements of Theorem 3.5; in fact:

$$\begin{array}{l} \|\mathtt{power(x,s(y),z)}\|_{POWER} = (x+1)^{y+1} \geq (x+1)^y = \|\mathtt{power(x,y,w)}\|_{POWER} \\ I_{POWER} \models \mathtt{power(x,y,w)} \Rightarrow \\ \|\mathtt{power(x,s(y),z)}\|_{POWER} = (x+1)^{y+1} \geq (x+1)x^y = |\mathtt{times(x,w,z)}|^*_{TIMES} \end{array}$$

Therefore, the program POWER is semi-acceptable w.r.t. I_{POWER} and $|.|^*_{POWER}$ defined as follows:

$$\begin{aligned} |\texttt{power(x,y,z)}|^*_{POWER} &= (x+1)^y + y \\ |\texttt{times(x,y,z)}|^*_{POWER} &= (x+1)y \\ |\texttt{sum(x,y,z)}|^*_{POWER} &= y. \end{aligned}$$

Binary search

Consider again the program SEARCH and the following level mapping:

$$\begin{aligned} |\texttt{search(x,xs,m)}| &= 2\ len(\texttt{xs}) \\ |\texttt{divide(xs,xs1,x,y,xs2)}| &= len(\texttt{xs}) \\ |\texttt{switch(x,x,y,xs1,xs2,m)}| &= 2\ (len(\texttt{xs1}) + len(\texttt{xs2})) + 1 \\ |\texttt{x} > \texttt{y}| = |\texttt{x} < \texttt{y}| &= 0 \end{aligned}$$

SEARCH is semi-acceptable w.r.t. the interpretation I_{SEARCH} as in Section 2.1 and $|.|$, provided that the program DIVIDE is; in fact:

$|\texttt{search(n,xs,m)}| = 2\ len(\texttt{xs}) \geq len(\texttt{xs}) = |\texttt{divide(xs,xs1,x,y,xs2)}|$
$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Rightarrow$
$|\texttt{search(n,xs,m)}| = 2\ len(\texttt{xs}) > 2\ (len(\texttt{xs1}) + len(\texttt{xs2})) + 1 =$
$= |\texttt{switch(n,x,y,xs1,xs2,m)}|$
$|\texttt{switch(n,x,y,xs1,xs2,m)}| = 2\ (len(\texttt{xs1}) + len(\texttt{xs2})) + 1 > 0 = |\texttt{n} > \texttt{x}| =$
$= |\texttt{n} < \texttt{x}|$
$|\texttt{switch(n,x,y,xs1,xs2,m)}| = 2\ (len(\texttt{xs1}) + len(\texttt{xs2})) + 1 > 2\ len(\texttt{xs2}) =$
$= |\texttt{search(n,xs2,m)}|$
$|\texttt{switch(n,x,y,xs1,xs2,m)}| = 2\ (len(\texttt{xs1}) + len(\texttt{xs2})) + 1 > 2\ len(\texttt{xs1}) =$
$= |\texttt{search(n,xs1,m)}|.$

4 Cyclically and acyclically acceptable programs

Our LISTMAX example, and a variety of examples in [AP94] point out that the concept of semi-acceptability enhances the modularity of the proposed method and, in most cases, it allows using natural level mappings. Unfortunately, this is not the case in the POWER and SEARCH examples, which still need complex level mappings.

In the case of program POWER, the problem is due the fact that the notion of semi-acceptability is not fully modular, since the condition on the "auxiliary" level mapping $\|.\|_P$ is not "local" to the clauses connecting P with Q (see Theorem 3.5). To find such a local condition we would like to drop the $\|A\|_P \geq \|B\|_P$ line, but this is incorrect. A local condition is needed, to ensure that the resolvent of any bounded goal in an LD-derivation is still bounded (see Lemma 4.3). This can be achieved by requiring a functional dependency between $\|A\|_P$ and $|A|_P$, i.e. $\|A\|_P = \Psi(|A|_P)$.

In the case of program SEARCH, the problem arises from the presence of mutual recursion between the predicates search and switch. Notice that none of the examples in [AP94] is mutually recursive. Definition 3.2 requires that the level mapping

decreases at each step of the LD-resolution, while it is sufficient that the level mapping decreases at each "cycle".

We propose a method which overcomes these problems, by building upon the notion of semi-acceptability. The following definition of a *cycle* will be useful later.

Definition 4.1 A *cycle* of a program P is an ordered sequence of n (with $n > 0$) clauses of P:

$$C = \{A_i \leftarrow As_i, B_i, Bs_i\} \qquad (i \in [1,n])$$

such that:

$$\begin{aligned} &rel(A_i) \neq rel(A_j) \quad &&if\ i \neq j \\ &rel(B_i) = rel(A_{i+1}) \quad &&if\ i \in [1, n-1] \\ &rel(B_n) = rel(A_1). \end{aligned}$$

The *order* of a cycle is the number n of clauses belonging to it. □

We now introduce the notions of *cyclically* and *acyclically acceptable* programs. The concept of cyclic acceptability generalizes the requirement of strict decreasing of a level mapping in the case of mutually recursive predicates. The concept of acyclic acceptability generalizes the requirement of non-strict decreasing of a level mapping.

Definition 4.2 Let P be a program, $|.|$ a level mapping and I_P an interpretation of P:

- A cycle of order n of P is *acceptable* w.r.t. $|.|$ and I_P iff I_P is its model and for every its ground instance $\{A_i \leftarrow As_i, B_i, Bs_i\}$ [1]:

$$\left[\begin{array}{ll} \forall\ i \in [1,n]: & I_P \models As_i \\ \forall\ i \in [1,n-1]: & B_i = A_{i+1} \end{array} \quad \wedge \right] \Rightarrow |A_1| > |B_n|$$

  A program is *cyclically acceptable* w.r.t. $|.|$ and I_P iff at least one cyclic permutation of all its cycles is acceptable w.r.t. $|.|$ and I_P. A program is *cyclically acceptable* iff it is cyclically acceptable w.r.t. some level mapping and interpretation.

- A clause is *acyclically acceptable* w.r.t. $|.|$ and I_P iff I_P is its model and exists a total function $\Psi : \omega \rightarrow \omega$, such that, for every ground instance $A \leftarrow As, B, Bs$ of the clause:

$$I_P \models As \Rightarrow \Psi(|A|) \geq |B|$$

  A program is *acyclically acceptable* w.r.t. $|.|$ and I_P iff every its clause is; a program is *acyclically acceptable* iff it is acyclically acceptable w.r.t. some level mapping and interpretation. □

[1] The scope of each variable name is always only the containing clause, so variables belonging to different clauses are always to be considered different, even if they share the same name, thus can be bounded to different terms.

Some results concerning cyclic and acyclic acceptability follows. The next Lemma points out the key property of acyclic acceptability, namely that the persistence of boundedness during the process of LD-resolution. In other words, the LD-resolvent of a bounded goal w.r.t. a acyclically acceptable program is in turn bounded.

Lemma 4.3 *Let P be a program, acyclically acceptable w.r.t. I_P and $|.|$, and G, G' be goals such that G' is a LD-resolvent of G in P. Then $|G| < \infty \Rightarrow |G'| < \infty$.* □

As a consequence of Lemma 4.3, we obtain that, for a program which is both cyclically and acyclically acceptable, all LD-derivations starting from a bounded goal are finite.

Theorem 4.4 *Let P be a program, cyclically and acyclically acceptable w.r.t. $|.|$ and I_P, and G a goal, bounded w.r.t. $|.|$ and I_P. Then every LD-derivation of $P \cup \{G\}$ is finite.* □

Finally, the following result points out the equivalence of the new notions with those of acceptability and semi-acceptability. Cyclic and acyclic acceptability, therefore, yield a sound and complete method for proving left termination.

Theorem 4.5 *Let P be a program; then the following characterizations are equivalent:*

(i) P is left terminating,

(ii) P is acceptable,

(iii) P is semi-acceptable,

(iv) P is cyclically and acyclically acceptable. □

4.1 Examples

We now show how the proof method based on the notions of cyclic and acyclic acceptability improves on the method in [AP94], by supporting a higher degree of modularity and allowing to use more natural level mappings. In particular, the new method fixes the cited problems in the POWER and SEARCH examples.

Powers

Consider again the program POWER and the level mapping $|.|$ defined as follows:

$$\begin{array}{lcl} |\texttt{power(x,y,z)}| & = & x + y \\ |\texttt{times(x,y,z)}| & = & x + y \\ |\texttt{sum(x,y,z)}| & = & y \end{array}$$

POWER is cyclically acceptable w.r.t. the interpretation I_{POWER} as in Section 2.1; in fact, the only cycles of the program are the recursive clauses, and therefore:

$$\begin{array}{l} |\texttt{power(x,s(y),z)}| = x + y + 1 > x + y = |\texttt{power(x,y,w)}| \\ |\texttt{times(x,s(y),z)}| = x + y + 1 > x + y = |\texttt{times(x,y,w)}| \\ |\texttt{sum(x,s(y),z)}| = y + 1 > y = |\texttt{sum(x,y,z)}| \,. \end{array}$$

Moreover POWER is acyclically acceptable w.r.t. I_{POWER} and $|.|$; in fact, using the auxiliary functions:

$$\begin{array}{lcl} \Psi_1(x) & = & x^x \\ \Psi_2(x) & = & x^2 \end{array}$$

we obtain:

$$\begin{array}{l} I_{POWER} \models \texttt{power(x,y,w)} \Rightarrow \\ \Psi_1(|\texttt{power(x,s(y),z)}|) = (x + y + 1)^{x+y+1} \geq x^y + x = |\texttt{times(x,w,z)}| \\ I_{POWER} \models \texttt{times(x,y,w)} \Rightarrow \\ \Psi_2(|\texttt{times(x,s(y),z)}|) = (x + y + 1)^2 > xy = |\texttt{sum(x,w,z)}| \end{array}$$

As a consequence POWER is left terminating. The proof structure is now more modular; moreover, the cost of introducing the "auxiliary" functions is balanced by the simplicity of the level mapping, which is now more natural.

Binary search

Consider again the program SEARCH and the level mapping $|.|$:

$$\begin{array}{lcl} |\texttt{search(x,xs,m)}| & = & \mathit{len}(\texttt{xs}) \\ |\texttt{divide(xs,xs1,x,y,xs2)}| & = & \mathit{len}(\texttt{xs}) \\ |\texttt{switch(x,x,y,xs1,xs2,m)}| & = & \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \\ |\texttt{x < y}| = |\texttt{x > y}| & = & 0 \end{array}$$

SEARCH is cyclically acceptable w.r.t. the interpretation I_{SEARCH} as in Section 2.1 and $|.|$, provided that the program DIVIDE is. To prove this fact, observe that the program contains only the following two cycles (and their cyclic permutation):

```
search(N,Xs,M) ← divide(Xs,Xs1,X,Y,Xs2), switch(N,X,Y,Xs1,Xs2,M)
switch(N,X,Y,Xs1,Xs2,M) ← N>X, search(N,Xs2,M)
```

and:

```
search(N,Xs,M) ← divide(Xs,Xs1,X,Y,Xs2), switch(N,X,Y,Xs1,Xs2,M)
switch(N,X,Y,Xs1,Xs2,M) ← N<X, search(N,Xs1,M)
```

The above cycles are acceptable w.r.t. I_{SEARCH} and $|.|$:

$$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Rightarrow$$
$$|\texttt{search(n,xs,m)}| =$$
$$= \mathit{len}(\texttt{xs}) > \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \geq \mathit{len}(\texttt{xs2}) =$$
$$= |\texttt{search(n,xs2,m)}|$$
$$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Rightarrow$$
$$|\texttt{search(n,xs,m)}| =$$
$$= \mathit{len}(\texttt{xs}) > \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \geq \mathit{len}(\texttt{xs1}) =$$
$$= |\texttt{search(n,xs1,m)}|$$

Moreover SEARCH is acyclically acceptable w.r.t. I_{SEARCH} and $|.|$; in fact, taking Ψ as the identity function we obtain:

$$|\texttt{search(n,xs,m)}| = \mathit{len}(\texttt{xs}) = |\texttt{divide(xs,xs1,x,y,xs2)}|$$
$$I_{SEARCH} \models \texttt{divide(xs,xs1,x,y,xs2)} \Rightarrow$$
$$|\texttt{search(n,xs,m)}| =$$
$$= \mathit{len}(\texttt{xs}) > \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) = |\texttt{switch(n,x,y,xs1,xs2,m)}|$$
$$|\texttt{switch(n,x,y,xs1,xs2,m)}| = \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \geq 0 =$$
$$= |\texttt{n} < \texttt{x}| = |\texttt{n} > \texttt{x}|$$
$$|\texttt{switch(n,x,y,xs1,xs2,m)}| = \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \geq \mathit{len}(\texttt{xs1}) =$$
$$= |\texttt{search(n,xs1,m)}|$$
$$|\texttt{switch(n,x,y,xs1,xs2,m)}| = \mathit{len}(\texttt{xs1}) + \mathit{len}(\texttt{xs2}) \geq \mathit{len}(\texttt{xs2}) =$$
$$= |\texttt{search(n,xs2,m)}|$$

As a consequence, SEARCH is left terminating. Observe that the adopted level mapping is now more natural than in the case of acceptability and semi-acceptability.

5 Conclusions

The paper presented a refinement of the method of Apt and Pedreschi for proving termination of Prolog program, which yields simpler and more modular proofs. The examples showed how a better proof structure is achieved, and more natural level mappings are adopted. Both improvements account for a larger amenability of the method to be partially supported by automatic tools. Indeed, the development of tools of this kind, capable of assisting the users in conducting termination proofs, is an important direction for future research. Automatic methods for termination proofs, however, form a whole stream of research (see the survey paper of De Schreye and Decorte [DSD93] for references.)

Moreover, further improvements of the method are still possible. First, the examples (in particular the program POWER) indicates that the property of acyclic acceptability should be extracted from the model used in the termination proof, by exploiting the interargument relationships defined by the model itself. Second, a simpler method should not adopt the Herbrand base, which is not a modular concept, but rather the subset of the Herbrand base which is relevant to the program under consideration.

References

[AP90] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In J.W. Lloyd, editor, *Symposium on Computional Logic*, pages 150–176, Berlin, 1990. Springer-Verlag.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[AP94] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Proceedings of the Fourth International School for Computer Science Researchers*. Oxford University Press, 1994. to appear.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[Bez89] M. Bezem. Characterizing termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80. The MIT Press, 1989.

[Bez93] M.A. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–98, 1993.

[DSD93] D. De Schreye and S. Decorte. Termination of Logic Programs: the Never–Ending Story. *Journal of Logic Programming*. Submitted, 1993.

[DSVB92] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 481–488. Institute for New Generation Computer Technology, 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.