Automated Modular Termination Proofs for Real Prolog Programs

Martin Müller
Thomas Glaß
Karl Stroetmann

Siemens AG
ZFE T SE 1
D–81730 München, Germany

{Martin.Mueller,Thomas.Glass,Karl.Stroetmann}@zfe.siemens.de
phone: +49 89 636 41687
fax: +49 89 636 42284

Abstract. We present a methodology for checking the termination of Prolog programs that can be automated and is scalable. Furthermore, the proposed method can be used to locate errors. It has been successfully implemented as part of a tool that uses static analysis based on formal methods in order to validate Prolog programs. This tool is aimed at supporting the design and maintenance of Prolog programs.

Our approach is based on a natural extension of the notion of *acceptable* programs developed in Apt and Pedreschi [AP90, AP93]. The main idea is to assign a measure of complexity to predicate calls. Then termination of a program is shown by proving this measure to be decreasing on recursive calls. While this measure is a natural number in [AP90, AP93], we extend this idea by using *tuples* of natural numbers as a measure of complexity. These tuples are then compared lexicographicly. The use of this kind of measure enables us to refine the notion of *acceptable* programs to the notion of *loop free* programs. This notion can be used to modularize the termination proof of Prolog programs to a greater extend than previously possible.

Keywords: Prolog, termination, formal methods, static analysis, automated theorem proving

1 Introduction

As software gets more and more complex, the problem of guaranteeing the reliability of a software system is getting increasingly difficult. Formal methods can make a valuable contribution towards solving this problem. However, at least for the time being, a complete specification and verification of most software systems is economically not viable since the process of verification can, in general, not be automated. Nevertheless, recent progress in automated deduction has shown that for declarative programming languages it is possible to verify certain kinds of properties automatically.

Of course, the question whether a program is terminating is, in general, undecidable. However, the paper of Apt and Pedreschi [AP90, AP93] gives a useful characterization of terminating programs: A program is terminating iff there exists a *norm* $|.|$ and an *interpretation* I such that the program is *acceptable* with respect to $|.|$ and I. We refine this notion into the notion of a program being *loop free* with respect to $|.|$ and I. This notion has been inspired by the work of De Schreye, Verschaetse and Bruynooghe [SVB92]. The property of a program being loop free with respect to a fixed norm $|.|$ and interpretation I becomes decidable if we restrict ourselves to the use of *linear norms*. Moreover, our practical experiences have shown that the notion of *loop freeness* is sufficiently strong to deal with most programs encountered in practice.

On a theoretical level, our work is a refinement of the ideas found in Apt and Pedreschi [AP93] and a variation of the approach suggested in Gröger and Plümer [GP92]. The question of the modularization of termination proofs is the main theme of [AP93]. Unfortunately, this solution is not entirely satisfactory: If two programs P and Q are given such that P *extends* Q, i.e. predicates from Q are used in P but not vice versa, then the termination proof for P can not be given independently from the termination proof for Q. Furthermore, [AP93] does not discuss the question of automating the termination proofs in sufficient detail.

The paper of Gröger and Plümer [GP92] is concerned with the question of automating the termination check. Although the methods presented in [GP92] are successful to prove the termination of programs, they seem to be not as well suited to locate errors leading to non-termination. The reason is that the approach of Gröger and Plümer does not require to specify either *inter-argument relations* or *norm definitions* but rather aims at computing this information automatically. This causes their method to fail for some programs that can be dealt with by our method. Furthermore, it prevents them from locating errors. However, in practice the ability to locate errors is one of the features that distinguishes a useful tool.

We present a method that supports modular termination proofs, is capable of being fully automated and, furthermore, can be used to locate errors in programs. It has been successfully implemented as part of a tool that is aimed at supporting the development and maintenance of software written in Prolog. Methodologically our approach is a natural extension of the notion of *acceptable* programs developed in [AP90, AP93]. The main idea there is to assign a measure of complexity to atomic formulas. The termination of a program is then shown by proving this measure to be decreasing on recursive calls. While this measure is a natural number in [AP90, AP93], we extend this idea by using *tuples* of natural numbers as a measure of complexity. These tuples are then compared lexicographicly. The first component of these tuples is the *level* of a predicate as defined in Clark [Cla78], while the second component is determined by the size of the arguments. The fact that the first component is independent of the arguments but only reflects the recursion structure of the program enables us to come up with a truly modular termination criterion.

The next section introduces some basic notions and notations. Section 3 introduces modes and types which are a prerequisite for our method. In Section 4 we define the central notion of the *norm* of an atomic formula. This norm is the measure of complexity which has been cited above. We refine the notion of Apt and Pedreschi of an *acceptable program* to the notion of a program being *loop free* in Section 5. This notion is the basis for the modular termination criterion given in Theorem 19 of Section 6. The question how this criterion can be automated is then tackled in Section 7, while the last section discusses an implementation of the method introduced in this paper.

2 Some Basic Notions

In the following we use $t, t_1, \ldots$ to denote terms, $p, q, p_1, \ldots$ to denote predicate symbols and $P, Q, R, P_1, \ldots$ to denote atomic formulas. If $P_1, \ldots, P_n, P$ are atomic formulas, then $(P_1, \ldots, P_n)$ is a query and P :- $P_1, \ldots, P_n$ is a clause. We use $\mathcal{Q}, \mathcal{R}, \ldots$ to denote queries and C to denote clauses. Given a fixed Prolog program we write

$$(P_1, \ldots, P_n) \rightarrow^{\mu}_{\text{res}} (Q_1\mu, \ldots, Q_m\mu, P_2\mu, \ldots, P_n\mu)$$

if there is a variant of a clause P :- $Q_1, \ldots, Q_m$ and a most general unifier μ of P and P_1. This is called an LD-resolution step. We write $\mathcal{Q} \rightarrow^{*\mu}_{\text{res}} \mathcal{Q}'$ as an abbreviation for $\mathcal{Q} \rightarrow^{\mu_1}_{\text{res}} \mathcal{Q}_1 \rightarrow^{\mu_2}_{\text{res}} \ldots \rightarrow^{\mu_n}_{\text{res}} \mathcal{Q}'$ and $\mu = \mu_n \circ \ldots \circ \mu_1$. We call μ a *solution* of $\mathcal{Q}$ if $\mathcal{Q} \rightarrow^{*\mu}_{\text{res}} ()$. The *search tree* belonging to a query $\mathcal{Q}$ has $\mathcal{Q}$ as its root node and a for every node $\mathcal{Q}'$ appearing in this tree we have that $\mathcal{Q}''$ is a child of $\mathcal{Q}'$ iff $\mathcal{Q}' \rightarrow^{\mu}_{\text{res}} \mathcal{Q}''$. The set of solutions of a given query which is found by a certain search strategy – we think of Prologs 'depth first' search strategy – may be a real subset of the set of all solutions of this query if the search tree is infinite.

We call a query $\mathcal{Q}$ *terminating* w.r.t. a Prolog program $\mathbf{P}$ if there is no infinite sequence $\mathcal{Q} = \mathcal{Q}_1 \rightarrow^{\mu_1}_{\text{res}} \mathcal{Q}_2 \rightarrow^{\mu_2}_{\text{res}} \mathcal{Q}_3 \rightarrow^{\mu_3}_{\text{res}} \ldots$ In this case the search tree is finite (by Königs Lemma, as a Prolog program has only a finite number of clauses) and all solutions are found by Prologs search strategy.

Referring to a given Prolog program we define the following relations between predicate symbols:

Definition 1. Let p, q be predicate symbols.

1. We define $p \rightarrow_{C,i} q$ if *p calls q in clause C at position i* that is C has the form $p(\ldots)$:- $q_1(\ldots), \ldots, q_n(\ldots)$ and $q = q_i$. We write $p \rightarrow q$ if $p \rightarrow_{C,i} q$ for some C and i.
2. We define $\sqsupseteq$ to be the reflexive and transitive closure of $\rightarrow$. We say *p depends on q* if $p \sqsupseteq q$.
3. We define p and q to be *mutually recursive* or *in the same recursive clique* and write $p \simeq q$ if $p \sqsupseteq q \wedge q \sqsupseteq p$.
4. We define $p \sqsupset q$ if *p calls q as a subprogram* that is $p \sqsupseteq q \wedge \neg q \sqsupseteq p$.

Observe that $\sqsupseteq$ is a preorder, $\simeq$ is a equivalence relation and $\sqsupset$ is a well-founded order, since there is only a finite number of predicate symbols in a Prolog program.

Definition 2. We assign a natural number to every predicate symbol p by defining $\text{level}(p) := \max\{\text{level}(q) + 1 : p \sqsupset q\}$, $(\max \emptyset = 0)$.

Observe that level is well-defined, as $\sqsupset$ is well-founded. This is in essence the level mapping given in Lloyd [Llo87, p. 83]. We extend these notions to atomic formulas in the obvious way, $\text{level}(p(\ldots)) = \text{level}(p)$, $p(\ldots) \simeq q(\ldots)$ iff $p \simeq q$, etc.

Lemma 3. *1. If $p \sqsupset q$ then level$(p) >$ level(q)*
2. If $p \simeq q$ then level$(p) =$ level(q) ■

Definition 4. The *call graph of a Prolog program* is the hypergraph which has the predicate symbols as vertices and edges $p \rightarrow_{C,i} q$.

We have $p \simeq q$ iff there is a cycle in the call graph containing both p and q.

We assume the reader to be familiar with:

- *wellorderings*: A linear ordering $>$ is called wellordering, if there is no infinite sequence $a_1 > a_2 > \ldots$).
- *lexicographic ordering*: if $<_1$ is a wellordering on a set M_1 and $<_2$ is a wellordering on a set M_2, then $>_{\text{lex}}$ is a wellordering on the set $M_1 \times M_2$, where $\langle a, b\rangle >_{\text{lex}} \langle c, d\rangle$ is defined by $a >_1 c$ or $a = c, b >_2 d$.
- *multiset ordering*: If $>$ is a wellordering on M, then $>_{\text{mul}}$ is a wellordering on the set of finite multisets of M, where $M >_{\text{mul}} N$, iff N can be obtained from M by iteratively replacing an element of M by a finite number of smaller elements.
- *ordinal numbers*: A canonical wellordering, every wellordering is isomorphic to an initial segment of ordinal numbers.

For ordinal numbers cf. e.g. Pohlers [Poh89], for multiset and lexicographic orderings cf. e.g. Bachmair [Bac91] or Avenhaus [Ave95].)

3 Typed Prolog

In this section we briefly explain "well-typedness". For that purpose we first explain what types are and then define well-typed queries and well-typed programs, respectively.

Definition 5. A type is a set of terms that is closed under substitution, i.e. if τ is a type and $t \in \tau$, then for all substitutions μ we have $t\mu \in \tau$.

Example 1. A recursive definition of the form

```
list := [] + '.'(term, list)
```

denotes that the type `list` consists of the constant `[]` and all terms of the form `'.'(T,L)` where `T` is of type `term`, i.e. is an arbitrary term, and `L` is of type `list`. Obviously `list` and `term` are types, i.e. they are closed under substitution. Similarly, the type `nat` which is defined by

```
nat := 0 + s(nat)
```

denotes the set of terms consisting of the constant `0` and all terms of the form `s(N)` where `N` is of type `nat`.

Now we define predicate signatures: we specify the input and the output types of a predicate by a signature. In general we write

$$p(\sigma_1 \rightarrow \tau_1, \ldots, \sigma_n \rightarrow \tau_n) \tag{1}$$

for the predicate p/n with *input types* $\sigma_1, \ldots, \sigma_n$ and *output types* $\tau_1, \ldots, \tau_n$, respectively. We abbreviate $\sigma \rightarrow \sigma$ by $+\sigma$ and `term` $\rightarrow \tau$ by $-\tau$.

Example 2. A signature for `append/3` is given by:

```
append(+list,+list,-list)
```

which means that whenever `append`(r, s, t) is called with r, s of type `list` and returns a solution μ, we have that $r\mu$,$s\mu$ and $t\mu$ are of type `list`. This is formalized as follows (suppressing argument lists):

Definition 6. Let $p_0, \ldots, p_n$ be predicates with signatures $p_i(\sigma_i \rightarrow \tau_i)$ for $i \leq n$ be given. We call the clause $p_0(s_0)$`:-` $p_1(s_1), \ldots, p_n(s_n)$ *well-typed* iff the following holds for all substitutions μ:

1. For all $i = 1, \ldots, n$ the following holds: If $s_0\mu$ is of type σ_0 and $s_1\mu, \ldots, s_{i-1}\mu$ are of type $\tau_1, \ldots, \tau_{i-1}$, respectively, then $s_i\mu$ is of type σ_i.
2. If $s_0\mu$ is of type σ_0 and $s_1\mu, \ldots, s_n\mu$ are of type $\tau_1, \ldots, \tau_n$, respectively, then $s_0\mu$ is of type τ_0.

A program is called *well-typed* iff all clauses are. For the rest of this paper we will assume that all programs are well-typed.

We call a query $(p_1(s_1), \ldots, p_m(s_m))$ *well-typed* iff for all substitutions μ and all $i = 1, \ldots, m$ such that $s_1\mu, \ldots, s_{i-1}\mu$ are of type $\tau_1, \ldots, \tau_{i-1}$ we have that s_i is of type σ_i.

A literal $p(s_1, \ldots, s_n)$ is called *well-typed* w.r.t. the signature (1) iff $s_1, \ldots, s_n$ are of type $\sigma_1, \ldots, \sigma_n$, respectively.

Example 3. The usual implementation of `append/3` is well-typed:

```
append([], L, L).
append([H|T], L, [H|TL]) :- append(T, L, TL).
```

The literal `append([1],[2],L)` is well-typed, whereas `append([1],2,L)` is not.

The property of a query being well-typed is invariant under resolution and substitution. For a proof compare Lemma 3.8 in [AM94].

4 Norms on Prolog Terms and Atomic Formulas

To prove termination we need a mapping from terms, atomic formulas and queries into a well-founded ordering. We use *norms* to assign natural numbers to terms. This mapping is then extended to atomic formulas.

Definition 7. A *norm* on terms is a mapping $|.|$: ground terms $\rightarrow \mathbb{N}$. A *linear norm* is a norm which can be defined by assigning to every n-ary function symbol f some natural numbers $a_1, \ldots, a_n, a \in \mathbb{N}$ and defining inductively on ground terms

$$|f(t_1, \ldots, t_n)| := a_1 \, |t_1| + \ldots + a_n \, |t_n| + a.$$

We call a term t *rigid* w.r.t. a norm $|.|$ if $|t_1| = |t_2|$ for all ground instances t_1, t_2 of t. We extend the norm $|.|$ to rigid terms t by defining $|t| = |t'|$ where t' is a ground instance of t. We call a type τ *rigid* w.r.t. a norm $|.|$ if every $t \in \tau$ is rigid w.r.t. $|.|$.

Example 4. A linear norm which assigns to every list its length is given by $|\texttt{[]}| = 0, |\texttt{[}A\,\texttt{|}\,L\texttt{]}| = |L| + 1$. The type `list` (cf. Example 1) is rigid w.r.t. this norm.

Definition 8. A *predicate norm* is a mapping $|.|$: atomic formulas $\rightarrow \mathsf{On}$ from atomic formulas to ordinal numbers such that $|P'| \leq |P|$ for every instance P' of P.

In a characterization of terminating programs (i.e. equivalence of termination and acceptability, cf. Apt and Pedreschi [AP90, AP93]) arbitrary predicate norms have to be taken into account. To get manageable norms (i.e. acceptability becomes decidable) we use *linear predicate norms* $|.|$: ground atomic formulas $\rightarrow$

$\mathbb{N}^K$,where $K \in \mathbb{N}$ and $\mathbb{N}^K$ is ordered lexicographically. These norms are defined in the following way using a linear norm on ground terms:

$$|p(t_1,\ldots,t_n)| := \langle \textstyle\sum_{j=1}^n a_{1,j}\;|t_j| + b_1, \ldots, \sum_{j=1}^n a_{K,j}\;|t_j| + b_K \rangle \tag{2}$$

where $a_{i,j}, b_i$ are natural numbers depending on p. If $K = 1$ we will omit the brackets and write s instead of $\langle s \rangle$.

The lexicographically ordered set $\mathbb{N}^K$ is isomorphic to the set of ordinal numbers up to ω^K. The isomorphism is $\langle a_K, \ldots, a_1 \rangle = \omega^{K-1} a_K + \ldots + \omega^0 a_1$. We can lift the predicate norm (2) to arbitrary atomic formulas by defining

$$|P| := \sup\{|P'| : \ P' \text{ is a ground instance of } P\ \} \tag{3}$$

Notice that we always have $|P| \leq \omega^K$ and (3) defines a predicate norm in the sense of Definition 8.

Definition 9. We call an atomic formula P *rigid* w.r.t. a predicate norm $|.|$ if $|P| = |P'|$ for every ground instance P' of P. We call a predicate norm *rigid* (w.r.t. a signature) if every well-typed atomic formula P is rigid w.r.t. the predicate norm.

A linear predicate norm defined as in (3) is rigid if for every predicate symbol p with signature $p(\sigma_1 \to \tau_1, \ldots, \sigma_n \to \tau_n)$ and every argument position $j \in \{1, \ldots, n\}$ either τ_j is rigid w.r.t. the term norm or $a_{i,j} = 0$ for all $i \in \{1, \ldots, K\}$.

Example 5. If we have the signature `append( + list, + list, - list )` and the predicate norm definition `|append( L1, L2, L )| = |L1|` where the norm of a list is its length (cf. Examples 2,4), then this predicate norm is rigid.

5 Acceptable Prolog Programs

In this section we extend the concept of *acceptable* Prolog programs from Apt and Pedreschi [AP93] by the use of ordinal numbers. Our predicate norm is a generalization of the level mapping given in [Bez89, Bez93, AP90, AP93]. The use of ordinal numbers will allow us to prove that every *loop free* program (c.f. Definition 17) is *acceptable* (c.f. Definition 10). This can not be done using the original definition of acceptability given in [AP90, AP93] where the level mapping only has natural numbers as value. Within this section let **P** be a well-typed Prolog program, I a (not necessarily Herbrand) interpretation of **P** and $|.|$ a rigid predicate norm.

Definition 10 (Apt,Pedreschi). A clause C of **P** is called *acceptable w.r.t.* $|.|$ *and* I if I is a model of C, $|.|$ is rigid, and $|P| > |Q_i|$ for every ground instance P:- $Q_1, \ldots, Q_n$ of C and every i such that $I \models Q_1 \wedge \ldots \wedge Q_{i-1}$.

A program **P** is called *acceptable w.r.t.* $|.|$ *and* I if all its clauses are. **P** is called *acceptable* if it is acceptable w.r.t. some predicate norm and some interpretation of **P**.

Example 6. The sorting algorithm in Figure 1 with signatures and type definitions

```
nat := 0 + s(nat)
nat_list := [] + '.'( nat, nat_list )
sort( + nat_list, -nat_list )
minlist( + nat_list, -nat )
min( + nat, + nat, - nat )
delete( + nat, + nat_list, - nat_list )
le( + nat, + nat )
```

is acceptable with respect to a predicate norm $|.|$ and an interpretation I defined in the following way: $|\mathtt{le}(n,m)| = |n|$, $|a\mathtt{\backslash=}b| = 0$, $|\mathtt{delete}(a,l_1,l_2)| = |l_1|$, $|\mathtt{min}(a,b,m)| = |a| + |b| + 1$, $|\mathtt{minlist}(l,m)| = |l| + 1$, $|\mathtt{sort}(l,s)| = |l| + 2$, where the norm on terms is defined by $|\mathtt{0}| = 0$, $|\mathtt{s}(n)| = |n| + 1$, $|\mathtt{[]}| = 0$, $|\mathtt{[}a\,\mathtt{|}\,l\mathtt{]}| = |a| + |l| + 1$. Notice that in this example the norm of a list is not its length, but the sum of its length and the values of its elements.

$$\begin{array}{rl} I = & \{\mathtt{sort}(l,s) : l, s \text{ ground}\} \\ & \cup\{\mathtt{minlist}(l,m) : l, m \text{ ground, } m \text{ is a member of } l\} \\ & \cup\{\mathtt{min}(a,b,a) : a, b \text{ ground}\} \cup \{\mathtt{min}(a,b,b) : a, b \text{ ground}\} \\ & \cup\{\mathtt{delete}(e,l_1,l_2) : \; e, l_1, l_2 \text{ ground, } |l_2| \leq |l_1|, \\ & \qquad\qquad\qquad\qquad |l_2| < |l_1| \text{ or } e \text{ is not a member of } l_1 \;\} \\ & \cup\{\mathtt{le}(a,b) : a, b \text{ are ground}\} \end{array}$$

Definitions of this kind are needed to establish acceptability in the sense of Apt and Pedreschi [AP90, AP93], however these definitions are a bit overloaded for the purpose of proving termination. Therefore we will introduce the notion of *loop freeness* in the next section. We will exemplarily show that the conditions of Definition 10 are satisfied for the second clause. Obviously I is a model of the second clause, because its head is true in I. Let

$$\mathtt{sort}(l,\mathtt{[}m\,\mathtt{|}\,s\mathtt{]})\mathtt{:-}\;\; l\mathtt{\backslash=[]},\mathtt{minlist}(l,m),\mathtt{delete}(m,l,l_2),\mathtt{sort}(l_2,s)$$

be a ground instance of the second clause. Then we trivially have $|l\mathtt{\backslash=[]}| < |\mathtt{sort}(l,\mathtt{[}m\,\mathtt{|}\,s\mathtt{]})|$, $|\mathtt{minlist}(l,m)| < |\mathtt{sort}(l,\mathtt{[}m\,\mathtt{|}\,s\mathtt{]})|$ and $|\mathtt{delete}(m,l,l_2)| < |\mathtt{sort}(l,\mathtt{[}m\,\mathtt{|}\,s\mathtt{]})|$. If $I \models l\mathtt{\backslash=[]} \wedge \mathtt{minlist}(l,m) \wedge \mathtt{delete}(m,l,l_2)$, that is m is a member of l and $|l_2| < |l|$, then $|\mathtt{sort}(l_2,s)| < |\mathtt{sort}(l,\mathtt{[}m\,\mathtt{|}\,s\mathtt{]})|$.

Example 7. The Ackermann function (cf. Figure 2) is acceptable w.r.t. the trivial interpretation $I = \{\mathtt{ack}(t_1,t_2,t_3) : \; t_1,t_2,t_3 \text{ ground}\}$ and the predicate norm $|\mathtt{ack}(n,m,a)| = \langle |n|, |m| \rangle$, where the norm on terms is defined by $|\mathtt{0}| = 0$ and $|\mathtt{s}(n)| = |n| + 1$. To verify this you only have to observe that $\langle |n|, |\mathtt{s(0)}| \rangle < \langle |\mathtt{s}(n)|, |\mathtt{0}| \rangle$, $\langle |\mathtt{s}(n)|, |m| \rangle < \langle |\mathtt{s}(n)|, |\mathtt{s}(m)| \rangle$ and $\langle |n|, |a| \rangle < \langle |\mathtt{s}(n)|, |\mathtt{s}(m)| \rangle$ for all terms m, n, a.

```
sort( [], [] ).
sort( L, [ Min | S ] ) :-
     L \= [],
     minlist( L, Min ),
     delete( Min, L, L2 ),
     sort( L2, S ).

minlist( [E], E ).
minlist( [ E | L ], Min ) :-
        L \= [],
        minlist( L, M1 ),
        min( E, M1, Min ).

min( A, B, A ) :- le(A, B).
min( A, B, B ) :- A \= B, le( B, A ).

delete( _E , [], [] ).
delete( E, [ E | L ], D ) :- delete( E, L, D ).
delete( E, [ F | L ], [ F | D ] ) :-
        E \= F,
        delete( E, L, D ).

le( 0, _ ).
le( s(X), s(Y) ) :- le( X, Y ).
```

Fig. 1. sorting

```
ack( 0, Y, s(Y) ).
ack( s(X), 0, Z ) :- ack( X, s(0), Z ).
ack( s(X), s(Y), Z ) :- ack( s(X), Y, A ), ack( X, A, Z ).
```

Fig. 2. Ackermann Function

Our aim is to show that LD-resolution terminates for every well-typed query if the Prolog program is acceptable. Modifying the notions from [AP93] we obtain the following definition.

Definition 11. With every query $Q = (P_1, \ldots, P_n)$ we associate n sets of ordinal numbers defined for $i \in \{1, \ldots, n\}$ as follows:

$$|Q|_i^I := \{|P_i'| + 1 : \ (P_1', \ldots, P_n') \text{ is a ground instance of } Q \\ \text{and } I \models P_1' \wedge \ldots \wedge P_{i-1}'\}$$

We define a multiset $|\mathcal{Q}|^I$ of ordinal numbers as follows

$$|\mathcal{Q}|^I := \{\sup|\mathcal{Q}|^I_1, \ldots, \sup|\mathcal{Q}|^I_n\}_{\text{mul}}.$$

Lemma 12. *If $\mathcal{Q}$ is a query and $\mathcal{Q}'$ is an instance of $\mathcal{Q}$, then we have*

$$|\mathcal{Q}|^I \geq_{\text{mul}} |\mathcal{Q}'|^I.$$

Proof. For every i we have $|\mathcal{Q}'|^I_i \subseteq |\mathcal{Q}|^I_i$. ■

Lemma 13. *If P:- $Q_1, \ldots, Q_n$ is an instance of some clause which is acceptable w.r.t. $|.|$ and I, furthermore $\mathcal{R} = R_1, \ldots, R_m$ are atomic formulas such that $(P, \mathcal{R})$ is a well-typed query, then we have*

$$|(P, \mathcal{R})|^I >_{\text{mul}} |(Q_1, \ldots, Q_n, \mathcal{R})|^I.$$

Proof. First we have for every $i \in \{1, \ldots, n\}$

$$\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i < \sup|(P, \mathcal{R})|^I_1. \tag{4}$$

This follows from the following observations:

1. $|(P, \mathcal{R})|^I_1 = \{|P| + 1\} = \{|P'| + 1\}$ for every ground instance P' of P as P is a well-typed literal by Definition 6 and $|.|$ is rigid.
2. $\sup|(P, \mathcal{R})|^I_1 = |P| + 1$ is not a limit ordinal.
3. For every $\alpha \in |(Q_1, \ldots, Q_n, \mathcal{R})|^I_i$ we have $\alpha < |P| + 1$, as we know : $\alpha = |Q'_i| + 1$ for some ground instance $(Q'_1, \ldots, Q'_n)$ of $(Q_1, \ldots, Q_n)$ which satisfies $I \models Q'_1 \wedge \ldots \wedge Q'_{i-1}$ (Definition 11). So for a ground instance P':- $Q'_1, \ldots, Q'_n$ of the acceptable clause P:- $Q_1, \ldots, Q_n$ we have by Definition 10 $|Q'_i| < |P'|$. We conclude $|Q'_i| + 1 < |P| + 1$ using 1.
4. If $\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i$ is not a limit ordinal, then there is a maximal element $\alpha \in |(Q_1, \ldots, Q_n, \mathcal{R})|^I_i$ and we have $\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i = \alpha < |P| + 1$ using 3.
5. If $\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i$ is a limit ordinal, then $\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i \leq |P| + 1$ holds by 3. and $\sup|(Q_1, \ldots, Q_n, \mathcal{R})|^I_i \neq |P| + 1$ by 2.

From 4. and 5. we conclude (4). Further we have for every $j \in \{1, \ldots, m\}$

$$\sup|(Q_1, \ldots, Q_n, R_1, \ldots, R_m)|^I_{j+n} \leq \sup|(P, R_1, \ldots, R_m)|^I_{j+1} \tag{5}$$

since by Definition 11

$$\begin{aligned}
&|(Q_1, \ldots, Q_n, R_1, \ldots, R_m)|^I_{j+n} \\
&= \{|R'_j| + 1 : (Q'_1, \ldots, Q'_n, R'_1, \ldots, R'_m) \text{ is a ground instance of} \\
&\qquad\qquad (Q_1, \ldots, Q_n, \mathcal{R}) \text{ and } I \models Q'_1 \wedge \ldots \wedge Q'_n \wedge R'_1 \wedge \ldots \wedge R'_{j-1}\} \\
&\quad [\text{as for some } P',\ P'\text{:- } Q'_1, \ldots, Q'_n \text{ is an instance of } P\text{:- } Q_1, \ldots, Q_n \\
&\quad \text{and } I \text{ is a model of } \mathbf{P}\ ] \\
&\subseteq \{|R'_j| + 1 : (P', R'_1, \ldots, R'_m) \text{ is a ground instance of} \\
&\qquad\qquad (P, R_1, \ldots, R_m) \text{ and } I \models P' \wedge R'_1 \wedge \ldots \wedge R'_{j-1}\} \\
&= |(P, R_1, \ldots, R_m)|^I_{j+1}
\end{aligned}$$

The claim follows by definition of the multiset ordering from (4) and (5). ■

Lemma 14. *Let* **P** *be a well-typed program that is acceptable w.r.t. a predicate norm* $|.|$ *and an interpretation* I*. Let* Q_1 *be a well-typed query and* Q_2 *a LD-resolvent of* Q_1*. Then we have* $|Q_1|^I >_{\text{mul}} |Q_2|^I$*.*

Proof. Let $Q_1 = (P_1, \ldots, P_n)$. There is a variant of a clause P :- $Q_1, \ldots, Q_m$ such that P_1 and P are unifiable with most general unifier μ and we have $Q_2 = (Q_1\mu, \ldots, Q_m\mu, P_2\mu, \ldots, P_n\mu)$. By Lemma 12 we have $|(P_1, \ldots, P_n)|^I \geq_{\text{mul}} |(P_1\mu, \ldots, P_n\mu)|^I$ and since an instance of a well-typed query is well-typed we know $|(P_1\mu, \ldots, P_n\mu)|^I >_{\text{mul}} |(Q_1\mu, \ldots, Q_m\mu, P_2\mu, \ldots, P_n\mu)|^I$ using Lemma 13. ■

Because the multiset extension of a well-founded order is well-founded and well-typedness is invariant under resolution we have proven:

Theorem 15. *If* **P** *is an acceptable and well-typed program and* Q *is a well-typed query, then* Q *is terminating with respect to* **P**. ■

6 A Modular Termination Criterion

In this section we are going to explain our termination criterion and prove its correctness. In contrast to the notion of acceptability, we only want to compare predicate norms of literals which are in the same recursive clique in order to obtain termination. We will show how from a predicate norm satisfying *loop freeness*,which is a much weaker condition than acceptability (Definition 10), a predicate norm can be constructed which establishes acceptability.

We divide the edges in the call graph into three classes:

Definition 16. Let **P** be a Prolog program, $|.|$ be a predicate norm and I a model of **P**.

- $p \rightarrow_{C,i} q$ is a *non-recursive call* if $p \sqsupset q$.
- $p \rightarrow_{C,i} q$ is a *descending recursive call w.r.t.* $|.|$ *and* I if $p \simeq q$ and $|P| \geq |Q_i|$ for every ground instance P :- $Q_1, \ldots, Q_n$ of the clause C such that $I \models Q_1 \wedge \ldots \wedge Q_{i-1}$.
- If $p \simeq q$ and $p \rightarrow_{C,i} q$ is not descending, then we call $p \rightarrow_{C,i} q$ a *recursive call which violates the termination conditions.*

Descending recursive calls are split into two subclasses:

- $p \rightarrow_{C,i} q$ is a *strongly descending recursive call w.r.t.* $|.|$ *and* I if $p \simeq q$ and $|P| > |Q_i|$ for every ground instance P :- $Q_1, \ldots, Q_n$ of the clause C such that $I \models Q_1 \wedge \ldots \wedge Q_{i-1}$.
- $p \rightarrow_{C,i} q$ is a *weakly descending recursive call w.r.t.* $|.|$ *and* I if it is descending but not strongly descending.

Definition 17. A program is called *loop free w.r.t.* $|.|$ *and* I if I is a model of **P**, every edge $p \rightarrow_{C,i} q$ such that $p \simeq q$ is descending w.r.t. $|.|$ and I, and every cycle in the call graph of the program contains at least one strongly descending edge $p \rightarrow_{C,i} q$. It is called *loop free* if it is loop free w.r.t. some predicate norm and some interpretation.

Note that every edge which appears in a cycle of the call graph of a loop free program is descending, as a cycle cannot leave a mutual recursive clique.

The concept of loop freeness has two advantages if it is compared with the concept of acceptability: First modular termination proofs become possible since predicate norms of atomic formulas have to be compared only if their predicate symbols are in the same recursive clique, no artificial offsets are needed in the definition of the predicate norm. Secondly mutual recursion can be handled with simpler norms, as every cycle in the call graph has to contain only one strongly descending edge (compare to the parser in Example 4.5 in De Schreye et al. [SVB92]).

Example 8. We can show the loop freeness of the sorting algorithm given in Figure 1 with less effort than its acceptability (cf. Example 6). Let $|.|$ be a norm on terms which assigns to every list its length (cf. Example 4) and to every natural number its value.

Define $|\texttt{le}(n,m)| = |n|$, $|a\texttt{\textbackslash=}b| = 0$, $|\texttt{delete}(a,l_1,l_2)| = |l_1|$, $|\texttt{min}(a,b,m)| = 0$, $|\texttt{minlist}(l,m)| = |l|$, $|\texttt{sort}(l,s)| = |l|$ and define I as in Example 6. Then this program is loop free w.r.t. the predicate norm $|.|$ and the interpretation I.

This can be shown with less effort than acceptability in Example 6. Let $\texttt{sort}(l,\texttt{[}m\texttt{|}s\texttt{]})\texttt{:-}\ l\texttt{\textbackslash=[]},\texttt{minlist}(l,m),\texttt{delete}(m,l,l_2),\texttt{sort}(l_2,s)$ be a ground instance of the second clause. We only have to observe that $|\texttt{sort}(l_2,s)| < |\texttt{sort}(l,\texttt{[}m\texttt{|}s\texttt{]})|$ if $I \models l\texttt{\textbackslash=[]} \wedge \texttt{minlist}(l,m) \wedge \texttt{delete}(m,l,l_2)$.

Definition 18. If **P** is loop free w.r.t. $|.|$ and I, then we assign to every predicate symbol p a natural number layer(p), defined as the longest path in the call graph of **P**, which starts from p and contains only weakly descending edges. In other words

$$\begin{aligned}\text{layer}(p) = \max\{m :\ & \text{There is a path } p = p_0 \rightarrow_{C_1,i_1} \ldots \rightarrow_{C_m,i_m} p_m \\ & \text{such that every } p_{j-1} \rightarrow_{C_j,i_j} p_j \text{ is weakly descending}\}\end{aligned}$$

The function layer is extended to atomic formulas in the obvious way, that is layer($p(\ldots)$) = layer(p).

Notice that layer is well-defined, since every cycle contains at least one strongly descending edge and the length of cycle free paths is limited by the number of predicate symbols.

Theorem 19. *If a program* **P** *is loop free w.r.t.* $|.|$ *and* I*, then there is a predicate norm* $|.|'$ *such that* **P** *is acceptable w.r.t.* $|.|'$ *and* I*.*

Proof. Define $|P|' := \langle \text{level}(P), |P|, \text{layer}(P)\rangle$ to be a mapping into $\mathbb{N} \times \alpha \times \mathbb{N}$ ordered lexicographicly, where $\alpha \subset \mathsf{On}$ is a set of ordinal number such that $|P| \in \alpha$ for every atomic formula P. Since $\mathbb{N} \times \alpha \times \mathbb{N}$ is a well-founded linear order it is order isomorphic to some initial segment of the ordinal numbers. Therefore we can identify $|P|'$ with an ordinal number. We claim that **P** is acceptable w.r.t. $|.|'$ and I. By assumption I is a model of P. Let $P\texttt{:-}\ Q_1,\ldots,Q_n$

be a ground instance of a clause C and choose any $i \in \{1, \ldots, n\}$ such that $I \models Q_1 \wedge \ldots \wedge Q_{i-1}$. We have to show $|P|' > |Q_i|'$. We distinguish the following cases:

1. If $P \sqsupset Q_i$, then we have level(P) > level(Q_i) and we are done.
2. If $P \simeq Q_i$ and $P \rightarrow_{C,i} Q_i$ is strongly descending, then we have level(P) = level(Q_i) by Lemma 3 and $|Q_i| < |P|$ by Definition 16.
3. $P \simeq Q_i$ and $P \rightarrow_{C,i} Q_i$ is weakly descending. Therefore we have level(P) = level(Q_i) by Lemma 3, $|P| \geq |Q_i|$ by Definition 16 and layer(P) > layer(Q_i) by Definition 18. ■

As an immediate consequence of Theorems 15 and 19 we obtain the following theorem.

Theorem 20. *If* **P** *is a loop free and well-typed program and Q is a well-typed query, then Q is terminating with respect to* **P**. ■

7 Linear Inter-Argument Relations

The question whether a given program is loop free w.r.t. a predicate norm $|.|$ and an interpretation I is decidable, if we restrict norms to linear predicate norms and interpretations to interpretations defined by *linear inter-argument relations*. A linear inter-argument relation for an n-ary predicate symbol p is an inequation

$$a_1 |X_1| + \ldots + a_n |X_n| + b \geq 0$$

where a_i, b are integers. Given a set of linear inter-argument relations for every predicate symbol p, an interpretation I can be defined by $I \models p(t_1, \ldots, t_n)$ iff the instance $a_1 |t_1| + \ldots + a_n |t_n| + b \geq 0$ of every every inter-argument relation for p is satisfied. Linear norms can be partially evaluated, that is for every term t and atomic formula P, respectively, we can find integers such that

$$|t\mu| = c_1 |X_1\mu| + \ldots + c_m |X_m\mu| + d$$
$$|P\mu| = \langle \sum c_{1,j} |X_j\mu| + d_1, \ldots, \sum c_{K,j} |X_j\mu| + d_K \rangle$$

where the X_i are the variables occurring in t and P, respectively. Therefore in the case of linear norms and an interpretation I defined by linear inter-argument relations the problem of being loop free w.r.t. I and $|.|$ can be reduced to the question whether a formula in Presburger arithmetic (cf. [End72, Pre29]) is true. In our implementation we used Bledsoe's Sup-Inf-method (cf. [Ble75]) for solving Presburger formulas.

For some examples like the sorting algorithm in Figure 1 (cf. Example 8) we need *conditional linear inter-argument relations*. These take the form

$$a_1 |X_1| + \ldots + a_n |X_n| + b \geq 0 \text{ if } P$$

where a_i, b are integers and P is an atomic formula such that only the variables $X_1, \ldots, X_n$ occur in P. The predicate symbol of P has to be defined explicitly, i.e. it must have a fixed interpretation identical to its interpretation in the least Herbrand model of the program. An interpretation I satisfies a conditional inter-argument relation if $I \models p(t_1, \ldots, t_n)$ implies

$$a_1 \, |t_1| + \ldots + a_n \, |t_n| + b \geq 0 \text{ or } \mathcal{H} \models \neg P[X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n]$$

where $\mathcal{H}$ is the least Herbrand model of the program.

In the case of conditional inter-argument relations we have no decision procedure for loop freeness, but we claim that in most cases loop freeness can be established by an automatic theorem prover from the conditional linear inter-argument relations and explicit definitions of the predicates used in the conditions. The treatment of conditional inter-argument relations was inspired by the work of Walther [Wal90].

8 Implementing Termination Checking

Automated termination checking is implemented within the framework of a general static analysis tool for Prolog programs, cf. Stroetmann, Glaß [SG95] and Stroetmann, Glaß, Müller [SGM96]. Naturally in practice we have to deal with certain problems which have not been described in the theoretical part, e.g. negative literals, predefined predicates, etc. Below we describe briefly some of the details which have to be dealt with to obtain a useful tool for automated termination checking of real Prolog programs.

8.1 Implementing Type Checking

The design of a type language has to respect the following conditions: it should be possible to define types such that a well-typed program (in the sense of Definition 6) is type safe, i.e. there should be no run-time type errors in the sense of [DEDC96]. Further, a type checking algorithm should catch as many programming mistakes as possible, therefore the usual data structures of a Prolog program have to be supported. In contrast to these conditions, nearly no program should have to be rewritten just to make it well-typed.

The language design is inspired by Hill and Topor [HT92], Yardeni, Frühwirth and Shapiro [YFS92] and Bronsard, Lakshmann and Reddy [BLR92]. We start from a finite set of so-called basic types (in the sense of [YFS92, 2.4]) which are necessary to describe the signatures of all ISO-Prolog predicates correctly, cf. [DEDC96], as e.g. the types `atom`, `float`, `int`. Then we define new types out of these by recursive equations (type rules in [HT92, 1.5], type definitions in [YFS92, 2.1]) similar to Example 1.

The main features of the implemented type system are:

- parametric polymorphism, i.e. the use of type parameters
- inclusion polymorphism, i.e. the use of a subtype relation
- support of strong typing, i.e. types are distinguished by names
- support of negation as failure

- restricted support of second-order predicates like `findall/3`, `bagof/3`
- restricted support of database manipulating predicates like `assert/1` and `retract/1`

In fact the main problem of type checking is to combine the first two aspects.

8.2 Automated Termination Checking

This subsection explains how the procedure of termination checking works in practice. We will start with an untyped program, cf. Figure 3, and augment it with information for the analysis tool.

```
minus(X, 0, X).
minus(X, s(Y), Z) :-  minus(X, Y, s(Z)).

lt(0, s(Y)).
lt(s(X), s(Y)) :- lt(X, Y).

le(0, Y).
le(s(X), s(Y)) :- le(X, Y).

mod(X, Y, X):- Y \= 0, lt(X, Y).
mod(X, Y, Z):- Y \= 0, le(Y, X), minus(X, Y, W), mod(W, Y, Z).
```

Fig. 3. modulo

To add type information we define the type `nat` of Example 1 and add signatures for all predicates to the source code. Since the code should run on an arbitrary Prolog system, we start all lines containing analysis information with the pragma `%#`, cf. Figure 4.

```
%# nat := 0 + s(nat).

%# predicate: minus(+nat, +nat, -nat).
%# predicate: lt(+nat, +nat).
%# predicate: le(+nat, +nat).
%# predicate: mod(+nat, +nat, -nat).
```

Fig. 4. adding type information

We continue to explain the steps to augment the program by annotations for proving termination: the norm $|.|$ on the type `nat` given by the equations: $|\texttt{0}| = 0, |\texttt{s(N)}| = |\texttt{N}| + 1$ is assumed by our system if there is not explicitly given a different definition of the norm.

Termination of `minus/3`, `lt/2`, `le/2` can be proved by observing that these predicates are defined via simple recursions on the second arguments. This is expressed via linear predicate norms:

```
%# pnorm: |minus(X, Y, Z)| = |Y|.
%# pnorm: |lt(X, Y)|       = |Y|.
%# pnorm: |le(X,Y)|        = |Y|.
```

Proving loop freeness for these predicates automatically does not make use of inter-argument relations, i.e. they are loop free w.r.t. the trivial interpretation

$$\begin{aligned} I = {} & \{\texttt{minus}(x, y, z) : x, y, z \text{ ground}\} \cup \{\texttt{lt}(x, y) : x, y \text{ ground}\} \cup \\ & \{\texttt{le}(x, y) : x, y \text{ ground}\} \cup \{\texttt{mod}(x, y, z) : x, y, z \text{ ground}\}. \end{aligned}$$

More involved is proving loop freeness for `mod/3`. The first clause of `mod/3` is obviously loop free since `mod/3` calls `\=` and `lt` as subprograms. The second clause

```
mod(X, Y, Z):- Y \= 0, le(Y, X), minus(X, Y, W), mod(W, Y, Z).
```

contains a recursive call of `mod/3`. We know that $|\texttt{W}|$ is less than $|\texttt{X}|$ in the recursive call. Thus we choose the linear norm for `mod/3` as follows:

```
%# pnorm: |mod(X, Y, Z)| = |X|.
```

But, in fact, the program is not loop free w.r.t. to the trivial interpretation I. Termination for `mod/3` depends on conditional linear inter-argument relations for `minus/3` which are annotated as follows:

```
%# dependency(minus(X, Y, Z)): |Z| =< |X|, |Z| < |X| if  Y \= 0.
```

It can be shown automatically that there is an interpretation I which satisfies this inter-argument relation, such that the program is loop free w.r.t. I and $|.|$.

8.3 Error Detection

One further benefit of the method of adding annotations to program code is that this allows a simple kind of error location. Assume that we have coded `mod/3` by the clauses:

```
mod(X, Y, X):- Y \= 0, lt(X, Y).
mod(X, Y, Z):- le(Y, X), minus(X, Y, W), mod(W, Y, Z).
```

This means that we have forgotten the test `Y \= 0` in the second clause. Trying to prove termination automatically, will end up with the following message (pointing to the recursive call of `mod/3` in the second clause):

```
>>> One of the following conditions has to be valid:
>>> le(Y,X) -> Y \= 0
```

The second clause is loop free if we can find an interpretation which satisfies this implication in addition to the program clauses and conditional inter-argument relations. Obviously this is impossible since every model of the program satisfies `le(0,0)`. Actually the query `mod(s(0),0,Z)` will cause the program to loop.

9 Conclusion: Practical Experiences

We have added annotations for the termination check to all clauses of our tool for static analysis of Prolog programs (22k lines of Prolog code with 14.1% annotations including type and signature definitions and including 3.9% annotations for the termination check, i.e. norm definitions and inter-argument relations).

Run time with IF/Prolog 5.0 on a Sun SPARCclassic with 48 MB main memory for the termination check (without parsing and type checking) is: 6.67 sec. cpu time for an average file containing 1k lines of annotated Prolog code, whereas type checking takes 9.93 sec.

References

[AM94] Krzysztof R. Apt and Elena Marchiori. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.

[AP90] Krzysztof R. Apt and Dino Pedreschi. Studies in pure Prolog: Termination. In John W. Lloyd, editor, *Symposium on Computational Logic*, pages 150–176. Springer-Verlag, 1990.

[AP93] Krysztof R. Apt and Dino Pedreschi. Modular termination proofs for logic and pure prolog programs. In G. Levi, editor, *Proceedings of Fourth International School for Computer Science Researchers*. Oxford University Press, 1993. Available as technical report CS-R9316 at `http://www.cwi.nl/cwi/publications/index.html`.

[Apt92] Krzysztof R. Apt, editor. *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, D. C., USA, November, 9–13 1992. The MIT Press.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme – Rechnen und Schließen in gleichungsdefinierten Strukturen*. Springer-Verlag, 1995.

[Bac91] Leo Bachmair. *Canonical Equational Proofs*. Birkhäuser Boston, Inc., Boston, MA, 1991.

[Bez89] Marc Bezem. Characterizing termination of logic programs with level mappings. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference*, volume 1, pages 69–80, Cleveland, Ohio, USA, October 16–20, 1989. The MIT Press, Cambridge, Massachusetts.

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

[Ble75] W. W. Bledsoe. A new method for proving Presburger formulas. In *4th International Joint Conference on Artificial Intelligence*, September 1975. Tibilisi, Georgia, U.S.S.R.

[BLR92] François Bronsard, T. K. Lakshman, and Uday S. Reddy. A framework of directionality for proving termination of logic programs. In Apt [Apt92], pages 321–335.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[DEDC96] P. Deransart, A. A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, 1996.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[GP92] Gerhard Gröger and Lutz Plümer. Handling of mutual recursion in automatic termination proofs for logic programs. In Apt [Apt92], pages 336–350.

[HT92] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In Pfenning [Pfe92], pages 1–62.

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

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming.* MIT, Cambridge, Mass./London, 1992.

[Poh89] Wolfram Pohlers. *Proof Theory. An Introduction.* Number 1407 in Lecture Notes in Mathematics. Springer-Verlag, Berlin/Heidelberg/New York, 1989.

[Pre29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Sprawozdanie z I Kongrescu Matematykow Krajow Slowkanskich Warszawa*, pages 92 – 101, 1929.

[SG95] Karl Stroetmann and Thomas Glaß. Augmented PROLOG — An evolutionary approach. In Donald A. Smith, Olivier Ridoux, and Peter Van Roy, editors, Proceedings of the Workshop *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor to Prolog*, pages 59–70, 1995. The Proceedings of this workshop are available at: `ftp://ps-ftp.dfki.uni-sb.de/pub/ILPS95-FutureLP/`.

[SGM96] Karl Stroetmann, Thomas Glaß, and Martin Müller. Implementing safety–critical systems in PROLOG: Experiences from the R & D at SIEMENS. In Peter Reintjes, editor, *Practical Applications of Prolog '96*, pages 391–403. The Practical Application Company, 1996.

[SVB92] D. De Schreye, Kristof Verschaetse, and Maurice Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 481–488. ICOT, 1992.

[Wal90] Christoph Walther. *Automatisierung von Terminierungsbeweisen.* Vieweg, Braunschweig, Germany, 1990.

[YFS92] Eyal Yardeni, Thom Frühwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Pfenning [Pfe92], pages 63–90.

This article was processed using the LaTeX macro package with LLNCS style