HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **18**; October 1998

ISSN 0783–540X
ISBN 951–22–4358–X

IMPLEMENTATION OF LOCAL GROUNDING FOR LOGIC PROGRAMS WITH STABLE MODEL SEMANTICS

Tommi Syrjänen

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **18**; October 1998

ISSN 0783–540X
ISBN 951–22–4358–X

Implementation of Local Grounding for Logic Programs with Stable Model Semantics

Tommi Syrjänen
Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Abstract: The aim of this work is to define a new class of logic programs, strongly range restricted logic programs, and to design and implement an efficient algorithm for grounding them locally. A grounding transforms a function-free logic program into an equivalent ground logic program. A grounding is local if can be done one rule at a time. The implementation works with **smodels**, an implementation for stable model semantics for normal logic programs which is developed at Helsinki University of Technology. The implementation is compared with old grounder of **smodels** and `dlv`, a knowledge representation system developed at Vienna University of Technology.

Printing: Libella Painopalvelut Oy 1998

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

List of Figures

1 Introduction

The logic programming originated in 1970's when Van Emden and Kowalski [14] presented the procedural semantics of logic programs. Their intuition was to view a program rule as a procedure declaration and each literal in the rule body as a subroutine call. The first and still the most widely used logic programming language Prolog implements the procedural semantics using SLDNF-resolution (Linear resolution with Selection function for Definite programs with Negation as finite Failure). However, SLDNF-resolution is not complete for even function-free programs because the query evaluator may be caught in infinite positive loops.

Several alternative declarative semantics have been proposed to overcome the problems with Prolog, the most notable being stable model semantics [3] and well-founded semantics [15]. The well-founded semantics has been implemented in XSB logic programming system using SLG-resolution[1, 4].

The finding of stable models is quite distinct from query evaluation of Prolog. The query evaluation tries to prove a specific goal in top-down fashion stopping when the desired proof is found, and the complete model is not computed. In case of stable models the whole model has to be generated at one time and query processing transforms to checking if there are stable models where the queried atom is true. In fact the stable model semantics may be considered as a form of constraint programming as the rules of a program can be seen as constraints to the models of the program.

Intuitively there is a close connection between stable model semantics and autoepistemic logic [7, 6]. A stable model of a program can be seen as "a possible set of beliefs that a rational agent might hold" when its inference rules are the rules of the program. The agent is also capable of doing negative introspection, that is to conclude not (B) if it is not able to conclude B. In this way the stable model semantics is non-monotonic, since given more evidence the agent may be able conclude B and has to retract not (B).

An efficient implementation of the stable model semantics for normal logic programs, **smodels**, has been developed in the Digital Systems Laboratory in Helsinki University of Technology [8, 10, 13]. The system processes programs in two phases. In the first phase the parser reads a function-free range-restricted normal logic program in and generates a variable-free ground logic program out of it. In the second phase the stable models of the ground program are calculated using an efficient linear space algorithm.

However, there is a significant weakness in **smodels**. The grounder, **parse**, keeps the whole ground program in memory during grounding. This is a problem, because the number of ground instances may grow exponentially with respect to number of rules in program. In practice, there are many problems where the grounding of the program takes orders of magnitude more time than the actual finding of stable models.

The aim of this work is to define a class of programs that can be grounded efficiently and to construct a C++-implementation, **lparse** which works with **smodels**.

The basic idea is to define a new class of normal logic programs, *strongly range restricted logic programs*, where every variable in a rule has a completely determined domain at the time of grounding the rule. This can be achieved by defining a class of *domain predicates* that determine the possible values of variables. After the domain predicates are found and grounded, the rest of the rules may be processed one at a time. The ground instances may be emitted immediately and there is no need to store them in memory.

The domain predicates make up the class of predicates that are not defined recursively in terms of each other. The simplest domain predicates are those that are defined in the program as facts, and the more complex ones are defined in terms of simpler domain predicates, using common relational algebra. All ground instances of domain predicates are always true in each stable model of the program.

1.1 Outline of the work

In Section 2 the definitions for stable model semantics are revised. In Section 3 a class of strongly range restricted logic programs is defined. Section 3.2 introduces dependency graphs and Section 3.3 defines a concept of domain predicates. In Section 3.4 the concept of strongly range restricted logic program is formally defined. In Section 3.5 few observations about domain generation are presented. In Section 3.6 a basic algorithm for grounding strongly range restricted programs is presented.

In Section 4 the implementation techniques for **lparse** are presented. The complexity analysis of the implementation is in Section 4.4. In Section 5 the performance of **lparse** is compared with the default grounding algorithm of **smodels** and the `dlv` system, which is a knowledge representation system based on disjunctive logic programming developed at Vienna University of Technology [2]. In Section 6 the conclusions are presented. The formal grammar of **lparse** is in Appendix A.

2 Theoretical background

An *atom* is of the form $p(a_1, \ldots, a_n)$ where p is a n-ary predicate symbol and $a_1, \ldots, a_n \quad (n \geq 0)$ are terms. A *term* may be either a variable, a constant or a function $f(t_1, \ldots, t_n)$. An atom with no variables is called a *ground* atom.

A *literal* L is either an atom A or its negation $\mathrm{not}\,(A)$. Literals of the form $\mathrm{not}\,(A)$ are called *not-atoms* or *negative literals* throughout this work. The set of atoms in a set of literals L is denoted by L^+ and the set of not-atoms is denoted by L^-. For a set of atoms A,

$$\mathrm{not}\,(A) = \{\mathrm{not}\,(a) \quad | \quad a \in A\} \tag{1}$$

A *rule* R is of the form

$$h \leftarrow l_1, \ldots, l_n \tag{2}$$

where the rule *head* h is an atom and $l_1, \ldots, l_n (n \geq 0)$ in the rule *body* are literals. If the rule body is empty $(n = 0)$ the rule is called a *fact*. A rule is ground if it is variable free. An expression is either an atom, a literal or a rule. A general *logic program* is a set of rules.

The *Herbrand universe* $\mathbf{HU}(P)$ of a logic program P is the set of all ground terms that can be constructed using function symbols and constants in P. If P is function-free then the $\mathbf{HU}(P)$ is finite otherwise it is infinite. In this work only function-free programs are considered. The *Herbrand base* $\mathbf{HB}(P)$ of P is the set of all ground atoms that can be constructed using predicates in P and terms in $\mathbf{HU}(P)$.

An *instance* of an expression is constructed by replacing all variables in it by ground terms. The *Herbrand instantiation* $\mathbf{HI}(P)$ is the set of all ground instances of rules in P, which may be constructed using terms in $\mathbf{HU}(P)$.

An *interpretation* $\mathrm{I}(P)$ of a program P is a subset of $\mathbf{HB}(P)$. An interpretation assigns a truth valuation to each element of $\mathbf{HB}(P)$. If an atom a is in $I(P)$, then its valuation is true in I, otherwise it is false. If $\mathrm{I}(P)$ makes every rule in $\mathbf{HI}(P)$ true then $\mathrm{I}(P)$ is a *Herbrand model* of the program. A Herbrand model is *minimal* if no proper subset of it is a Herbrand model. A logic program P without negative literals has a unique minimal Herbrand model [14] which is called the *meaning* of P. The minimal model can be found using operator T_P, where

$$T_P(M) = \{H \quad | \quad H \leftarrow L_1, \ldots, L_n \in P, \{L1, \ldots, L_n\} \in M\}. \tag{3}$$

The minimal model is the least fixed point $\Delta = T_P \uparrow \omega$, where

$$\begin{aligned} &T_P \uparrow 0 = \emptyset \\ &T_P \uparrow \alpha = T_P(T_P \uparrow \alpha - 1) \\ &T_P \uparrow \omega = \bigcup \{T_P \uparrow \alpha \quad | \quad \alpha < \omega\} \end{aligned} \tag{4}$$

The operator T_P was first defined in [14].

2.1 Stable model semantics

The stable model semantics for a ground logic program P is defined as follows [3]. Let M be any set of atoms from P. Then the *reduct* P^M of P with respect to M is obtained by deleting

1. each rule that has a negative literal $\mathrm{not}\,(a)$ in its body with $a \in M$ and
2. all negative literals in the bodies of the remaining rules.

As P^M is negation free, it has a unique minimal Herbrand model. If this model coincides with M, then M is a *stable model* of P. A program may have none, one or many stable models.

Example 1 Let P be the set of rules $\{a \leftarrow \mathrm{not}\, b; b \leftarrow \mathrm{not}\, a\}$. Then P has two stable models, $\{a\}$ and $\{b\}$. This is seen by taking $\{a\}$ as M. Then $P^M = \{a \leftarrow\}$ which has a minimal Herbrand model $\{a\} = M$. The other model is found similarly.

The stable models can be seen as "possible sets of beliefs that a rational agent might hold" if P is given as premises. The intuitive explanation of reduct P^M is that if M is a set of ground atoms that are considered true then any rule that depends on a literal $\mathrm{not}\,(b), b \in M$ cannot be used in deduction and may be discarded. Also, every literal $\mathrm{not}\,(b), b \notin M$ is trivially true and may be discarded. If M is the set of atoms that logically follow from P^M then it is "rational" to believe in M.

The set of stable models for a non-ground logic program P is defined to be the set of stable models of the Herbrand instantiation $\mathbf{HI}(P)$.

The *deductive closure* $\mathrm{Dcl}\,(P, L)$ of a program P and a set of literals L is the minimal model of the rules

$$\begin{aligned} P(L) = \{h \leftarrow a_1, \ldots, a_n \quad | \\ & h \leftarrow a_1, \ldots, a_n, \mathrm{not}\, b_1, \ldots, \mathrm{not}\, b_m \in P, \\ & \{\mathrm{not}\, b_1, \ldots, \mathrm{not}\, b_m\} \subseteq L.\} \end{aligned} \tag{5}$$

It should be noted that for all $L' \subseteq L, \mathrm{Dcl}(P, L') \subseteq \mathrm{Dcl}(P, L)$ and thus $\mathrm{Dcl}(P, \cdot)$ is monotonic. Also, with $P' \subseteq P, \mathrm{Dcl}(P', L) \subseteq \mathrm{Dcl}(P, L)$.

Proposition 1 *The set Δ is a stable model of a set of rules P if and only if $\Delta = \mathrm{Dcl}(P, \mathrm{not}(\bar{\Delta}))$.*

The proposition is proved in [8, 10].

3 Strongly range restricted logic programs

3.1 Grounding

The problem of existence of a stable model for a general logic program is NP-complete [6]. However, there exists an efficient linear space algorithm for calculating stable models for ground logic programs [11, 13].

A *grounding* transforms a logic program P into an equivalent ground logic program P_G where equivalence is defined as having the same set of stable models. A *variable binding* is a substitution that maps a subset of variables of a rule into ground terms.

A naive method of grounding would use full Herbrand instantiation of the program. In practice it is not possible to generate all Herbrand instances of a given logic program as the Herbrand instantiation is often far too big. If the size of the $\mathbf{HU}(P)$ is m, there are k rules, and the maximum arity predicates in a rule is n then the worst case number of the Herbrand instances is km^n.

In most cases many of the rule bodies in $\mathbf{HI}(P)$ are such that their bodies cannot be true in any stable models and they may be discarded without affecting the stable models of P. However, it must be made certain that no stable models are lost when trimming the grounding. This is accomplished by setting some restrictions to variable use in rules.

The grounder of **smodels**, **parse**, works with *range restricted* programs. A rule is range restricted if it holds that if a variable appears in the head or in a negative literal, it also appears in a positive literal in the body. A program is range restricted if all rules in it are range restricted.

The grounding algorithm of **parse** is fully described in [13]. basically it starts from the set of facts in the body and calculates a deductive closure while disregarding the negative literals in the rules. While the algorithm gives quite concise programs, it has one serious problem: the whole closure has to be kept in memory during grounding. High memory usage combined with the cost of finding the deductive closure leads to poor performance in many cases. In Section 5 there are test cases where the grounding of the program takes orders of magnitude more time than the computation of a stable model.

A more efficient way is to do a *local grounding*, that is, ground P one rule at time to obtain ground program P_{LG}. Unfortunately, it is not possible to ground range restricted programs locally, since the whole deductive closure has to be calculated before we know which ground instances are needed. To facilitate local grounding we have to define a new class of logic programs, namely *strongly range restricted programs.*

The basic idea is to divide predicates into two classes, domain and non-domain predicates. The domain of the variables of a given rule is given by the domain predicates in the rule body. The notion is formally defined later in this section. Informally domain predicates those predicates, that are not defined recursively or in terms of recursively defined predicates.

Example 2 Let P be the set of rules

$$\begin{aligned} &\{d_1(1), d_2(2), d_3(3), \\ &a(X) \leftarrow d_1(X), \operatorname{not} b(X), \\ &b(X) \leftarrow d_1(X), \operatorname{not} a(X)\} \end{aligned} \tag{6}$$

Here d_i are domain predicates, a and b are non-domain predicates. The Herbrand instantiation of the program is

$$\begin{aligned} \mathbf{HI}(P) = \{&d_1(1), d_2(2), d_3(3), \\ &a(1) \leftarrow d_1(1), \operatorname{not} b(1), \\ &a(2) \leftarrow d_1(2), \operatorname{not} b(2), \\ &a(3) \leftarrow d_1(3), \operatorname{not} b(3), \\ &b(1) \leftarrow d_1(1), \operatorname{not} a(1), \\ &b(2) \leftarrow d_1(2), \operatorname{not} a(2), \\ &b(3) \leftarrow d_1(3), \operatorname{not} a(3)\} \end{aligned} \tag{7}$$

The Herbrand instantiation has two stable models: $\{d_1(1), d_2(2), d_3(3), a(1)\}$ and $\{d_1(1), d_2(2), d_3(3), b(1)\}$. The atoms $d_1(2)$ and $d_1(3)$ cannot be deduced and thus all rules depending on them don't have any impact on stable models. We may remove those rules and have program

$$\begin{aligned} P_{LG} = \{&d_1(1), d_2(2), d_3(3), \\ &b(1) \leftarrow d_1(1), \operatorname{not} a(1) \\ &a(1) \leftarrow d_1(1), \operatorname{not} b(1)\} \end{aligned} \tag{8}$$

Which has the same stable models as $\mathbf{HI}(P)$.

The size of the local grounding of P may be considerably smaller than $\mathbf{HI}(P)$. If P has k rules, maximum domain size is d and maximum number of variables in a rule is v then in the worst case there are at most kd^v ground instances. As seen above, if the size of the $\mathbf{HU}(P)$ is m, the size of the $\mathbf{HI}(P)$ is at most km^n.

If there are just a few large domains ($d \approx m$) and there are many variables in every rule ($v \approx n$) then the savings are small but if $d \ll m$ the difference may be many orders of magnitude. For example, if $m = 100, d = 10, k = 150, n = 4, v = 4$ then $|\mathbf{HI}(P)| \leq 1.5 \times 10^{10}$ but $|P_{LG}| \leq 1.5 \times 10^6$.

Example 3 The exponential worst case size of the grounding occurs also in practice. Let

$$\begin{aligned} P = \{p(X_1, \ldots, X_n) \leftarrow d(X_1), \ldots, d(X_n) \\ d(a_1); \ldots; d(a_m)\} \end{aligned} \tag{9}$$

The program P contains $m + 1$ rules and the program P_{LG} contains $m + m^n$ rules.

3.2 Dependency graphs

Definition 1 *The* dependency graph $G_P = (V_P, E_P)$ *of a logic program P is constructed as follows:*

1. $V_P = \{A \quad | \text{ } A$ *is a predicate symbol in* $P\}$
2. $(a, b) \in E_P$ *iff there exists a rule $R \in P$ where a is the predicate symbol in head and b is a predicate symbol in body.*

A predicate a depends *on a predicate b iff there exists a path from a to b in the dependency graph.*

It should be noted that the above definition of dependency graphs is slightly different from the one presented in [13] where edges are defined in terms of atoms, not predicates.

Example 4 A small family relations database might be coded in following way:

$$\begin{aligned} ancestor(X, Y) &\leftarrow ancestor(X, Z), parent(Z, Y) \\ ancestor(X, Y) &\leftarrow parent(X, Y) \\ son(X, Y) &\leftarrow parent(Y, X), male(X) \\ daughter(X, Y) &\leftarrow parent(Y, X), female(X) \end{aligned} \tag{10}$$

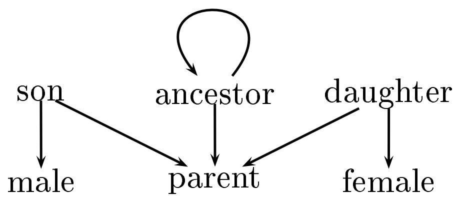

Figure 1: The dependency graph of family relationships

The dependency graph of the program is shown in Figure 1.

The dependency graph of a logic program shows how predicates are defined in terms of other predicates. In particular, recursive definitions cause loops in the graph (like *ancestor* in Figure 1) and leaf nodes (*parent*, *male* and *female*) are predicates whose extensions are completely specified by facts of the program.

As we can see, the extension of predicate *daughter* is simply determined by the extensions of *parent* and *female*. In fact, the extension can be calculated with a single natural join operation on *parent* and *male*.

3.3 Domain predicates

As long as a predicate is not defined recursively or in terms of recursively defined predicates its extension can be calculated with relational algebra. So it is possible to define the concept of a domain predicate in terms of paths in the dependency graph.

Definition 2 *A predicate* $p \in P$ *is a* domain predicate *iff it holds that every path starting from node* $p \in V_P$ *is cycle-free.*

Example 5 In the family database all other predicates but *ancestor* are domain predicates.

The set of domain predicates in a set of rules P is denoted by $\mathrm{Dom}(P)$. If the set of rules has only one member R the braces are left out so that $\mathrm{Dom}(\{R\}) = \mathrm{Dom}(R)$. The sets of domain predicates that occur in negative literals in P and in positive literals are denoted by $\mathrm{Dom}^-(P)$ and $\mathrm{Dom}^+(P)$, respectively. A predicate may be in both $\mathrm{Dom}^-(P)$ and $\mathrm{Dom}^+(P)$.

The set of all ground instances of the domain predicates of a program P is denoted by $\mathrm{Dom}_{\mathrm{G}}(P)$. Similarly, the set of ground instances of $\mathrm{Dom}^-(P)$ is denoted by $\mathrm{Dom}^-_{\mathrm{G}}(P)$. All ground instances of domain predicates are true in each stable model of the program P.

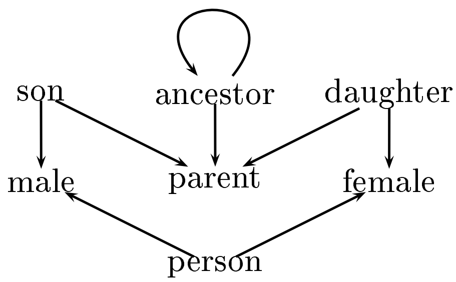

Figure 2: The dependency graph of modified family relationships

Example 6 Let

$$\begin{aligned} P = \{ & d_1(a,b); d_1(c,b); d_2(a,b); d_2(b,c); d_3(a); \\ & p(X,Z) \leftarrow d_1(X,Y), d_2(Y,Z), q(X,Y), \mathrm{not}\, d_3(X); \\ & q(X,b) \leftarrow d_3(X), p(X,b)\}. \end{aligned} \tag{11}$$

Here

$$\begin{aligned} \mathrm{Dom}(P) &= \{d_1, d_2, d_3\} \\ \mathrm{Dom}^-(P) &= \{d_3\} \\ \mathrm{Dom}_\mathrm{G}(P) &= \{d_1(a,b), d_1(c,b), d_2(a,b), d_2(b,c), d_3(a)\} \\ \mathrm{Dom}_\mathrm{G}^-(P) &= \{d_3(a)\}. \end{aligned}$$

3.4 Strongly range restricted logic rules

Having defined domain predicates it's now possible to define a class of rules which may be grounded locally:

Definition 3 *A rule R is* strongly range restricted *if it holds that if a variable appears in the rule it also appears in a positive domain predicate in the body.*

Unlike normal range restriction, it is not possible to say whether a given rule is a strongly range restricted or not without looking at the whole program. Also, if a new rule is added to the program the domain status of some predicates may change, and thus a rule may either lose or gain the property.

Definition 4 *A logic program is* strongly range restricted program *iff every rule in it is strongly range restricted.*

Example 7 The program (10) is not strongly range restricted, since the variable X in the first rule of *ancestor* doesn't appear in a domain predicate.

However, we can redeem the situation by adding domain for X, resulting in rules:

$$\begin{aligned} ancestor(X,Y) &\leftarrow ancestor(X,Z), parent(Z,Y), person(X) \\ person(X) &\leftarrow male(X) \\ person(X) &\leftarrow female(X) \end{aligned}$$

The dependency graph of the modified program is in Figure 2.

Example 8 We may encode the problem of finding domain predicates of a logic program P as a strongly range restricted logic program in following way:

$$\begin{aligned} dp(X) &\leftarrow pred(X), \mathrm{not}\, cyclic(X) \\ cyclic(X) &\leftarrow pred(X), pred(Y), depends(X,Y), depends(Y,Y) \qquad (12) \\ depends(X,Y) &\leftarrow edge(X,Y) \\ depends(X,Z) &\leftarrow pred(X), depends(X,Y), edge(Y,Z) \end{aligned}$$

Here $dp(X)$ is true if X is a domain predicate, $cyclic(X)$ is true if X depends on a some predicate that depends on itself, and predicates *pred* and *edge* are given by the dependency graph of P.

3.5 Domains

It is possible to construct complex domains from simple ones by using set arithmetic. All basic set operations can be encoded as normal logic programs quite easily. The rules corresponding to set operations are shown in Table 1.

The only set operation missing from Table 1 is complementation. The obvious way to compute complementation with rule $C(X) \leftarrow \mathrm{not}\, P(X)$ doesn't work since the rule is not range restricted. A better way is to define a domain Q with all possible values for P and then use set difference $C(X) \leftarrow Q(X), \mathrm{not}\, P(X)$.

3.6 Algorithm for grounding strongly range restricted programs

The first step in grounding a logic program P is to construct the dependency graph of P, find the domain predicates and calculate their extensions. The grounding of other predicates is done using algorithm **Ground**(P), which is presented in Figure 3. The basic algorithm has a few shortcomings and a refined one is presented in Section 4.3.

The algorithm **Ground**(P) starts by adding all ground instances of the domain predicates to the local grounding P_{LG}. In the next step all rules with

```
function Ground(P)
    P_LG := Dom_G(P)
    P := P − {R ∈ P  | the head of R ∈ Dom(P)}
    foreach R ∈ P do
        P_LG := P_LG ∪ GroundRule(R)
    end for
    return P_LG
end

function GroundRule(R)
    P_R := ∅
    if R is variable-free do
        P_R := {R}
        return P_R
    end if
    D := Dom⁺(R)
    d := the predicate with smallest domain in D
    D := D − d
    B := Bind(d)
    while D is not empty do
        d := choose the smallest domain in D
        D := D − d
        B := B ⋈ Bind(d)
    end while
    foreach b ∈ B do
        discard := false
        N := Dom⁻(R)
        while N ≠ ∅ and discard = false do
            atom := ground a literal n ∈ N with variable binding b
            N := N − n if atom ∈ Dom_G⁻(R) do
                discard := true
            end if
        end while
        if discard = false do
            P_R := P_R ∪ {R grounded with binding b}
        end if
    end for
    return P_R
end
```

Figure 3: The grounding algorithm for strongly range restricted programs

Operation	Rules
$U = P \cup R$	$U(X) \leftarrow P(X)$ $U(X) \leftarrow R(X)$
$I = P \cap R$	$I(X) \leftarrow P(X), R(X)$
$D = P \setminus R$	$D(X) \leftarrow P(X), \mathrm{not}\, R(X)$
$C = P \times R$	$C(X,Y) \leftarrow P(X), R(Y)$
$J = P \bowtie R$	$J(X,Y,Z) \leftarrow P(X,Y), R(Y,Z)$
$S = P \triangle R$	$S(X) \leftarrow P(X), \mathrm{not}\, R(X)$ $S(X) \leftarrow R(X), \mathrm{not}\, P(X)$

Table 1: Set operations for domains

a domain predicate as the head are discarded from P. Then each remaining rule $R \in P$ is grounded using algorithm **GroundRule**(R).

The basic idea of **GroundRule**(R) is to construct a relation B of all possible variable bindings for the rule. The relation has as many attributes as there are variables in the rule, and it is constructed by creating the natural join of all positive domain predicates in the rule body. After the relation is completed each possible variable binding is checked against negative domain literals in the rule body. If a binding makes some negative domain predicate true it is discarded from B, otherwise the variables in the rule body are substituted according to the binding and the result is added to P_{LG}.

In function **GroundRule**(P) D is the set of positive domain predicates in the body that have not been used yet, B is the set of all possible variable bindings and N is the set of negative domain predicates. The function Bind(d) returns the variable bindings of the domain predicate d.

Example 9 The algorithm **Ground**(P) works for program (11) as follows.

$$\begin{aligned} P = \{ & d_1(a,b); d_1(c,b); d_2(a,b); d_2(b,c); d_3(a); \\ & p(X,Z) \leftarrow d_1(X,Y), d_2(Y,Z), q(X,Y), \mathrm{not}\, d_3(X); \\ & q(X,b) \leftarrow d_3(X), p(X,b)\}. \end{aligned} \tag{11}$$

Here d_i are domain predicates, p and q are non-domain predicates. When grounding the program we first initialize

$$P_{LG} = \{d_1(a,b); d_1(c,b); d_2(a,b); d_2(b,c); d_3(a)\}.$$

After removing the rules with domain predicates as heads only two rules are left. The function **GroundRule**(R) proceeds for rule with p as head as follows.

1st step			2nd step		
X	Y	Z	X	Y	Z
	a	b	a	b	c
	b	c	c	b	c

Figure 4: Construction of the relation B in the Example 9

1. Initialize $P_R := \emptyset, D := \{d_1, d_2\}, B := \emptyset$.
2. Choose $d := d_1$. Set $D := \{d_2\}, B := \{\{X/a, Y/b\}, \{X/c, Y/b\}\}$. Since both domains were of the same size, either could have been chosen.
3. As D is not empty, choose $d := d_2, D := \emptyset$.
4. Calculate join

$$\begin{aligned} B :=& B \bowtie \{\{Y/a, Z/b\}, \{Y/b, Z/c\}\} \\ =& \{\{X/a, Y/b, Z/c\}, \{X/c, Y/b, Z/c\}\} \end{aligned}$$

5. As D is empty, start iterating through all found bindings. Choose $b := \{X/a, Y/b, Z/c\}$, set discard $:= false$ and $N := \{d_3\}$.
6. Choose $n := d_3$, and set $atom := d_3(X/a) = d_3(a)$.
7. Since $d_3(a) \in \mathrm{Dom_G}(\{d_3\})$, set discard $:= true$.
8. Set N is now empty and the inner loop condition fails.
9. As discard $= true$, jump back to start of outer loop.
10. Choose $b := \{X/c, Y/b, Z/c\}$, set discard $:= false$ and $N := \{d_3\}$.
11. Choose $n := d_3$, and set $atom := d_3(X/c) = d_3(c)$.
12. Atom $d_3(c) \notin \mathrm{Dom_G}(\{d_3\})$, so discard remains $false$.
13. Set N is empty, and discard $= false$ so set

$$P_R := \emptyset \cup \{p(c, b) \leftarrow d_1(c, b), d_2(b, c), \mathrm{not}\, d_3(c), q(c, b)\}$$

14. Set B is now empty, so P_R is returned.

The same is done to the only rule left, which returns

$$P_R = \{q(a, b) \leftarrow d_3(a), p(a, b)\},$$

giving the total ground program

$$
\begin{aligned}
P_{LG} = \{ & d_1(a,b); d_1(c,b); d_2(a,b); d_3(a) \\
& p(c,b) \leftarrow d_1(c,b), d_2(b,c), \mathrm{not}\, d_3(c), q(c,b) \\
& q(a,b) \leftarrow d_3(a), p(a,b)\}
\end{aligned} \tag{13}
$$

Theorem 1 *If P is a strongly range restricted logic program, P_G is the set of Herbrand instances **HI**(P) and P_{LG} the local grounding obtained by using algorithm **Ground**(P), then Δ is a stable model of P_G if and only if Δ is a stable model of P_{LG}.*

Proof.

By Proposition 1 $\Delta = \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta}))$, so we have to prove that

$$\mathrm{Dcl}\,(P_{LG}, \mathrm{not}\,(\bar{\Delta})) = \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta})).$$

Because deductive closure $\mathrm{Dcl}\,(\cdot,\cdot)$ is monotonic, and $P_{LG} \subseteq P_G$, it follows that $\mathrm{Dcl}\,(P_{LG}, \mathrm{not}\,(\bar{\Delta})) \subseteq \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta}))$.

As Δ is the minimal model of P_G^{Δ}, the other direction may be proved by showing that at each step of construction (4) $T_{P_G^{\Delta}} \uparrow n \subseteq T_{P_{LG}^{\Delta}} \uparrow n$. The proof is constructed inductively.

When $n = 0$, $T_{P_G^{\Delta}} \uparrow n$ and $T_{P_{LG}^{\Delta}} \uparrow n$ are both $\emptyset$, and $T_{P_G^{\Delta}} \uparrow n \subseteq T_{P_{LG}^{\Delta}} \uparrow n$ trivially.

We make an induction hypothesis that for some k it holds that for all a, $a \in T_{P_G^{\Delta}} \uparrow i$ implies $a \in T_{P_{LG}^{\Delta}} \uparrow i, i \leq k$. If there is an atom $a \notin T_{P_G^{\Delta}} \uparrow k$ but $a \in T_{P_G^{\Delta}} \uparrow k+1$, then there has to be a rule $R_G \in P_G^{\Delta}$ of the form

$$a \leftarrow d_1, \ldots, d_n, l_1, \ldots, l_m, \tag{14}$$

where $d_1, \ldots, d_n, l_1, \ldots, l_m \in T_{P_G^{\Delta}} \uparrow k$. Here d_i are the domain predicates in the rule body. By induction hypothesis $d_1, \ldots, d_n, l_1, \ldots, l_m \in T_{P_{LG}^{\Delta}} \uparrow k$ and we have to show only that $R_G \in P_{LG}^{\Delta}$.

The rule R_G is instantiated from some rule $R \in P$ with a variable binding b. As P is strongly range restricted, each variable in the body of R occurs in a positive domain predicate. Thus for each argument t of literals l_i it holds that either t is a variable substitution and so it occurs as an argument of some d_j, or t is a constant and so it is same in each instantiation of R.

As d_i are all members of $T_{P_{LG}^{\Delta}} \uparrow k$ by induction hypothesis, and $\mathrm{Dcl}\,(\cdot,\cdot)$ is monotonic, d_is must all belong to Δ and thus they belong to $\mathrm{Dom_G}(P)$. Because in function **GroundRule**(R) the set B of all possible variable bindings

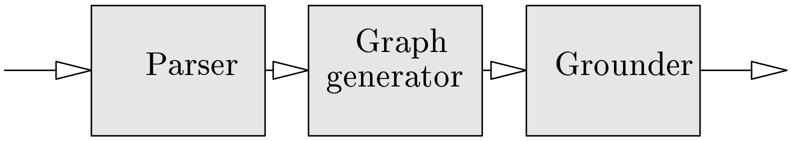

Figure 5: Processing of a logic program

is constructed by taking a natural join over extensions of all $\mathrm{Dom}^{+}(R)$, the binding $b \in B$, so $R \in P_{LG}$ and $a \in T_{P_{LG}^{\Delta}} \uparrow k+1$. This completes the induction.

As

$$\begin{gathered}\mathrm{Dcl}\,(P_{LG}, \mathrm{not}\,(\bar{\Delta})) \subseteq \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta})) \text{ and} \\ \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta})) \subseteq \mathrm{Dcl}\,(P_{LG}, \mathrm{not}\,(\bar{\Delta})) \\ \text{it follows that} \\ \mathrm{Dcl}\,(P_G, \mathrm{not}\,(\bar{\Delta})) = \mathrm{Dcl}\,(P_{LG}, \mathrm{not}\,(\bar{\Delta})),\end{gathered}$$

and the proof is completed. □

4 Implementation

The local grounder, **lparse**, is implemented in C++ on PC-Linux system using standard GNU tools. It has also been tested on Digital Alpha platform. The implementation is available at

`http://www.tcs.hut.fi/pub/smodels/lparse/index.html`

Logically the processing of a logic program can be divided in four parts:

1. Parsing the input.
2. Generation of dependency graph and range checking.
3. Calculating the extensions of domain predicates
4. Grounding rules

In actual implementation the phases two and three are mixed so that the predicate domains are calculated as domain predicates are found. The process is illustrated in Figure 5.

4.1 Parser

The parser is implemented with Bison and it takes as its input the logic program and produces a parse tree corresponding to it as its output. The full grammar is in Appendix A.

In addition to normal logic programs the grammar of **lparse** also allows a few notational shortcuts: ranges and functions.

Ranges. A *range* is a sequence of non-negative integer constants which may occur in a rule where a variable could. A range is a shorthand notation for defining domains of predicates, for example $p(1..n)$ defines predicates $p(1)$, ... , $p(n)$. Internally a range is transformed into a definition of a new domain predicate, so that

$$p(i..k) \tag{15}$$

is equivalent to

$$\begin{aligned} & p(X) \leftarrow dom_1(X) \\ & dom_1(i) \\ & \quad\vdots \\ & dom_1(k) \end{aligned} \tag{16}$$

Functions. The functions in **lparse** are not "true" functions in the sense that they only affect the grounding and **smodels** doesn't see them. In fact they behave more like macros, giving a convenient notation for possibly complex logic programs. A logic program P with **lparse** functions could always be replaced by an equivalent function-free program P'. The full syntax is described in **lparse** User's manual, which is available at

`http://www.tcs.hut.fi/pub/smodels/lparse/lparse.ps.gz`

In a program P a n-ary function $f(a_1, \ldots, a_n)$ may occur in two roles:

1. As an argument to a predicate or a function. For example, let

   $$P = \{t(1..2); next(plus(X,1), X) \leftarrow t(X)\} \tag{17}$$

   where *next* is a predicate and *plus* is a function. Semantically, this program is equivalent to

   $$P' = \{t(1); t(2); next(2,1) \leftarrow t(1); next(3,2) \leftarrow t(2)\} \tag{18}$$

2. As a literal in a rule body. A function f in a rule body works as a test. When grounding the rule with a given variable binding the return value of f is checked. If it is $true$[1] then f is removed from the body and the resulting ground rule is added to P_{LG}, otherwise the binding is discarded. For example, let

$$P = \{t(1..2); p(X,Y) \leftarrow t(X), t(Y), X > Y\} \tag{19}$$

is equivalent to program

$$P' = \{t(1); t(2); p(2,1) \leftarrow t(2), t(1)\} \tag{20}$$

Example 10 An undirected n-node ladder graph (Figure 10) could be encoded using ranges and functions in following way

$$\begin{aligned} & vertex(1..n) \\ arc(X, X-2) \leftarrow & vertex(X), X > 2 \\ arc(X, X-1) \leftarrow & vertex(X), X \bmod 2 = 0, X > 1 \end{aligned} \tag{21}$$

4.2 Dependency graphs and domain generation

After parsing the resulting parse tree is processed. Every identifier is added to the correct symbol table, literals and rules are constructed and associated with correct predicates. At the same time one node is allocated in dependency graph for each predicate and an arc is added for each literal in rule bodies.

After the graph is generated the domain predicates are searched using algorithm depicted in Figure 6. The algorithm is based on R. E. Tarjan's depth-first algorithm for finding strongly connected components of a graph[12, pp. 481–483]. Tarjan's construction could be simplified because for a given predicate we only want to know if it is a part of some cycle, and we are not interested in finding the actual cycles.

The idea is to make a depth-first search through all starting nodes and search for cycles. The search information is held in two arrays, domain and visited. Domain holds the domain status of predicates. At start it is initialized to value *unknown*. When the domain status of a predicate p is obtained the value of domain[p] is set to $true$ or $false$. Visited holds the information which nodes have been visited during current branch of the search.

When a node nd is visited the first thing to do is to set visited[nd] and domain[nd] both to $true$. After that the status of each child ch of nd is checked. If we have visited one of the child nodes (i.e. visited[ch] $= true$) earlier in

[1] As in C/C++ all non-zero return values are considered true.

```
function CheckDomains(P)
    foreach predicate p ∈ P do
        domain[p] := unknown
        visited[p] := false
    end for
    foreach predicate p ∈ P do
        if domain[p] = unknown do
            Visit(p)
        end if
    end for
end

function Visit(node nd)
    val = true
    visited[nd] := true
    domain[nd] := true
    foreach child ch of nd do
        if visited[ch] = true do
            val := false
        else if domain[ch] = unknown do
            val := Visit(ch)
        else do
            val := domain[ch]
        end if
        if val = false do
            domain[nd] := false
        end if
    end for
    if domain[nd] = true do
        { Calculate the extension of predicate nd }
    end if
    visited[nd] := false
    return domain[nd]
end
```

Figure 6: Algorithm for finding domain predicates

current branch of the search tree, then we have found a cycle and can mark the node nd as a non-domain predicate ($\mathsf{domain}[nd] := false$).

If the child ch is not visited, we examine it's domain status. If $domain[ch] = true$ we know that ch is a domain predicate and we can continue with next child. If $domain[ch] = false$, we know that nd depends on a non-domain predicate and thus it cannot be a domain predicate, and we may set $\mathsf{domain}[nd] := false$. Finally, if $domain[ch] = unknown$ we have to recursively visit in the node ch before we can decide whether nd is a domain predicate or not.

If all children are domain predicates, the extension of nd is calculated using algorithm $\mathbf{GroundRule}_2(R)$ presented in next subsection for each rule R that has nd as head.

When the domain status of a node is calculated, it is not necessary to visit it again, because the domain-array holds the correct value. Since each node has to be visited exactly once, **Visit**(nd) has a linear time complexity with respect to the size of the dependency graph.

4.3 Grounding

The algorithm **GroundRule**(R) may be improved in at least two ways. The first flaw of the basic algorithm is that the intermediary results of the join $B \bowtie \mathrm{Bind}(d)$ have to be stored in memory. The following example shows why this is not acceptable.

Example 11 Let

$$\begin{aligned} P = \{&d_1(1..n); d_2(n+1..n+k); \\ &p(X_1, \ldots, X_m) \leftarrow d_1(X_1), \ldots d_1(X_m), d_2(X_1)\} \end{aligned} \tag{22}$$

where $1 < n < k - 1, m \geq 1$. As domains of d_1 and d_2 are disjoint there are no ground instances for the only nontrivial rule of P. However, because $|d_1| < |d_2|$ **GroundRule**(R) calculates first the m-way join of d_1 which has n^m items which have to be stored in memory. Even with small domain of size $n = 100$ and $m = 4$ this takes over 10^8 words of memory.

The other flaw in **GroundRule**(R) is that the bindings that don't satisfy the negative literals are filtered out in the end. It is better to weed them as soon as we can, that is, immediately after we get a binding for each variable in the negative literal. Also, we may extend the check to cover all negative literals, instead of testing just negative ground literals. The motive for this is that there may be some ground instances of a non-domain predicate given as facts in the program and all rules that depend on negations of those facts can be discarded.

Example 12 Let

$$P = \{d_1(1, 1..n); d_2(1..n, 1..n); d_3(1); \qquad (23)$$
$$p(X, Z) \leftarrow d_1(X, Y), d_2(Y, Z), \mathrm{not}\, d_3(X)\}.$$

Here again there is no ground instances for the only nontrivial rule. If the join $d_1(X, Y) \bowtie d_2(Y, Z)$ is calculated first then the existence of $d_3(1)$ has to be checked n^3 times. Since the literal $\mathrm{not}\, d_3(X)$ doesn't have any variables common with atom $d_2(Y, Z)$, the check can be done before the join and only n tests has to be made.

The two optimizations above are by no means the only possible ones, and all standard database optimizations could be implemented. However, the two implemented optimizations was seen to be the most important ones, for reasons given above.

The improved algorithm **GroundRule**$_2(R)$ is presented in Figure 7. The main idea is to put the domain predicates in the array $\mathsf{literals}$ sorted by increasing sizes of the domains. The ground instances of domain predicates are stored in array $\mathsf{instances}$, and $\mathsf{instances}[i] = \mathrm{Dom}_{\mathrm{G}}(\mathsf{literals}[i])$.

All ground instances are generated by first selecting an instance of $\mathsf{literals}[1]$ from $\mathsf{instances}[1]$ and then going through all compatible instances of $\mathsf{literals}[2], \ldots, \mathsf{literals}[n]$, when there are n positive domain predicates in the rule body.

The negative literals are stored in an array of sets of literals (array neglit). A negative literal l is in $\mathsf{neglit}[n]$ if its truth valuation can be checked after we have fetched an instance of $\mathsf{instances}[n]$. In Example 12 $d_3(X) \in \mathsf{neglit}[1]$, because the value of variable X is obtained from the first domain predicate in the rule body. If the predicate would have been $d_3(Y)$ instead, then it would have been in $\mathsf{neglit}[2]$.

Given a literal L, ground instance I of L and a variable binding B, the algorithm **Bind**(B, L, I) (Figure 8) checks if I is compatible with B. If a variable has a different value in I than it has in B the binding fails and **Bind**(B, L, I) leaves B untouched. The ground instance I is subsequently discarded in **GroundRule**$_2(R)$. If I and B are compatible we augment B by adding bindings for variables in L that were not in B before.

Example 13 The program (11) was grounded before with the basic algorithm, and one rule of it is now grounded using algorithm **GroundRule**$_2(R)$.

```
function GroundRule₂(R)
    P_R = ∅
    max := |Dom⁺(R)|
    pos := 1
    B := ∅
    array instances
    array literals := sort Dom⁺(R)
    for i := 1 to |Dom⁺(R)| do
        instances[i] := Dom_G(literals[i])
    end for
    { for every variable in R find the position when it becomes defined }
    { for each negative literal in R find at what value of pos
    it can be calculated and add it to neglit[pos] }
    while pos > 0 do
        discard := false
        if instances[pos] = ∅ do
            instances[pos] := Dom_G(literals[pos])
            pos := pos -1
        else
            Let atom ∈ instances[pos]
            instances[pos] := instances[pos] - atom
            ⟨B, b⟩ = Bind(B, literals[pos], atom)
            if b = false do
                discard := true
            end if
            foreach n ∈ neglit[pos] do
                it := ground n with binding B
                if it ∈ Dom⁻_G(R) do
                    discard := true
                end if
            end for
            if discard = false do
                if pos = max do
                    P_R := P_R ∪ {R grounded with binding B }
                else
                    pos := pos + 1
                end if
            end if
        end if
        { clear variable bindings of level pos and above}
    end while
    return P_R
end
```

Figure 7: The improved algorithm for grounding a rule R

```
function Bind(B, L, I)
    i := 1
    discard := false
    b := ∅
    while i ≤ the arity of L and discard = false do
        if ith argument of L is a variable v do
            val := I[i]
            if there is a conflicting value for v in B do
                discard := true
            else
                b := b ∪ {v/val}
            end if
        end if
        i := i + 1
    end while
    if discard = true
        return ⟨B, false⟩
    else
        return ⟨B ∪ b, true⟩
    end if
end
```

Figure 8: An algorithm for binding variables

The rule chosen is the same as before.

$$\begin{aligned}P = \{&d_1(a,b); d_1(c,b); d_2(a,b); d_2(b,c); d_3(a);\\ &p(X,Z) \leftarrow d_1(X,Y), d_2(Y,Z), q(X,Y), \mathrm{not}\, d_3(X);\\ &q(X,b) \leftarrow d_3(X), p(X,b)\}.\end{aligned} \tag{11}$$

1. Set $P_R := \emptyset$, $\mathsf{max} := 2$, $\mathsf{pos} := 1$, and $B := \emptyset$.

2. Sort $\mathrm{Dom}(R)$ and set $\mathsf{literals}[1] := d_1$, $\mathsf{literals}[2] = d_2$, $\mathsf{instances}[1] := \{(a,b),(c,b)\}$, and $\mathsf{instances}[2] := \{(a,b),(b,c)\}$.

3. As the only variable in $d_3(X)$ gets its value from literal $d_1(X,Y)$, which is calculated in $\mathsf{pos} = 1$, set $\mathsf{neglit}[1] := \{d_3\}$, and $\mathsf{neglit}[2] := \emptyset$.

4. Position $\mathsf{pos} > 0$, so set $\mathsf{discard} := false$.

5. Since $\mathsf{instances}[1] \neq \emptyset$,choose $atom := (a,b)$, set $\mathsf{instances}[1] := \{(c,b)\}$.

6. As B is empty, there is no conflicting values of variables, and binding succeeds. $B := \{X/a, Y/B\}$.

7. Choose the only possible $n \in \mathsf{neglit}[1] (n := d_3)$, and ground it using binding B to get $it := d_3(a)$. As $it \in \mathrm{Dom}^-_{\mathrm{G}}(R)$, set $\mathsf{discard} := true$.

8. Since $\mathsf{discard} := true$, clear variable binding $B := \emptyset$ and jump back to the top of while loop.

9. Position $\mathsf{pos} > 0$, so set $\mathsf{discard} := false$.

10. Since $\mathsf{instances}[1] \neq \emptyset$,choose $atom := (c,b)$, set $\mathsf{instances}[1] := \emptyset\}$. Binding of $atom$ also succeeds, and we get $B := \{X/c, Y/b\}$.

11. Because $d_3(c) \notin \mathrm{Dom}^-_{\mathrm{G}}(R)$, there are no unsatisfied negative literals in the rule body, and $\mathsf{discard}$ remains $false$.

12. Since $\mathsf{discard} = false$ and $\mathsf{pos} < \mathsf{max}$, set $\mathsf{pos} := 2$ and jump back to the top of while loop.

13. Choose $atom := (a,b)$, set $\mathsf{instances}[2] := \{(b,c)\}$, and try to bind $atom$. However, the binding fails because variable Y is already binded to b in B.

14. Choose $atom := (b,c)$, set $\mathsf{instances}[2] := \{(b,c)\}$, and bind $atom$ to get $B := \{X/c, Y/b, Z/c\}$.

15. There are no negative literals at $\mathsf{pos} = 2$, so in the end $\mathsf{discard} = false$ and $\mathsf{pos} = \mathsf{max}$. The rule is now grounded to get

$$P_R = \{\} \leftarrow d_1(c, b), d_2(b, c), \text{not}\, d_3(c), q(c, b)$$

16. Now $\mathsf{instances}[2]$ is empty and it is reinitialized to $\mathrm{Dom}_{\mathrm{G}}(d_2)$. Also, set $\mathsf{pos} := 1$.

17. $\mathsf{instances}[1]$ is also empty, so set $\mathsf{pos} := 0$.

18. As $\mathsf{pos} = 0$ the algorithm terminates.

4.4 Complexity of grounding algorithm

Proposition 2 *The algorithm **GroundRule**$_2(R)$ has linear space complexity with respect to the number of domain predicates in the rule body.*

This can be seen by noting that the only memory allocated in **GroundRule**$_2(R)$ is for the arrays literals, domains, and neglit. The size of each array is $|\mathrm{Dom}^+(R)|$, so the total space needed is $3 \cdot |\mathrm{Dom}^+(R)|$.

After a rule R is grounded using **GroundRule**$_2(R)$ it is of no use in any way, so there is no need to store it in memory. Even though grounding of rules for non-domain predicates can be done in linear space, the worst case space complexity of the whole grounding is still exponential because we have to store all ground instances of domain predicates in memory during grounding, and there may be exponential number of them. An example of this occurrence is program (9) where every predicate is a domain predicate and the size of the largest domain is exponential.

Proposition 3 *The worst case space complexity of the algorithm **Ground**(P) is linear with respect to the total size of the extensions of domain predicates in P, but exponential with respect to the size of P.*

In practice the exponential size is often avoided. For example all graph problems problems considered in Section 5 have only a linear number of ground instances of domain predicates. Even if the total size of local grounding is exponential, the size of the domains and thus the total space used may still be linear.

Example 14 Let

$$\begin{aligned}P = \{&d_1(x_1), \ldots, d_n(x_n), \mathrm{not}\, q(x_1, \ldots, x_n) \leftarrow p(x_1, \ldots, x_n)\\ &d(x_1), \ldots, d(x_n), \mathrm{not}\, p(x_1, \ldots, x_n) \leftarrow q(x_1, \ldots, x_n)\\ &d(a_1); \ldots; d(a_m)\end{aligned}$$

The program P has only m ground instances of domain predicates, but the number of rules in local grounding $P_{LG} = 2m^n + m$.

4.5 Data structures

Ground instances. Ground instances of predicates are implemented as arrays of integers which are stored in a closed hash table. The hash table resizes automatically when the load factor exceeds 0.5.

Hash tables allow constant time insert and seek operations for single items, but it is not possible to find all ground instances having a given value at a given position efficiently. However, the latter ability is vital for efficient joining.

Example 15 Let

$$\begin{aligned}P = \{&d_1(1..100, 2..11); d_2(1..2, 1..1000)\\ &p(X, Z) \leftarrow d_1(X, Y), d_2(Y, Z)\}\end{aligned} \tag{24}$$

When calculating the join $d_1 \bowtie d_2$ we have to iterate through all 1000 instances of d_1 and find all compatible instances of d_2. If we don't have an efficient algorithm to find all items of d_2 for a given value of Y, we have to compare all 2000 instances of d_2 with all 1000 instances of d_1 to find the 10^5 instances of $p(X, Z)$, the total number of comparisons being roughly 2×10^6.

However, if we can generate an index for storing all instances of d_2 with a given value of Y in the same place, we need only 1 comparison for each instance of d_1 when $Y \neq 2$, and 1001 comparisons when $Y = 2$ with a total of approximately 10^5 comparisons.

The indices are implemented as Red-Black–Trees. The ordinary rb–tree implementation has also the flaw that the keys with same values are not stored together. This can be circumvented by storing all instances with identical key values to a linked list and storing pointers to this list in tree. The generation of indices is not globally optimized, but if within a rule there is a choice between few possible indices, a pre-existing index is used if possible.

Predicates. The predicates are implemented as a C++ class with following attributes:

- `arity`: arity of the predicate.
- `rules`: a list of all rules for the predicate.
- `atoms`: ground instances of the predicate.
- `status`: is the predicate a domain predicate or not?
- `indices`: the calculation of joins is made more efficient by using indices. Indices are created when needed and an existing index is used if possible. However, there is no global optimization for indices.

Rules. There is also a C++ class for rules with following attributes:

- `head`: well, the literal that's the head of the rule
- `positive`: list of all positive literals in the rule body.
- `negative`: list of all negative literals in the rule body.
- `functions`: list of all functions in the rule body.

Literals. The literals have following attributes:

- `ground`: is the literal a ground literal?
- `negative`: is the literal a not-atom?
- `arity`: arity of the literal.
- `numvars`: number of variables in the literal.
- `pred`: the predicate symbol of the literal
- `args`: types (constant, variable, function) of the arguments of the literal.
- `vars, cons, funs`: arrays containing the arguments of the literal. The arguments are stored as pointers to the corresponding symbol tables.

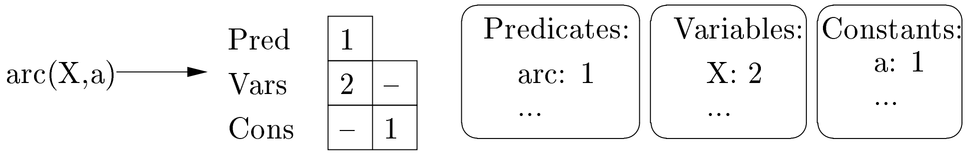

Figure 9: Storage of literals

Example 16 Let $L = arc(X, a)$. Then L is stored in memory in manner described in Figure 9. The attribute `pred` holds pointer to the correct predicate in symbol table. The arrays `vars` and `cons` hold the information about the arguments of the literal. As the first argument of L is variable X, `vars[1]` holds the pointer to the correct variable in symbol table. The second argument is a constant a and `cons[2]` points to it. The other two array places (`vars[2]` and `cons[1]`) hold null values.

Symbol tables. There are five symbol tables. One for each of predicates, variables, atoms, functions and grounded atoms. The symbol tables are implemented with closed hash tables.

5 Evaluation

The efficiency of local grounding was tested using few classical problems. The combination of local grounder and **smodels** was compared against **smodels** with **parse** and `dlv`, a knowledge representation system based on disjunctive logic programming developed at Vienna University of Technology [2].

5.1 Problems

Graph colorability. Given a set of colors and graph represented by facts of the form $vertex(X)$ and $arc(X, Y)$ assign each node one of colors such that no two adjacent nodes have the same color.

The 3-colorability of graphs was tested using two types of graphs, ladder and simplex graphs. The 4-colorability of graphs was tested using random plane graphs in addition to the types used with 3-colorability. All graphs were undirected and there was at most one arc between any two nodes. All three systems used same graphs.

The graph 3-colorability can be represented as a strongly range restricted

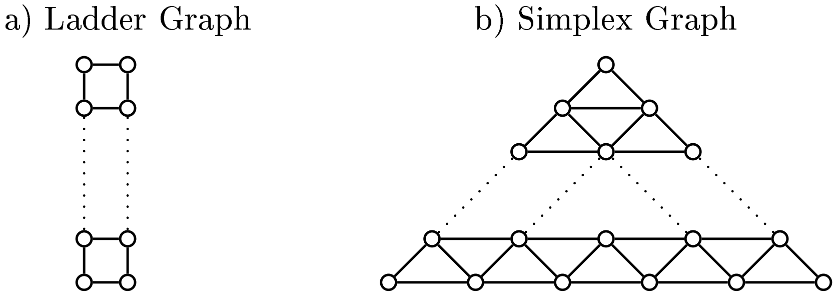

Figure 10: Graph types used in testing graph colorability

logic program in following way:

$$\begin{aligned}
col(X,r) &\leftarrow vertex(X), \text{not}\, col(X,g), \text{not}\, col(X,b)\\
col(X,g) &\leftarrow vertex(X), \text{not}\, col(X,r), \text{not}\, col(X,b)\\
col(X,b) &\leftarrow vertex(X), \text{not}\, col(X,g), \text{not}\, col(X,r)\\
fail &\leftarrow arc(X,Y), color(C), col(X,C), col(Y,C)\\
color(r).&color(g).color(b).
\end{aligned} \tag{25}$$

For the graph 4-colorability one new rule was added for the new color and

The encoding of problem for `dlv` was taken from [2]:

$$\begin{aligned}
col(X,r) \vee col(X,g) \vee col(X,b) &\leftarrow vertex(X)\\
&\leftarrow arc(X,Y), col(X,C), col(Y,C)
\end{aligned} \tag{26}$$

Blocks World. The blocks world is a simple domain where exists a number of blocks that can be moved and stacked. Initial and goal states are given and the problem is to find a set of legal moves to reach goal state from initial state.

The encoding of this problem is quite large and it is not shown here. However, it is described in [9].

Hamilton circuits. Given a graph the problem of Hamilton cycles is to find a path from a given initial node that visits each node in the graph exactly once and returns back to initial node.

The problem was tested using graphs presented in Figure 11. The number of Hamiltonian circuits in a n-node graph of that form is $2^{(n-1)/2}$ when $n \geq 3$. This can be seen by noting that starting from the leftmost node and

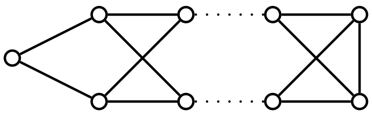

Figure 11: The graphs used in testing Hamilton circuits

proceeding rightwards there are exactly $\frac{n-1}{2}$ choice points each having two possible routes.

For each graph both the time to find one Hamilton circuit and the time to find all circuits was tested.

The strongly range restricted **smodels** encoding of the problem is following:

$$\begin{aligned}
hc(V1,V2) &\leftarrow arc(V1,V2), \mathrm{not}\, otherroute(V1,V2), \mathrm{not}\, eq(V1,V2) \\
otherroute(V1,V2) &\leftarrow arc(V1,V2), hc(V1,V3), arc(V1,V3), \mathrm{not}\, eq(V2,V3) \\
otherroute(V1,V2) &\leftarrow arc(V1,V2), hc(V3,V2), arc(V3,V2), \mathrm{not}\, eq(V3,V1) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (27) \\
reached(V2) &\leftarrow arc(V1,V2), hc(V1,V2), reached(V1), \mathrm{not}\, eq(V1,V2) \\
reached(V2) &\leftarrow arc(V1,V2), hc(V1,V2), initialnode(V1), \mathrm{not}\, eq(V1,V2) \\
fail &\leftarrow vertex(V), \mathrm{not}\, reached(V) \\
& initialnode(0)
\end{aligned}$$

The `dlv` version is essentially identical.

5.2 Results

The tests were run on a 433MHz Digital Alpha workstation with 128MB memory under Digital UNIX V4.0C. Timing information was obtained with Unix `time` utility by summing up `user` and `system` times. The `real` times were not used, because the test computers couldn't be reserved for exclusive use.

Unless otherwise stated only one stable model was searched for in each case. The tests were run three times and arithmetic mean was taken from the results.

The program versions used in the testing were: **smodels** 1.10, **parse** 1.3, **lparse** 0.9.0 and `dlv` 1.0.3. The local grounder was run with command line `lparse -d n test.case`, i.e. all domain predicates were dropped after grounding, and other programs were run without arguments.

The **smodels** implementation is available at

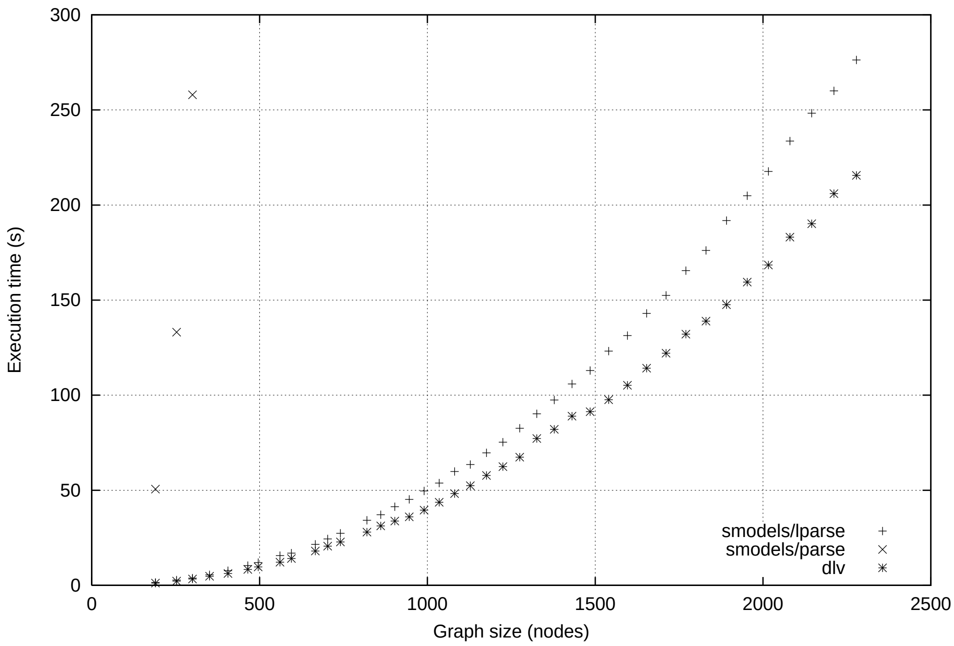

Figure 12: Results of graph 3-colorability for simplex graphs

`http://www.tcs.hut.fi/pub/smodels/`

The `dlv` system is available at

`http://www.dbai.tuwien.ac.at/proj/dlv/`

The test cases are available at

`http://www.tcs.hut.fi/pub/smodels/tests/lparse-tests.tar.gz`

Graph colorability. Simplex graphs were tested up to size of 2300 nodes and ladder graphs were tested up to size of 3000 nodes. The plane graphs were randomly generated using Stanford GraphBase [5] with three different graphs of each size. The results are shown in Figures 12–16.

With structured graphs (simplex and ladder) **smodels** with **parse** was clearly the worst, being able to solve graph colorings up to size of about 400 nodes at best. The **smodels** with **lparse** and `dlv` had similar performance, `dlv` being the best for ladder-4 and simplex-3, and **lparse** being the best for simplex-4 and ladder-3.

While nearly all other test cases fit nicely in the expected exponential curve the **lparse** with simplex-4 was quite erratic. There were two test cases (2016 and 2211 nodes) where **smodels** couldn't find a solution and a couple more (1128, 1431, 1711, and 1770 nodes) where a solution was found only after

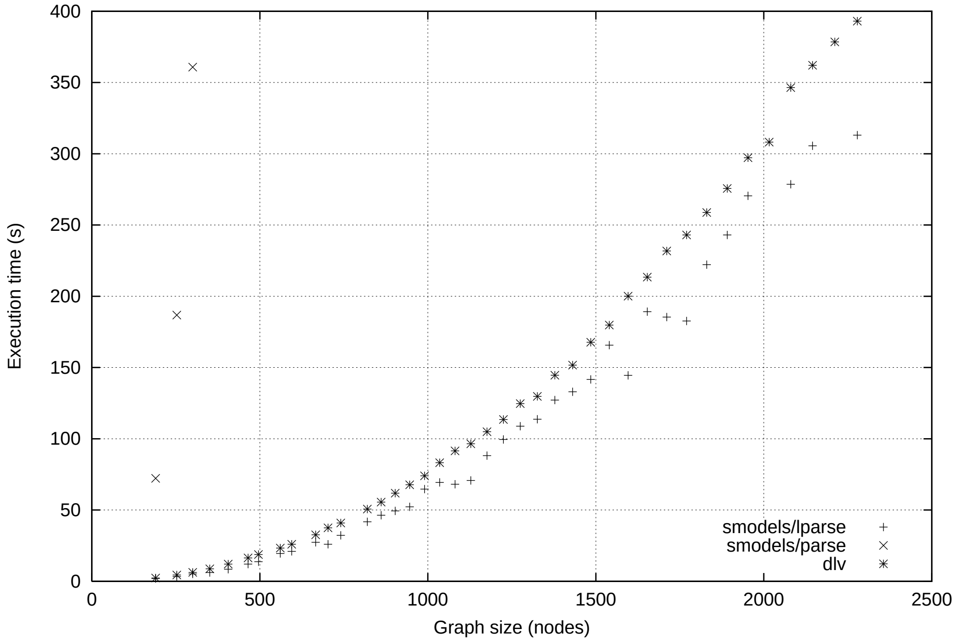

Figure 13: Results of graph 4-colorability for simplex graphs

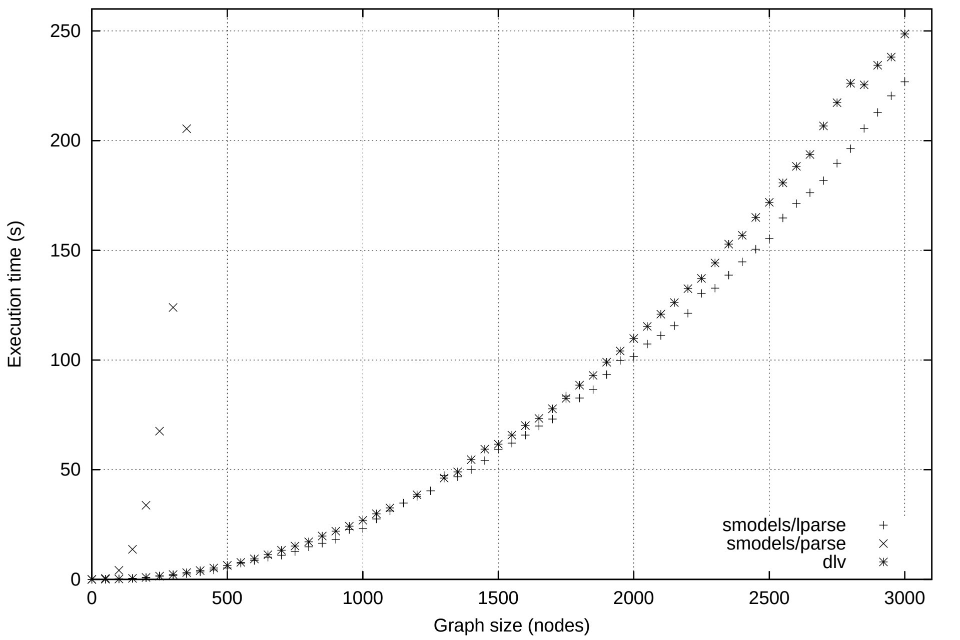

Figure 14: Results of graph 3-colorability for ladder graphs

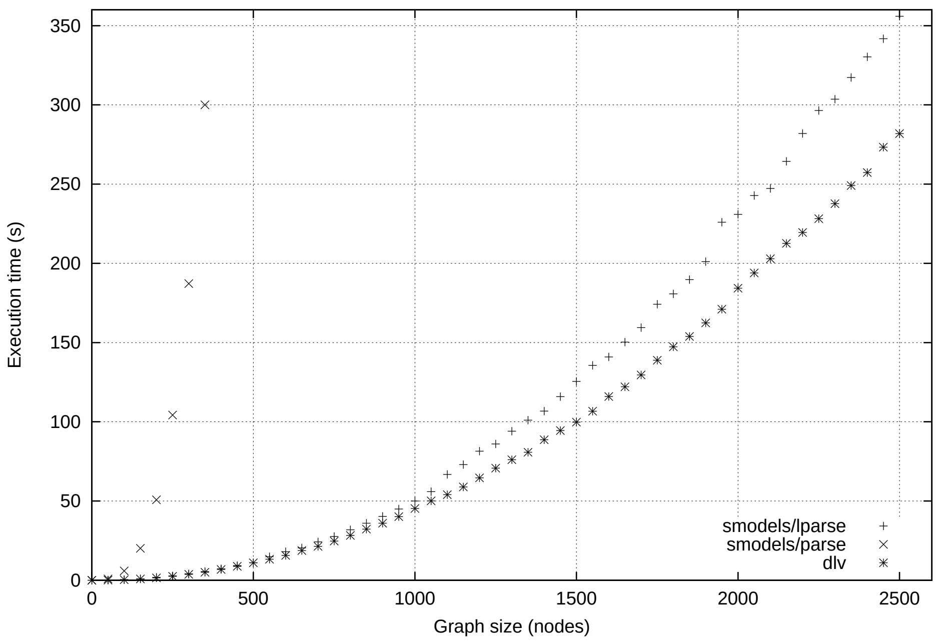

Figure 15: Results of graph 4-colorability for ladder graphs

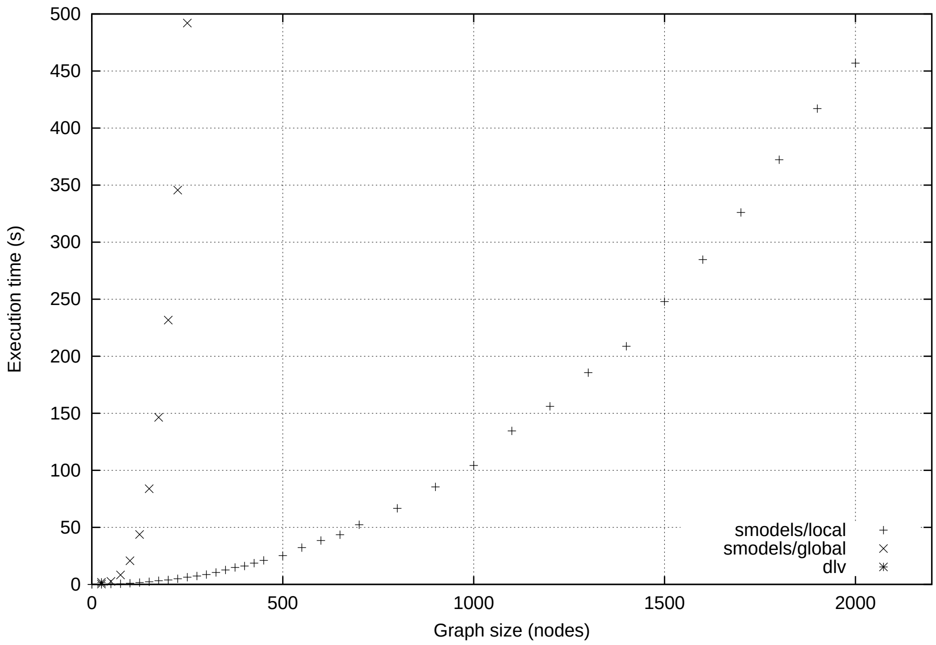

Figure 16: Results of graph 4-colorability for plane graphs

Problem	Blocks	Length
bw-large.c	15	8
bw-large.d	17	9
bw-large.e	19	10

Table 2: Size of blocks world problems

reordering the rules of the ground program. This seems to suggest that the algorithm **smodels** is not entirely stable with regards to rule organization.

The random plane graphs proved to be the most difficult case for each system. Here **smodels** with **lparse** had the best performance being able to find 4-colorings for plane graphs up to size of 2000 nodes in a reasonable time. In this case `dlv` got surprisingly bad results. Even problems of size of 50 nodes took well over $1000s$ to solve.The rule reordering was tried also in this case, but it didn't help much: after swapping of the two rules of the program (26) `dlv` was able to find 4-coloring for one of the three 50-node graphs in less than 10 seconds but the two other graphs were still hard. It seems that the heuristics of `dlv` work quite well when problems have rigid inner structure but with random cases the heuristics breaks down.

For **parse** nearly all of the total execution time was spent grounding the program as **smodels** found models in a couple of seconds in each case. For **lparse** the time ratios were reversed as **lparse** grounded each program using under $5s$.

Blocks World. Three blocks world problems were solved by **smodels** with both grounders. `dlv` was not tested in blocks world domain, because the encoding for **smodels** would have required major restructuring to make it a valid `dlv` program. The sizes of the three problems are in Table 2.

The results of the test runs are shown in Table 3 and graphically in Figure 17.

The grounding of the test cases was clearly faster with local grounding than with the global one as **lparse** performed 13–23 times faster than **parse**. While in each case **parse** generated fewer atoms than **lparse**, the total size of the grounded program was smaller with local grounding. The execution of **smodels** for grounded programs was also faster with locally grounded programs but the difference being about 15-20% was not as remarkable. The total time spent with problem using **lparse** was about 58% of that spent using **parse**.

Graph	GTime	STime	Time	Atoms	Rules	Σ
bw-large.c						
lparse	4.0	40.8	44.8	11017	85978	186401
parse	54.5	52.7	107.2	7546	90097	639537
bw-large.d						
lparse	6.2	75.0	81.2	14641	133172	285997
parse	103.6	91.8	195.4	10499	139927	998287
bw-large.e						
lparse	9.0	147.7	156.7	18925	197666	421269
parse	212.7	171.4	384.1	14140	207929	1489637

Table 3: Results for blocks world problems.
GTime is time used in grounding process, STime is time **smodels** used searching model, and Time = Gtime + Stime. Atoms, Rules and Σ are the size of the grounding.

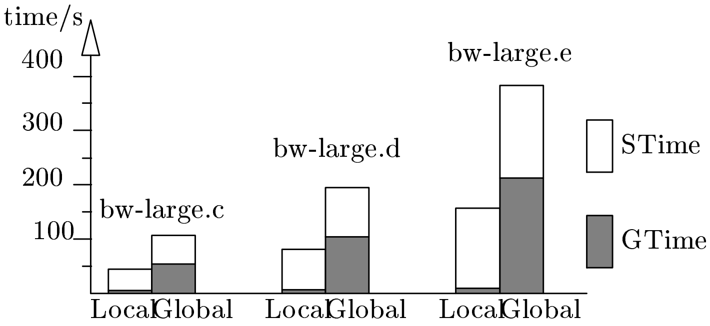

Figure 17: Execution times of blocks world problems.

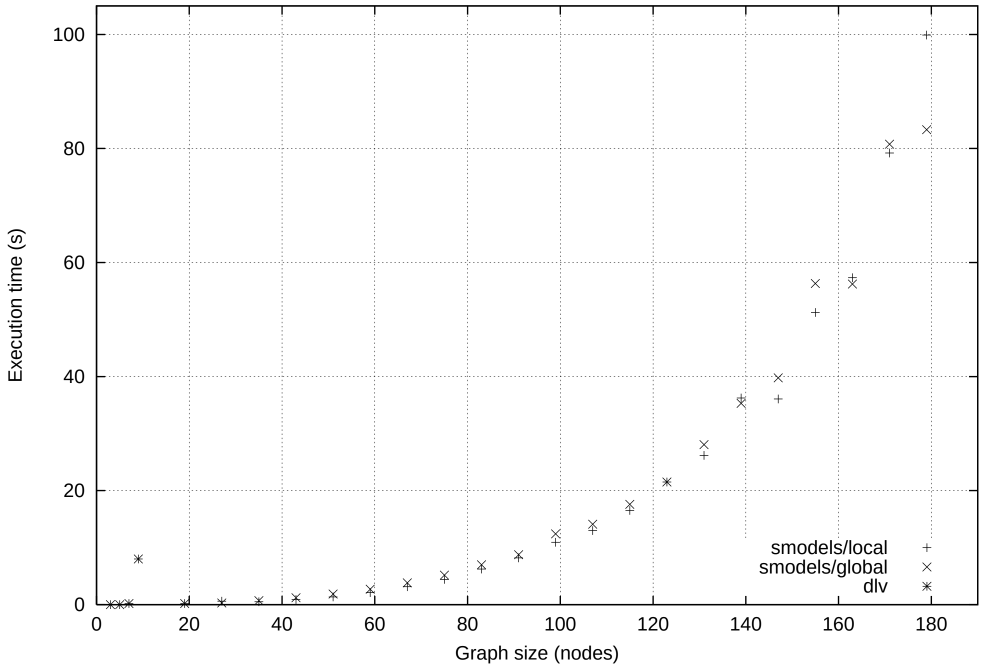

Figure 18: Finding one Hamilton circuit

Hamilton circuits. Hamilton circuits were tested with graphs (Figure 11) up to size of 31 nodes with 32768 circuits in case of finding all circuits and with graphs up to 179 nodes when only one circuit had to be found. The results are presented in Figures 18 and 19.

Here the results parallel the results of colorability of random plane graphs, with `dlv` having the poorest performance in both tests. However, in this test both **lparse** and **parse** had almost equal performance for finding one circuit. When searching all Hamilton circuits grounding prepared by **lparse** led to better results but here again the growth rates were practically identical.

Hamilton circuits also showed that the simplest encoding of the problem is not always the best. In the first encoding of program (27) the predicate *otherroute* was encoded as follows:

$$otherroute(V_1, V_2) \leftarrow vertex(V_1), vertex(V_2), vertex(V_3), hc(V_1, V_3), \mathrm{not}\, eq(V_1, V_2) \tag{28}$$

The resulting grounded programs were huge and had lots of unnecessary rules trying to fit Hamilton circuits between nodes without common edges. In fact, the performance was far worse than with global grounding. The key was to choose *arc* as a basic domain predicate instead of *vertex* and that way restrict the possible values of variables to those with actual edges between the nodes.

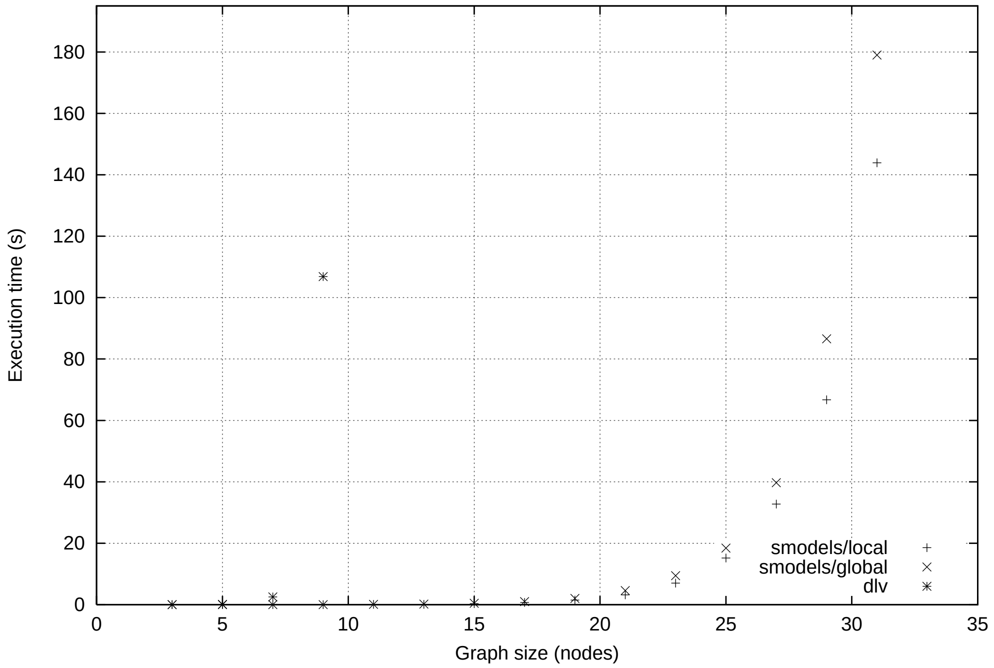

Figure 19: Finding all Hamilton circuits

The final rule was

$$otherroute(V1, V2) \leftarrow arc(V1, V2), hc(V1, V3), arc(V1, V3), \mathrm{not}\, eq(V2, V3). \tag{29}$$

6 Conclusions

In this work the class of strongly range restricted logic programs was defined and a local grounding algorithm **Ground**(P) was presented for them. An algorithm **GroundRule**$_2(R)$ was developed to efficiently ground strongly range restricted rules. The algorithm **Ground**(P) was shown to have linear space complexity with respect to the total size of the extensions of domain predicates. However, the number of ground instances of domain predicates was shown to be exponential in worst case. The worst case is often avoided in practice, and even if the local grounding has an exponential number of rules the size of domains may be only linear.

The performance of the combination of the **lparse** and **smodels** was compared with the **smodels**' global grounder **parse** and `dlv`-system. The results were favourable.

The three problems used in testing had all different characteristics as far as local grounding is considered. In the graph coloring there was only one join operation and it had only three (or four) element domain as one operand and no unnecessary rules were generated. The result was very fast and efficient grounding that improved performance of **smodels** by orders of magnitude.

The blocks world domain had a lot of joins and so was slower to ground than graphs. However, the grounding still took less than 10% of the total time in each case and the total execution time was smaller than the grounding time of **parse**. In addition, **smodels** processed the locally grounded program somewhat faster than globally grounded one. In the circumstances the result must be considered good.

In Hamilton circuit problems both **parse** and **lparse** grounded even the largest programs in few seconds and almost all time was spent searching for models. As total execution times were essentially identical, **lparse** didn't outperform **parse** like it did in other tests.

In conclusion the local grounding of strongly range restricted programs seems to be more effective than global grounding of range restricted programs. The actual savings vary from a problem to another, the problems having simple domain structure getting the most benefit.

References

[1] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the Association for Computing Machinery*, 43:20–74, 1 1996.

[2] Eiter, T., Leone, N., Pfeifer G., Mateis C., and Scarcello, F. The kr system `dlv`: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[3] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[4] K. Sagonas, T. Swift, and D.S. Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 274–289, Bonn, Germany, September 1996. The MIT Press.

[5] D.E. Knuth. The Stanford GraphBase, 1993. A collection of programs and data. Availabe by ftp from labrea.stanford.edu:pub/sgb.

[6] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.

[7] R.C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[8] I. Niemelä. Towards efficient default reasoning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 312–318, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[9] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79. Helsinki University of Technology, Digital Systems Laboratory, Research Report A52, May 1998.

[10] I. Niemelä and P. Simons. Evaluating an algorithm for default reasoning. In *Working Notes of the IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems, Montreal, Canada*, pages 66–72, Montreal, Canada, August 1995.

[11] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[12] Robert Sedgewick. *Algorithms in C.* Addison-Wesley Publishing Company, Inc, 1990.

[13] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report 47, Helsinki University of Technology, Helsinki, Finland, August 1997.

[14] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.

[15] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, July 1991.

A Grammar

$$
\begin{array}{rcl}
program & := & declarations \quad rules \\
declarations & := & declarations \quad declaration \\
 & | & e \\
declaration & := & \text{function} \quad identifier \quad . \\
 & | & \text{define} \quad symbol \quad symbol \quad . \\
 & | & \text{const} \quad identifier \quad . \\
rules & := & rules \quad rule \\
 & | & rules \quad compute\_stmt \\
 & | & e \\
rule & := & head \quad . \\
 & | & head \quad :- \quad tail \quad . \\
 & | & :- \quad tail \quad . \\
head & := & atom \\
 & | & head \quad | \quad atom \\
tail & := & basictail \\
 & | & weighttail \\
 & | & weighttail, basictail'.' \\
basictail & := & basictail \quad , \quad literal \\
 & | & literal \\
 & | & basictail \quad , \quad expr \\
 & | & expr \\
weighttail & := & constant \quad \{ \quad literal\_list \quad \} \\
literal & := & atom \\
 & | & \text{not} \quad atom \\
atom & := & identifier \\
 & | & identifier \quad (\quad arguments \quad) \\
arguments & := & arglist \\
 & | & arguments \quad ; \quad arglist \\
function & := & identifier \quad (\quad arglist \quad) \\
 & | & \text{not} \quad identifier \quad (\quad arglist \quad) \\
arglist & := & arglist \quad , \quad term \\
 & | & term \\
term & := & range \\
 & | & expr \\
constant & := & identifier \\
 & | & number \\
range & := & constant \quad .. \quad constant \\
 & | & constant \quad ,, \quad constant \\
expr & := & (\quad expr \quad) \\
 & | & '|' \quad expr \quad '|' \\
 & | & - \quad expr
\end{array}
$$

$$
\begin{array}{rcl}
 & | & expr \quad op \quad expr \\
 & | & variable \\
 & | & function \\
 & | & constant \\
compute\_stmt & := & \text{compute} \quad models \quad \{ \quad c\_list \quad \} \\
c\_list & := & l\_list \\
 & | & e \\
l\_list & := & l\_list \quad literal \\
 & | & literal \\
models & := & number \\
 & | & \text{all} \\
 & | & e \\
identifier & := & [a-z][A-Za-z0-9^{*}_{]} \\
op & := & == |!=|<|>|<=|>=|+|-|*|/|\text{mod} \\
symbol & := & identifier \quad | \quad op \quad | \quad '|''|' \\
number & := & [0-9]^{+} \\
variable & := & [A-Z][A-Za-z0-9^{*}_{]}
\end{array}
$$