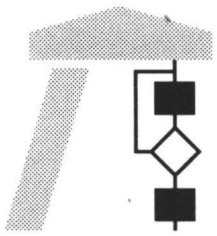
Interner Bericht

Implementation issues in Inductive Logic Programming

Robert Kolter

August 2003

325
~~352~~/03



FACHBEREICH
INFORMATIK

UNIVERSITÄT
KAISERSLAUTERN

Postfach 3049 · D-67653 Kaiserslautern

Implementation issues in Inductive Logic Programming

Robert Kolter

August 2003

325
~~352~~/03

Abstract

We propose several algorithms for efficient Testing of logical Implication in the case of ground objects. Because the problem of Testing a set of propositional formulas for (un)satisfiability is $\mathcal{NP}$-complete there's strong evidence that there exist examples for which every algorithm which solves the problem of testing for (un)satisfiability has a runtime that is exponential in the length of the input. So will have our algorithms. We will therefore point out classes of logic programs for which our algorithms have a lower runtime.
At the end of this paper we will give an outline of an algorithm for theory refinement which is based on the algorithms described above.

Keywords: Logic Programming, ILP, Theorem Proving

Contents

1 Introduction

Inductive Logic Programming (ILP) deals with the issue of constructing first order theories from given examples. Most of the time the examples are atomic ground formulas. Every example e for a theory T can either be a positive one or a negative one. In the first case $T \models e$, in the second case $T \not\models e$. The basic task in ILP is: given a set of positive examples

$$E^+ = \{e_1^+, \ldots, e_{n_1}^+\}$$

and a set of negative examples

$$E^- = \{e_1^-, \ldots, e_{n_2}^-\},$$

find a theory T with $T \models e_i^+$ for $i \in \{1, \ldots, n_1\}$ and $T \not\models e_i^-$ for $i \in \{1, \ldots, n_2\}$.

2 Preliminaries

We assume that the reader is familiar with the basic concepts of first order logic (we refer the reader to [17]). The definitions from logic programming which are necessary for the constructions will now be given. A *clause* is a finite disjunction of literals. A *horn-clause* is a clause with at most one positive literal. Clauses C can be seen as sets of literals: $C = \{L_1, \ldots, L_n\}$. So $C \equiv L_1 \vee \cdots \vee L_n$. In the special case where C is a horn-clause we have

$$\begin{aligned} C &= \{B, \neg A_1, \ldots, \neg A_{n-1}\} \\ &\equiv B \vee \bigvee_{i=1}^{n-1} \neg A_i \\ &\equiv B \vee \neg \bigwedge_{i=1}^{n-1} A_i \\ &\equiv B \vee \neg(A_1 \wedge \cdots \wedge A_{n-1}) \\ &\equiv B \leftarrow A_1 \wedge \cdots \wedge A_{n-1} \\ &=: B \leftarrow A_1, \ldots, A_{n-1} \end{aligned}$$

In a horn-clause C the element B is called the *head* of C and the set of elements A_i is called the *body* of C. In the special case where C consists of a single atom B we call C a *unit-clause*. If C consists only of negative atoms we call C a *goal* or a *query*. The clause which contains no atoms in the body and no head is denoted by $\square$. It is called the *empty clause*.

A *program clause* is a horn-clause which is no goal. A *logic program* P consists of a finite set of program clauses. For the sake of simplicity we will from now on call a logic program P simply a *program*. Every clause and every goal in logic programming is considered universally-quantified. The set of variables in a logic object O (a clause, a literal or an atom) is called $Var(O)$. An object which contains no variables is called *ground*. If $\{p_1 \ldots, p_l\}$ is the set of relation symbols which appear in a program P we define the *Herbrand-Base* HB_P for P

to be the set of all ground-instances of $p_1, \ldots, p_l$.
Given a program P and a goal $\leftarrow G$ the task in logic programming is to prove if $P \cup \{\leftarrow G\}$ is unsatisfiable. This is archieved by the application of SLD-Resolution (see [9]). If P is a program and $G = \leftarrow \psi_1, \ldots, \psi_m$ is a query, we assume that there is an index j with the property that the $j-$th query and the head of the $i-$th program clause are unifiable. Formally this means: If $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$ then a most general unifier $\sigma = \mathrm{mgu}(\psi_j, B_i)$ exists. The *resolvent* of G, P_i and σ is the new query

$$G' = \mathrm{Res}(G, P_i, \sigma) = \left(\leftarrow \psi_1, \ldots, \psi_{j-1}, A_1^{(i)}, \ldots, A_{n_i}^{(i)}, \psi_{j+1}, \ldots, , \psi_m\right)\sigma$$

We write $P \vdash_{\mathrm{SLD}} G'$ if $G' = \mathrm{Res}(G, P_i, \sigma)$ and $P \vdash^*_{\mathrm{SLD}} G'$ if there is an $n \in \mathbb{N}$ and goals $G_1, \ldots, G_n$ with $G_n = G$ and for all $i \in \{1, \ldots, n-1\}$ it holds that $G_{i+1} = \mathrm{res}(G_i, P_j, \mathrm{mgu}(G_i, P_j))$ for some j. Because SLD-Resolution is a sound and complete refutation procedure (see [1, 4]) $P \cup \{\leftarrow G\}$ is unsatisfiable iff $P \cup \{\neg G\} \models \square$ iff $P \cup \{\leftarrow G\} \vdash^*_{\mathrm{SLD}} \square$.
So if $G = G_1 \wedge \cdots \wedge G_k$, we have

$$P \cup \{\leftarrow G_1, \ldots G_k\} \models \square \quad \text{iff} \quad P \models \exists\left(G_1\left(\vec{x}^{(1)}\right) \wedge \cdots \wedge G_k\left(\vec{x}^{(k)}\right)\right)$$

where every tuple $\vec{x}^{(i)}$ is a subset of $Var(G_1) \cup \cdots \cup Var(G_k)$.
SLD-Resolution is an easy to understand procedure but if we want an implementation of it to be complete we have to be careful in applying the search strategy we use (fairness (see [10]) is crucial for the completeness). In the case of ground objects the situation is much easier as we will see in the next section.

3 BDDs and Groote's algorithm

Because of the fact that there are 2^n different input assignments for a propositional formula with n input variables, a realisation of boolean logic based on truth-tables has exponential runtime and storrage effort. So one has to search for a more efficient solution. In [3] BDDs are used for the representation of boolean formulas. Roughly speaking, BDDs are graphs with nodes which are labeled with variable indices and in which at every node a shannon-decomposition wrt. the variable with the index from the node is performed. For this let a first order language over the set F of function symbols, the set Pr of predicate symbols and the set $V = \{x_1, x_2, \ldots\}$ of variables be given. Then $\mathbb{P}(Pr, F, V)$ is the set of all atomic formulas over these sets (i.e. over the signature $sig = (F, Pr)$). We use the definition of BDDs given in [7] instead of Bryant's definition although our definition might look a little bit cumbersome.

Definition 1 *Given a boolean formula $f : \mathbb{B}^n \to \mathbb{B}$. A BDD for f is a directed, acyclig, node labeled graph (DAG) $B = \left(Q, l, \xrightarrow{\mathrm{f}}, \xrightarrow{\mathrm{t}}, s, 0, 1\right)$ with the following properties:*

- *Q is a finite set (the set of nodes),*
- *$l : Q \cup \{0,1\} \to \mathbb{P}(Pr, F, V) \cup \{0,1\}$ is a mapping with $l(0) = 0, l(1) = 1$ and $l(q) \notin \{0,1\}$ for all $q \in Q$ (the node labeling),*

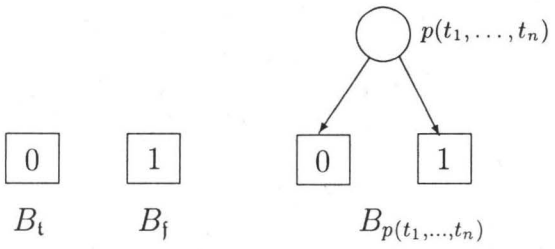


Figure 1: BDDs for basic functions

- $\xrightarrow{\mathfrak{t}}: Q \to Q \cup \{0,1\}$ *is the true continuation,*
- $\xrightarrow{\mathfrak{f}}: Q \to Q \cup \{0,1\}$ *is the false continuation,*
- $s \in Q \cup \{0,1\}$ *is the start node and*
- $0 \notin Q, 1 \notin Q$.

so that each path from the start node s to a terminal node which visits the node $q_1, \ldots, q_k$ *has the property that* $q_i \xrightarrow{\mathfrak{t}} q_{i+1}$ *iff the formula with which* q_i *is labeled is true wrt. to the selected interpretation.*

The symbols 0 and 1 are chosen to represent *false* and *true*. In [3] BDDs are used to represent propositional formulas. The definition we use is a little bit more general. In [3] nodes in a BDD are labeled with variables from V, where we want nodes to be labeled with names of atoms. All other aspects of BDDs carry over from [3] to our definition. Especially the fact that BDDs with a given ordering are a canonical form.
All BDDs can be built from some basic BDDs representing basic functions. It is a well known fact that the set $\{\neg, \wedge\}$ is a complete set of operations, i.e. every boolean function can be realized by using only these two operations. So for the realization of predicate logic formulas it suffices to give definitions for building BDDs for the *basic formulas* $\mathfrak{t}$ (*true*), $\mathfrak{f}$ (*false*) and all atoms of the form $p(t_1, \ldots, t_n)$ and describing how more complicated BDDs can be derived from simple ones. Figure 1 shows three kinds of BDDs: $B_{\mathfrak{t}}$ representing *true*, $B_{\mathfrak{f}}$, representing *false* and $B_{p(t_1,\ldots,t_n)}$ representing the atom $p(t_1, \ldots, t_n)$.
Given BDDs for functions f_1, f_2 we can build BDDs for functions which are composed from them. As we have mentioned above every logic formula can be expressed in the language that contains only the connectives $\neg$ and $\wedge$. So it suffices to give BDDs for $\neg\varphi$ and $\varphi \wedge \psi$ given BDDs forφ and ψ. The BDDs are depicted in figure 2.

We mention the following logical equivalences:

$$\begin{aligned}
\varphi \vee \psi &\equiv \neg(\neg\varphi \wedge \neg\psi) \\
\varphi \to \psi &\equiv \neg\varphi \vee \psi \\
&\equiv \neg(\varphi \wedge \neg\psi) \\
\varphi \leftrightarrow \psi &\equiv (\varphi \to \psi) \wedge (\psi \to \varphi) \\
&\equiv \neg(\varphi \wedge \neg\psi) \wedge \neg(\psi \wedge \neg\varphi)
\end{aligned}$$

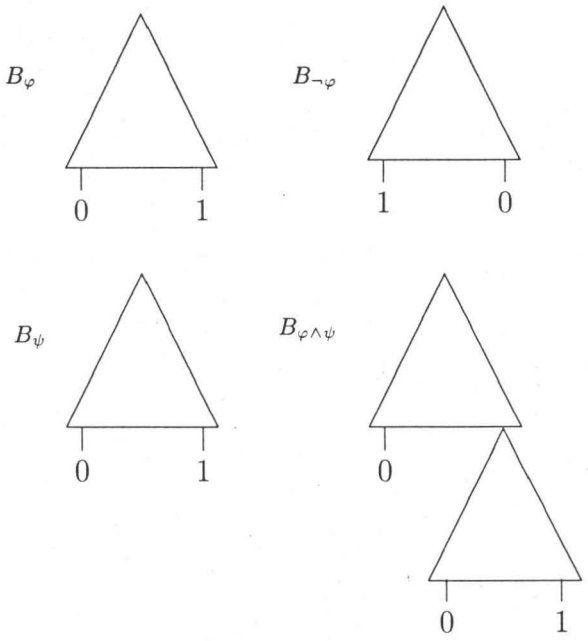


Figure 2: BDDs for composite functions

These equivalences allow us to create BDDs for formulas of arbitrary complexity.
Now we cite six Operators on BDDs from [7] which are used to reduce a given BDD (note that *reduce* might sound a bit misleading, because a reduced BDD might be larger in size than the initial BDD). The first two operators are only cited for the sake of completeness because they will never be applied *by hand* since most popular BDD-Packages handle BDDs in a way that these operators are automatically applied whenever this is necessary. These operators are the *Neglect-operator* and the *Join-operator*. Intuitively the Neglect Operator removes a node which is unnecessary in the sense that his true continuation and his false continuation are identical (in boolean algebra this means that the represented boolean function does not depend on the variable which is represented by the node). The Join-operator is a little bit more complicated. Its parameters are two nodes p and q with identical labels and identical true and false continuations. Then one of these nodes can be deleted. Formally we define the two operators as follows.

Definition 2 *Let* $B = \left(Q, l, \overset{\mathrm{f}}{\rightarrow}, \overset{\mathrm{t}}{\rightarrow}, s, 0, 1\right)$ *be a BDD. The Operators* N_p *and* $J_{p,q}$ *are definied as follows:*

1. *Let* p *be a node in* Q *with the property* $p \overset{\mathrm{t}}{\rightarrow} q, p \overset{\mathrm{f}}{\rightarrow} q$ *for some* $q \in Q$*. Then* $N_p(B)$ *is defined as*

$$N_p(B) = \left(Q', l, \overset{\mathrm{f}}{\rightarrow}', \overset{\mathrm{t}}{\rightarrow}', s', 0, 1\right)$$

where

- $\overset{\mathrm{f}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{f}}{\rightarrow} | r_1 \neq p, r_2 \neq p\right\} \cup \left\{\langle r_1, q\rangle | r_1 \overset{\mathrm{f}}{\rightarrow} q\right\}$
- $\overset{\mathrm{t}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{t}}{\rightarrow} | r_1 \neq p, r_2 \neq p\right\} \cup \left\{\langle r_1, q\rangle | r_1 \overset{\mathrm{t}}{\rightarrow} q\right\}$
- $s' = \begin{cases} s & \Leftrightarrow s \neq p \\ q & \Leftrightarrow s = p \end{cases}$
- $Q' = Q \setminus \{q\}$

2. *Let p and q be nodes in Q with $p \neq q, l(p) = l(q)$ and $p \overset{\mathrm{f}}{\rightarrow} r, p \overset{\mathrm{t}}{\rightarrow} r', q \overset{\mathrm{f}}{\rightarrow} r, q \overset{\mathrm{f}}{\rightarrow} r'$ for two nodes $r, r' \in Q$. Then $J_{p,q}$ is definied as*

$$J_{p,q}(B) = \left(Q', l, \overset{\mathrm{f}}{\rightarrow}', \overset{\mathrm{t}}{\rightarrow}', s', 0, 1\right)$$

where

- $\overset{\mathrm{f}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{f}}{\rightarrow} | r_2 \neq q\right\} \cup \left\{\langle r_1, p\rangle | r_1 \overset{\mathrm{f}}{\rightarrow} q\right\}$
- $\overset{\mathrm{t}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{t}}{\rightarrow} | r_2 \neq q\right\} \cup \left\{\langle r_1, p\rangle | r_1 \overset{\mathrm{t}}{\rightarrow} q\right\}$
- $s' = \begin{cases} s & \Leftrightarrow s \neq q \\ p & \Leftrightarrow s = q \end{cases}$
- $Q' = Q \setminus \{q\}$

As mentioned before the operators N_p and $J_{p,q}$ are implicitly used in most BDD-packages (see [18]). The next operators which we define have a more complicated semantics. This includes merging and sorting of BDDs. Here sorting takes place with respect to a total ordering $<$ on literals. At this point of the discussion $<$ is fixed. Given this ordering we want the BDD to have a strict ordering on every path. This if achieved by the application of the *Merge-* and *Sort-operators.*

Definition 3 *Let $B = \left(Q, l, \overset{\mathrm{f}}{\rightarrow}, \overset{\mathrm{t}}{\rightarrow}, s, 0, 1\right)$ be a BDD and let p and q be nodes from Q with $p \overset{\mathrm{f}}{\rightarrow} q$ and $l(p) = l(q)$. Then we define the* f*-merge-operator M_p^{f} as*

$$M_p^{\mathrm{f}}(B) = \left(Q', l, \overset{\mathrm{f}}{\rightarrow}', \overset{\mathrm{t}}{\rightarrow}, s, 0, 1\right)$$

with

- $\overset{\mathrm{f}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{f}}{\rightarrow} | r_1 \neq p \;\&\; r_2 \neq q\right\} \cup \left\{\langle p, q\rangle | q \overset{\mathrm{f}}{\rightarrow} r\right\}$
- *Q' contains only the nodes $q \in Q$ which are reachable from s.*

The t*-merge-operator M_p^{t} is defined as*

$$M_p^{\mathrm{t}}(B) = \left(Q', l, \overset{\mathrm{f}}{\rightarrow}, \overset{\mathrm{t}}{\rightarrow}', s, 0, 1\right)$$

with

- $\overset{\mathrm{t}}{\rightarrow}' = \left\{\langle r_1, r_2\rangle \in \overset{\mathrm{t}}{\rightarrow} | r_1 \neq p \;\&\; r_2 \neq q\right\} \cup \left\{\langle p, q\rangle | q \overset{\mathrm{t}}{\rightarrow} r\right\}$

- *Q' contains only the nodes $q \in Q$ which are reachable from s.*

These two operators allow us to check if there are two nodes p and q with the same label where one is a successor of the other. Applying these operators ensures that each label occurs at most once on every path. If this is the case we can order the BDD with respect to $<$. This is done by the application of the t− and f−sort-operators which are definied now.

Definition 4 *Let $B = \left(Q, l, \xrightarrow{\mathrm{f}}, \xrightarrow{\mathrm{t}}, s, 0, 1\right)$ be a BDD and let $<$ be a total ordering on $\mathbb{P}(Pr, F, V)$. Let p and q be nodes in Q with $p \xrightarrow{\mathrm{f}} q$ and $l(p) > l(q)$. Then we define the f−sort-operation S_p^{f} as*

$$S_p^{\mathrm{f}}(B) = \left(Q', l', \xrightarrow{\mathrm{f}}{}', \xrightarrow{\mathrm{t}}{}', s', 0, 1\right)$$

with

- $\xrightarrow{\mathrm{f}}{}' = \{\langle r_1, r_2\rangle \in \xrightarrow{\mathrm{f}} \mid r_1 \neq p \text{ or } r_2 \neq p\} \cup \left\{\langle r, p''\rangle | r \xrightarrow{\mathrm{f}} p\right\} \cup \{\langle p'', p\rangle\} \cup \left\{\langle p', r\rangle | q \xrightarrow{\mathrm{t}} r\right\} \cup \left\{\langle p, r\rangle | q \xrightarrow{\mathrm{f}} r\right\}$
- $\xrightarrow{\mathrm{t}}{}' = \left\{\langle r_1, r_2\rangle \in \xrightarrow{\mathrm{t}} | r_2 \neq p\right\} \cup \left\{\langle r, p''\rangle | r \xrightarrow{\mathrm{t}} p\right\} \cup \{\langle p'', p'\rangle\} \cup \left\{\langle p', r\rangle | p \xrightarrow{\mathrm{t}} r\right\}$
- $l'(r) = \begin{cases} l(p) & \Leftrightarrow r = p' \\ l(q) & \Leftrightarrow r = p'' \\ l(r) & \text{otherwise} \end{cases}$
- *Q' consists of the subset of reachable nodes of $Q \cup \{p', p''\}$.*

where p' and p'' are new nodes.

In analogy to S_p^{f} we define the operator S_p^{t}.

Definition 5 *Let $B = \left(Q, l, \xrightarrow{\mathrm{f}}, \xrightarrow{\mathrm{t}}, s, 0, 1\right)$ be a BDD and let $<$ be a total ordering on $\mathbb{P}(Pr, F, V)$. Let p and q be nodes in Q with $p \xrightarrow{\mathrm{f}} q$ and $l(p) > l(q)$. Then we define the t−sort-operation S_p^{t} as*

$$S_p^{\mathrm{t}}(B) = \left(Q', l', \xrightarrow{\mathrm{f}}{}', \xrightarrow{\mathrm{t}}{}', s', 0, 1\right)$$

with

- $\xrightarrow{\mathrm{t}}{}' = \{\langle r_1, r_2\rangle \in \xrightarrow{\mathrm{t}} \mid r_1 \neq p \text{ or } r_2 \neq p\} \cup \left\{\langle r, p''\rangle | r \xrightarrow{\mathrm{t}} p\right\} \cup \{\langle p'', p\rangle\} \cup \left\{\langle p', r\rangle | q \xrightarrow{\mathrm{f}} r\right\} \cup \left\{\langle p, r\rangle | q \xrightarrow{\mathrm{t}} r\right\}$
- $\xrightarrow{\mathrm{f}}{}' = \left\{\langle r_1, r_2\rangle \in \xrightarrow{\mathrm{f}} | r_2 \neq p\right\} \cup \left\{\langle r, p''\rangle | r \xrightarrow{\mathrm{f}} p\right\} \cup \{\langle p'', p'\rangle\} \cup \left\{\langle p', r\rangle | p \xrightarrow{\mathrm{f}} r\right\}$
- $l'(r) = \begin{cases} l(p) & \Leftrightarrow r = p' \\ l(q) & \Leftrightarrow r = p'' \\ l(r) & \text{otherwise} \end{cases}$

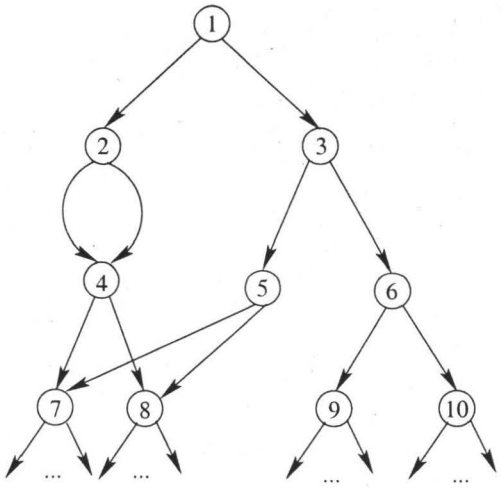


Figure 3: initial BDD

- *Q' consists of the subset of reachable nodes of $Q \cup \{p', p''\}$.*

where q' and q'' are new nodes.

These last two operators allow us *sort* the BDD in a way that each pair of nodes p, q with $p \xrightarrow{\mathrm{f}} q$ or $p \xrightarrow{\mathrm{t}} q$ has the property that $l(p) < l(q)$. So each path from the starting-node to a leaf visits nodes with ascending labels (wrt. $<$).

Example 1 *Assume for now that the operators N_p anf $J_{p,q}$ are not run automatically during the construction of a BDD. Now let the BDD B as depicted in figure 3 be given (we assume that the true-continuation of a node is represented as the node referenced by the right outgoing edge and the false-continuation of a node is referenced by the left outgoing edge).*
So we have $B = \left(Q, l, \xrightarrow{\mathrm{f}}, \xrightarrow{\mathrm{t}}, s, 0, 1\right)$ with

- $Q = \{1, 2, \ldots, 10, ...\}$
- $\xrightarrow{\mathrm{f}} = \{\langle 1,2\rangle, \langle 2,4\rangle, \langle 4,7\rangle, \langle 3,5\rangle, \langle 5,7\rangle, \langle 6,9\rangle, \ldots\}$
- $\xrightarrow{\mathrm{t}} = \{\langle 1,3\rangle, \langle 2,4\rangle, \langle 4,8\rangle\langle 3.6\rangle, \langle 5,8\rangle, \langle 6,10\rangle, \ldots\}$
- $s = 1$

Clearly the Neglect-operator can be applied to node 2 since $2 \xrightarrow{\mathrm{t}} 4$ and $2 \xrightarrow{\mathrm{f}} 4$.
After applying N_2 we have the situation as depicted in figure 4
Now the Join-operator can be applied to 4 and 5. Then we have the situation as depicted in figure 5.
After these steps our actual data looks like this:

- $\xrightarrow{\mathrm{f}} = \{\langle 1,4\rangle, \langle 4,7\rangle, \langle 3,4\rangle\langle 6,9\rangle, \ldots\}$
- $\xrightarrow{\mathrm{t}} = \{\langle 4,8\rangle, \langle 1,3\rangle, \langle 3,6\rangle\langle 6,10\rangle, \ldots\}$

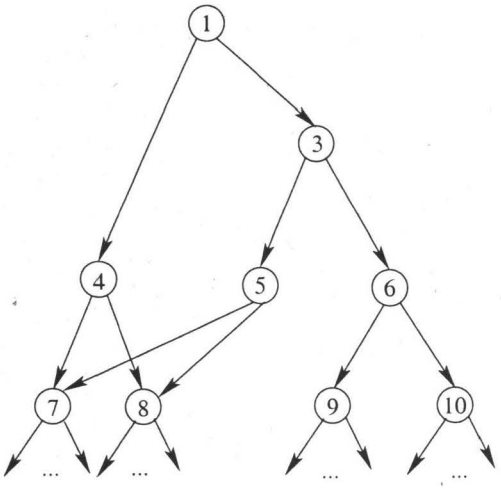


Figure 4: BDD after application of N_p

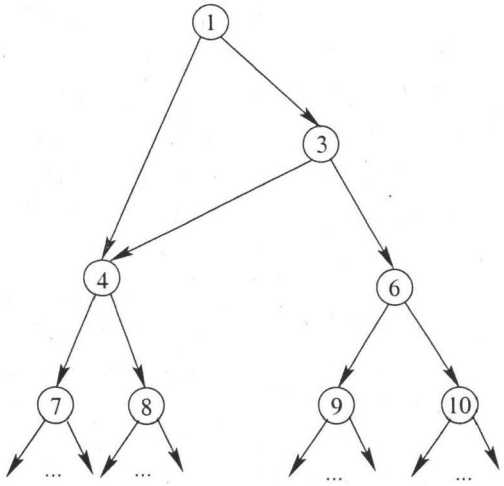


Figure 5: BDD after application of $J_{p,q}$

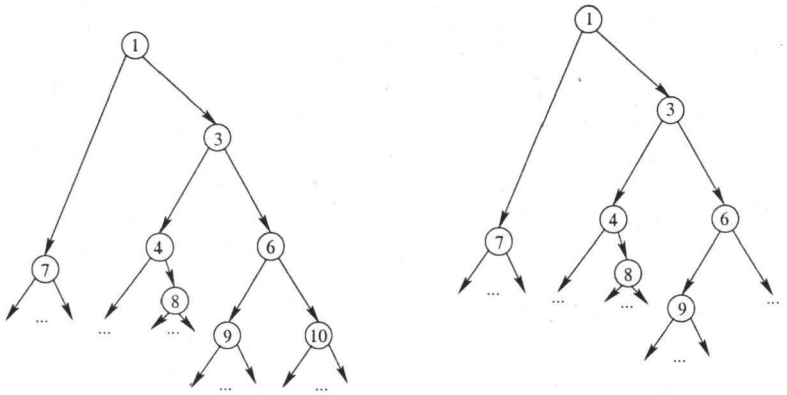


Figure 6: BDD after application of $M_p^{\mathfrak{f}}$ (left) and $M_p^{\mathfrak{t}}$ (right)

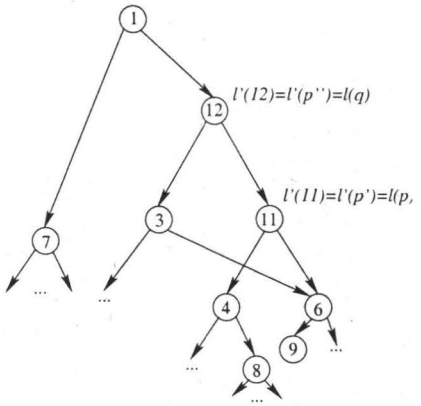


Figure 7: BDD after application of $M_p^{\mathfrak{f}}$ (left) and $M_p^{\mathfrak{t}}$ (right)

Assume now that $l(4) = l(7)$ *and* $l(6) = l(10)$. *Then the operators* $M_p^{\mathfrak{f}}$ *and* $M_p^{\mathfrak{t}}$ *can be applied to these nodes. This yields the BDDs in figure 6 (on the left we see the result of the application of* $M_4^{\mathfrak{f}}$, *on the right we see the result of* $M_6^{\mathfrak{t}}$*). Note that in the BDD on the right node* 10 *has been deleted because it was not reachable from the starting-node* s *after the application of the merge-operators. Assume now that* $l(3) > l(4)$. *Then the* $S_p^{\mathfrak{f}}-$*operator can be applied. This yields the final BDD as depicted in figure 7. Note that the nodes with the numbers* 11 *and* 12 *correspond with the nodes* p' *and* p''.

In [7] BDDs are called *reduced* if none of the operators introduced above can be applied. The reduced form of a BDD B is denoted with $R(B)$. There it is also proved that reduced BDDs are a canonical forml, i.e. for a fixed ordering $<$ and every formula φ B_φ is uniquely determined up to renaming of variables. Now we demonstrate how reduced BDDs can be used for proving logical implication. Let Φ be a finite set of universally quantified formulas. We want to prove that a skolemized formula φ is a logical consequence of Φ, i.e. $\Phi \models \varphi$.
We need some more definitions. Recall that a *substitution* θ is a mapping from variables to terms, i.e. $\theta : V \to \mathbb{T}(F, V)$ where $\mathbb{T}(F, V)$ is the set of all terms which can be constructed with function symbols in F and variables in V. A

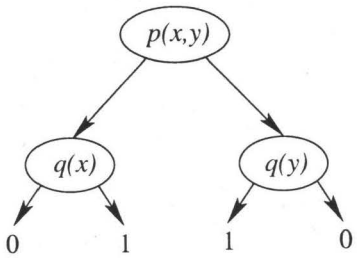


Figure 8: BDD B before copying

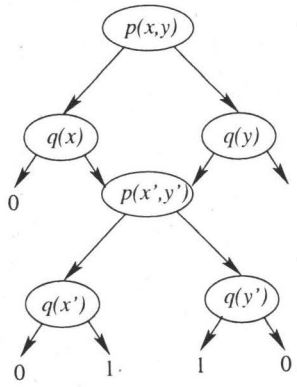


Figure 9: Result of $C(B)$

substitution θ is called a *unifier* of two literals $p(t_1, \ldots, t_n)$ and $q(u_1, \ldots, u_n)$ if $p(t_1, \ldots, t_n)\theta = q(u_1, \ldots, u_n)\theta$, i.e. if the application of θ to two (syntactical different) objects yields one and the same object. If such a unifier exists, the two literals are called *unifiable*. Unification is one of the basic tasks in automatic theorem proving and logic programming.
Let the BDD $B = \left(Q, l, \overset{\mathrm{f}}{\rightarrow}, \overset{\mathrm{t}}{\rightarrow}, s, 0, 1\right)$ be given. A variable $x \in V$ *occurs in* B if it occurs in at least one of the atoms labelling the nodes of B. If $x_1, \ldots, x_n$ is a set of variables, then $B\,[x_1 := x'_1, \ldots, x_n := x'_n]$ is the BDD that results from B if all occurences of the variable x_i are replaced by x'_i $(i = 1, \ldots, n)$.

Definition 6 *For a BDD B in which exactly the variables $x_1, \ldots, x_n$ occur the copy-operator $C(B)$ is defined as*

$$C(B) = B \wedge B\,[x_1 := x'_1, \ldots, x_n := x'_n]$$

where $x'_1, \ldots, x'_n$ are variables not occuring in B.

Example 2 *Assume that B is the BDD in figure 8.*
The only variables occuring in B are x and y. So $C(B)$ is the BDD in figure 9.

So we see that in general the copy-operator creates larger BDDs (except in the case that no variables occur in B).
Now let's go back to substitutions and unifiers. If θ is a substitution, then application of θ to the literals in the nodes of a BDD B yields again a BDD. This BDD is called $\theta(B)$.

Definition 7 *Let* $B = \left(Q, l, \overset{\mathfrak{f}}{\rightarrow}, \overset{\mathfrak{t}}{\rightarrow}, s, 0, 1\right)$ *be a BDD and let* $\theta : V \rightarrow \mathbb{T}(F, V)$ *be a substitution. Then* $\theta(B)$ *is the BDD defined as*

$$\theta(B) = \left(Q, \theta(l), \overset{\mathfrak{f}}{\rightarrow}, \overset{\mathfrak{t}}{\rightarrow}, s, 0, 1\right)$$

where $\theta(l)$ *denotes the mapping which maps each variable* x *to* $\theta(l(x))$.

We need some more details for the formulation of Groote's algorithm which we will enhance and modify. Assume that there is a path $p_0 \overset{\diamond_0}{\rightarrow} p_1 \overset{\diamond_1}{\rightarrow} \ldots \overset{\diamond_{n-1}}{\rightarrow} p_n \overset{\diamond_n}{\rightarrow} l \in \{0,1\}$ with $\diamond_i \in \{\mathfrak{t}, \mathfrak{f}\}$ for all i. Call this path

- *allowed* if there doesn't exist indices i, j with $i < j, l(i) = l(j)$ and $\diamond_i \neq \diamond_j$. Intuitively This means that there is no literal which has to take both truth values.
- *truth-truth-capable* if it is allowed and $\diamond_0 = \mathfrak{t}, l = 1$. That means: if the first literal in the path is evaluated to $\mathfrak{t}$, then there exists a possibility to evaluate the whole formula to $\mathfrak{t}$.

A unifier θ is called a unifier for a BDD B if θ unifies at least two literals with which the noeds in B are labelled. Now consider a unifier θ for a BDD B. This unifier is called *relevant* if there exists an allowed path

$$s = p_0 \overset{\diamond_0}{\rightarrow} p_1 \overset{\diamond_1}{\rightarrow} \ldots \overset{\diamond_{n-1}}{\rightarrow} p_n \overset{\diamond_n}{\rightarrow} 1$$

from the root node to the node 1 with the following property: If there exists an index i with $\diamond_i = \mathfrak{f}$, then the node p_i is not truth-truth-capable and there exists an index j with $\diamond_j = \mathfrak{t}$ and θ is a most general unifier for $l(i)$ and $l(j)$. In [7] it is proved that it suffices to concentrate on relevant unifiers if we want to prove that a formula is unsatisfiable. This fact will be exploited in Groote's algorithm.
Now let η be such a relevant unifier for B. We define an operator which manipulates a BDD by applying η.

Definition 8 *If* η *is a relevant unifier of* B, *the unification-operator* U_η *is defined by*

$$U_\eta(B) = \eta(B).$$

Note that $U_\eta(B)$ can never have more nodes than B if a *reasonable* BDD-representation is chosen (see [2]). Because of the definition of relevant unifiers these unifiers are easy to find in a BDD B. It suffices to check the rightmost path from s to 1.
Now we can present Groote's algorithm for testing a formula φ for unsatisfiability.
The following theorem holds ([7]).

Theorem 1 *Algorithm SATTEST stops with Input* φ *and Output unsatisfiable iff* $\varphi \equiv \mathfrak{f}$.

In SATTEST the desired property is checked for **every relevant unificator**. How can we find all these unifiers? In [7] the following lemma is proved:

Algorithm 1 Groote's algorithm

Input: φ in skolemized form.

proc SATTEST

1: $B = R(B_\varphi)$
2: **while** true **do**
3: RED(B)
4: $B \leftarrow R(C(B))$
5: **end while**

proc RED

1: **if** $B = B_\mathfrak{f}$ **then**
2: Stop! Output: *unsatisfiable*
3: **end if**
4: **for** all relevant unifiers θ of B **do**
5: RED($R(U_\theta(B))$)
6: **end for**

Lemma 1 *Let B be a reduced BDD. Then ξ is a relevant unifier for B iff*

$$p_o \overset{\diamond_0}{\to} p_1 \overset{\diamond_1}{\to} \ldots \overset{\diamond_{n-1}}{\to} p_n \overset{\diamond_n}{\to} 1$$

is the rightmost path in B and there exist i and $j, 0 \leq i, j \leq n$ with $\diamond_i = \mathfrak{f}, \diamond_j = \mathfrak{t}$ and ξ is a most general unifier of $l(p_i)$ and $l(p_j)$.

This lemma is the motivation for the next algorithm. Algorithm RELEVANT_UNIFIERS calculates the set of relevant unifiers.

Algorithm 2 Algorithm for the calculation of the relevant unifiers of B

Input: Reduced BDD B.

Output: Set S of relevant unifiers for B.

proc RELEVANT_UNIFIERS

1: $S \leftarrow \emptyset$
2: Let $p_o \overset{\diamond_0}{\to} p_1 \overset{\diamond_1}{\to} \ldots \overset{\diamond_{n-1}}{\to} p_n \overset{\diamond_n}{\to} 1$ be the rightmost path from $s = p_0$ to 1.
3: **for** $i = 1, \ldots, n$ **do**
4: **for** $j = 2, \ldots, n$ **do**
5: **if** $i \neq j$ **and** $\sigma = \mathrm{mgu}(l(p_i), l(p_j))$ **then**
6: $S \leftarrow S \cup \{\sigma\}$
7: **end if**
8: **end for**
9: **end for**
10: return S

4 Testing implication for ground objects in logic programs

Now we will concentrate on the application of a variant of Groote's algorithm on logic programs. First note that programs have quite simple BDDs. There's

no need to handle implication or equivalence expressions in the formulas.
We will now describe a detailed construction for the BDD of a program P. Let $P = \{P_1, \ldots, P_n\}$ be a program. We define the formula $\hat{P}$ as

$$\hat{P} := \bigwedge_{i=1}^{n} P_i$$

Every statement P_i in P is a horn-clause, so it has the form

$$P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$$

Then we get the following equivalence:

$$\begin{aligned}
\hat{P} &= \bigwedge_{i=1}^{n} P_i \\
&= \bigwedge_{i=1}^{n} B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)} \\
&\equiv \bigwedge_{i=1}^{n} B_i \vee \neg \left(A_1^{(i)} \wedge \cdots \wedge A_{n_i}^{(i)} \right) \\
&\equiv \bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}
\end{aligned}$$

Now let B be any element of HB_P. It is easily seen that $P \models B$ iff $\hat{P} \models B$. So we get

$$\begin{aligned}
P \models B \quad \text{iff} \quad & \hat{P} \models B \\
\text{iff} \quad & \bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \models B \\
\text{iff} \quad & \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \neg B \text{ is unsatisfiable} \\
\text{iff} \quad & \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \neg B \models \square
\end{aligned}$$

We now define the formula φ to be

$$\varphi := \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \neg B \tag{1}$$

Due to equation 1 we can prove the unsatisfiability of φ and therefore of $P \models B$ by constructing the BDD B_φ and applying Groote's algorithm.
The construction above can be easily generalized from ground atoms to ground formulas. Let ψ be the formula

$$\psi := \psi_1 \wedge \cdots \wedge \psi_m.$$

Then ψ can be seen as a query to the program P which contains of m single queries ψ_i. We get

$$
\begin{aligned}
P \models \psi \quad \text{iff} \quad & \hat{P} \models \psi \\
\text{iff} \quad & \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \neg\psi \text{ is unsatisfiable} \\
\text{iff} \quad & \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \neg\left(\psi_1 \wedge \cdots \wedge \psi_m\right) \text{ is unsatisfiable} \\
\text{iff} \quad & \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)} \right) \wedge \bigvee_{i=1}^{m} \neg\psi_i \models \square
\end{aligned}
$$

Let us illustrate this construction in an example.

Example 3 *Let P be the following program:*

$$
\begin{aligned}
p(x') &\leftarrow q(x'), r(x', x') \\
r(x'', x'') &\leftarrow q(x'') \\
q(a) &\leftarrow \\
q(b) &\leftarrow
\end{aligned}
$$

So we have

$$
\begin{aligned}
P &= (p(x') \leftarrow q(x'), r(x', x')) \wedge (r(x'', x'') \leftarrow q(x'')) \wedge q(a) \wedge q(b) \\
&\equiv \underbrace{(p(x') \vee \neg\left(q(x') \wedge r(x', x')\right))}_{=:C_1} \wedge \underbrace{(r(x'', x'') \vee \neg q(x''))}_{=:C_2} \wedge \underbrace{q(a)}_{=:C_3} \wedge \underbrace{q(b)}_{=:C_4}
\end{aligned}
$$

So if B denotes the BDD for P we have

$$B = B_{C_1 \wedge C_2 \wedge C_3 \wedge C_4} = B_{C_1} \wedge B_{C_2} \wedge B_{C_3} \wedge B_{C_4}$$

The single BDDs for the clauses are depicted in figure 10, the result of the composition in figure 11.
Assume we want to prove that $P \models p(a)$. Because $P \models p(a)$ iff $P \cup \{\neg p(a)\} \models \square$ we construct the BDD for $P \wedge \neg p(a)$. The result is shown in figure 12.

Formally a primitive algorithm for proving $P \models \psi$ looks as follows: Given Input P and ψ construct the BDD $B_{P \cup \{\leftarrow \psi\}}$ and apply Groote's algorithm. The formulation of this idea is given in algorithm 3.
The soundness of algorithm 3 is immediate from the soundness of SATTEST. PRIMITIVE_PROVER works fine in the case that indeed $P \models \psi$ holds. But what happens if $P \not\models \psi$? A nearly trivial fact is stated in the following lemma:

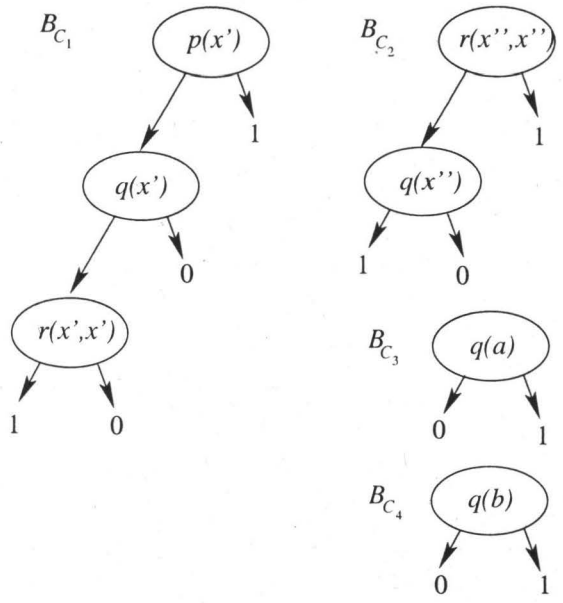


Figure 10: BDDs for $C_1, \ldots, C_4$

Algorithm 3 Checking-Algorithm, Version 1

Input: Program P, query $e \in HB_P$

proc PRIMITIVE_PROVER

1: Let $P \leftarrow \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$

2: Let $\varphi \leftarrow \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \neg e$

3: skolemize φ

4: call SATTEST(φ)

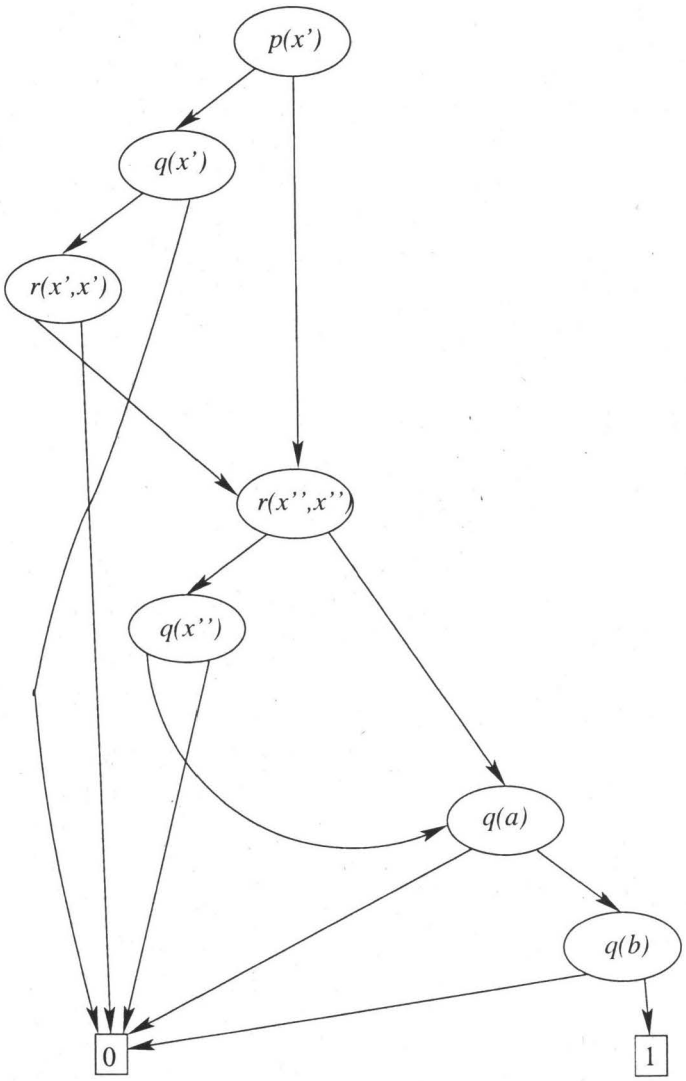


Figure 11: BDD for P

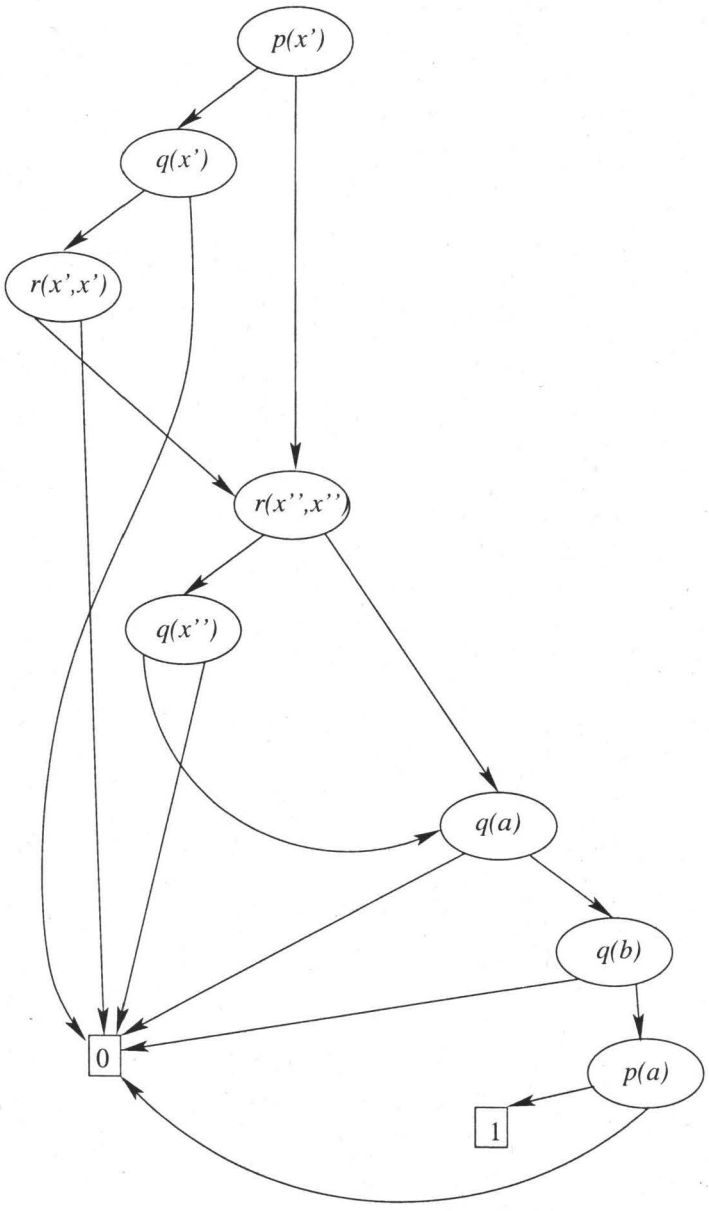


Figure 12: BDD which has to be reduced to B_{f}

Lemma 2 *Let P be a program and $\psi = \psi_1 \wedge \cdots \wedge \psi_m$ a query. If $P \not\models \psi$, then there exists at least on index i with $P \not\models \psi_i$.*

Proof. Assume that the claim does not hold. Then for all $i \in \{1, \ldots, m\}$ we have $P \models \psi_i$ and therefore $P \models \psi_1 \wedge \cdots \wedge \psi_m$ which implies $P \models \psi$, which contradicts the assumption. ♦

Since all queries ψ are conjunctions of ground atoms, we have that if i is such an index with $P \not\models \psi_i$ then $P \models \neg\psi_i$. This leads us to the problem of handling negation in our algorithm. As the problem of Testing $P \models \varphi$ is in general undecidable, we can't hope for an algorithm which proves that $P \not\models \psi_i$. But restriction on some *simpler* classes of formulas leads to the possibility to derive negative information and therefore prove $P \not\models \psi_i$. In many practical applications one uses a restricted form which can be handeled easier, namely the so called *negation as failure* (see [5]).
Negation as failure relies on a fairness constraint, i.e. the rule which choses literals in the goal to produce the SLD-derivation, has to be fair. This means, that there's no possible resolution-step which has to *wait* an infinitely long time. In other words: every resolution-step which is possible will be done after a finite amount of time.
Assume now that we have chosen a fair selection rule S. A *finitely failed* SLD-tree wrt. S is an SLD-tree which is constructed under S, which is finite and which has no success branches (i.e. it doesn't contain the empty clause $\square$). Assume further that FFS(P) is the set of all ground atoms A for which a finitely failed SLD-tree exists (regardless of the selection rule). The central result is stated in the following theorem.

Theorem 2 *The following conditions are equivalent:*

1. *$A \in FFS(P)$*
2. *Every fair SLD-tree which is rooted with $\leftarrow A$ is finitely failed.*

Now we refine algorithm 3 in a way that it can deal with negation as failure. First we prove the following lemmata. For this let a BDDs B and an SLD-tree S (see [10]) be given. We say that B and S are considered to be *isomorphic* if the sequence of BDDs constructed in algorithm 3 represents S, i.e. every refutation which can be found using B can also be found using S and vice versa. Especially we have that if S is finite, then B can either be reduced to B_f or there is a situation in algorithm 3 where no copy operation is possible because the set of relevant unifiers is empty. So we have the following lemma.

Lemma 3 *Let $B = B_{P \cup \{\leftarrow e\}}$. Then B is isomorphic to an SLD-tree rooted with $\leftarrow e$.*

Proof. This is an immediate consequence of the soundness and completeness of SATTEST. ♦

Lemma 4 *Let $B = B_{P \cup \{\leftarrow e\}} \neq B_\mathsf{f}$ be a reduced BDD for a program P and a goal e. If $S = RELEVANT\_UNIFIERS(B) = \emptyset$, then $e \in FFS(P)$.*

Proof. First note that SATTEST is a fair procedure, since **every** relevant unifier is applied if necessary. So $B_{P\cup\{\leftarrow e\}}$ is isomorphic to a fair SLD-tree rooted with $\leftarrow e$. To show that this SLD-tree is finitely failed. Assume the converse, i.e. $e \notin \mathrm{FFS}(P)$. Then there are two cases.

Case 1 "$P \models e$": Our assumption says that B is reduced and not equal to $B_\mathfrak{f}$. Since there are no relevant unifiers, SATTEST does **not** report, that $P \models e$, which is a contradiction to the completeness of SATTEST.

Case 2 "$P \not\models e$": Then there exists an infinite, fair SLD-derivation and therefore an infinite, fair SLD-tree. Due to theorem 2 the SLD-tree to which B is isomorphic is also infinite. So B must be infinite. But B can only grow, if a copy-operation is carried out. This cannot happen, since there are no relevant unifiers.

So we have a contradiction, i.e. the claim is proved. ♦

This sufficiently condition enables us to detect if an atom is in the finite failure set of a program P. However this condition is not necessary, which is a consequence of the undecidability of first order logic.

Algorithm 4 Checking-Algorithm, Version 2

Input: Program P, query $e \in HB_P$

proc IMPROVED_PROVER

1: Let $P = \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$
2: $\varphi \leftarrow \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge e$
3: skolemize φ
4: call SATTEST_2(φ)

proc SATTEST_2

1: $B \leftarrow R(B_\varphi)$
2: **while** true **do**
3: RED_2(B)
4: $B \leftarrow R(C(B))$
5: **end while**

proc RED_2

1: **if** $B = B_\mathfrak{f}$ **then**
2: Stop! Output: *unsatisfiable*
3: **else**
4: $S \leftarrow$ RELEVANT_UNIFIERS(B) /* We know: $B \neq B_\mathfrak{f}$*/
5: **if** $S = \emptyset$ **then**
6: Stop! Output: $e \in \mathrm{FFS}(P)$ /* Lemma 4 */
7: **else**
8: **for** all $\theta \in S$ **do**
9: call RED_2($R(U_\theta(B))$) /* eventually generates an infinite BDD */
10: **end for**
11: **end if**
12: **end if**

Theorem 3 *Given input P and $e \in HB_P$, algorithm IMPROVED_PROVER stops with*

1. *Output "unsatisfiable" if $P \models e$ and*
2. *Output "$e \in FFS(P)$" if $e \in FFS(P)$.*

Proof.

1. If $P \models e$, algorithm IMPROVED_PROVER acts exactly as algorithm PRIMITIVE_PROVER does. So the claim follows immediately from the soundness of PRIMITIVE_PROVER.
2. This a consequence of Lemma 4.

♦

As pointed out in the proof of theorem 3 the only possibility that algorithm 4 does not terminate is the generation of larger and larger BDDs, which shows that the function represented by the underlying program P is partial. Due to the undecidability of the halting problem there is no general way to recognize this situation. However there's a sufficient condition for the Nontermination of algorithm 4. In the sequel let $\mathbb{BDD}(Pr, F, V)$ be the set of all finite BDDs which can be built using predicate symbols from Pr, function symbols from F and variables from V. As every $\varphi \in \mathbb{P}(Pr, F, V)$ is a finite string and Pr, F and V are countable, so is $\mathbb{P}(Pr, F, V)$. Due to our assumption, every $B \in \mathbb{BDD}(Pr, F.V)$ contains only a finite number of nodes labeled with $l(\varphi)$ for some $\varphi \in \mathbb{P}(Pr, F, V)$, the set $\mathbb{BDD}(Pr, F, V)$ is also countable. That means that there is an injective function

$$\alpha : \mathbb{BDD}(Pr, F, V) \to \mathbb{N}$$

which maps a BDD to its unique index wrt α. This function α can be used to check algorithm IMPROVED_PROVER for nontermination. We assume that we have a table t where we save the $\alpha-$indices of the BDDs generated in a run of IMPROVED_PROVER (lines 3 and 4 in SATTEST_2). So t contains $\alpha(B_1), \ldots, \alpha(B_k)$ if B_k is the last BDD which has been generated.

Definition 9 *A BDD B contains a loop, if*

1. $B_k := B$,
2. $B_{k+1} = R(C(B_k))$ *and*
3. $\alpha(B_{k+1}) = \alpha(B_j)$ *for some* $j \in \{0, 1, \ldots, k\}$.

This definition leads to the following lemmata.

Lemma 5 *If $B = B_{P \cup \{\leftarrow e\}}$. contains loops then IMPROVED_PROVER does not terminate given input P and e.*

Proof. trivial. ♦

Lemma 6 *Let $B = B_{P \cup \{\leftarrow e\}}$ contain loops. Then $P \not\models e$.*

Proof. Assume $P \models e$. Then there's a sequence of relevant unifiers $\eta_1, \ldots, \eta_n$ such that $R(U_{\eta_0}(R(U_{\eta_1}(\ldots(R(U_{\eta_n}(B)))\ldots))) = B_\mathfrak{f}$ (see [7]). But in this case IMPROVED_PROVER stops and outputs that $P \cup \{\leftarrow e\}$ is unsatisfiable. This is a contradiction to lemma 5. So $P \not\models e$. ♦

The integration of lemma 5 and lemma 6 into algorithm 4 leads to algorithm 5.

Algorithm 5 Checking-Algorithm, Version 3

Input: Program P, query $e \in HB_P$

proc MORE_IMPROVED_PROVER

$t \leftarrow \emptyset$ /* Table for $\alpha-$values */
Let $P = \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$
$\varphi \leftarrow \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \neg e$
skolemize φ
call SATTEST_3(φ)

proc SATTEST_3

$B \leftarrow R(B_\varphi)$
$v \leftarrow \alpha(B)$
if $v \in t$ **then**
  Stop! Output: $P \not\models e$ /* Lemma 5, 6 */
end if
$t \leftarrow t \cup \{v\}$
while true **do**
  RED_3(B)
  $B \leftarrow R(C(B))$
end while

proc RED_3

if $B = B_\mathfrak{f}$ **then**
  Stop! Output: *unsatisfiable*
else
  $S \leftarrow$ RELEVANT_UNIFIERS(B)
  if $S = \emptyset$ **then**
    Stop! Output: $P \not\models e, e \in \mathrm{FFS}(P)$
  else
    for all $\theta \in S$ **do**
      call RED_3($R(U_\theta(B))$)
    end for
  end if
end if

Again this algorithm is sound.

Theorem 4 *Given input P and $e \in HB_P$ algorithm MORE_IMPROVED_PROVER stops with*

1. *Output "unsatisfiable" if $P \models e$,*
2. *Output "$P \not\models e, e \in FFS(P)$" if $e \in FFS(P)$ and*
3. *Output "$P \not\models e$" if $B_{P \cup \{\leftarrow e\}}$ contains a loop.*

Up to now we have only concentrated on goals $\leftarrow e$, i.e. queries which consist of a single atom. But in many practical applications one has to deal with multiple queries. So a query has the form $\leftarrow e_1, \ldots, e_n$. Our algorithms are also applicable to these kinds of queries with one limitation: in the case that algorithm 5 stops with output $P \not\models e$ we have to refine algorithm 5 in a way that it is able to find this e. If the situation is as sketched above, there is at least one i with $P \not\models e_i$. Algorithm 6 which is described below will output such an index i if it is detected. Algorithm 6 works as follows: As soon as we're in a situation where it is clear that we can't reduce B to $B_\mathfrak{f}$, we start searching for the atoms $e_1, \ldots, e_n$ in B. Because of our concentration on ground atoms, these atoms are contained in B without having been modified by the application of a unifier. So if we find an atom e_i wich the property that there is a node labeled with e_i in B, we return i.

Theorem 5 *If algorithm FINAL_PROVER given program P and query $e = e_1 \wedge \cdots \wedge e_m$ as input returns $i \in \mathbb{N}$, then $P \not\models e_i$.*

5 Complexity Issues

We will now have a look at the complexity of the algorithm FINAL_PROVER. This is a difficult task because it is possible that our algorithm does not terminate at all. So we have to concentrate on the case that it indeed terminates.
We assume for now that the signature with which we work, contains the constants 0 and 1 as well as an unary function-symbol s representing the successor-function for natural numbers. Each $n \in \mathbb{N}$ can then be realized by the term $s^n(0)$. Let P_1 be the program which defines the behaviour of the addition function $\mathtt{add} : \mathbb{N}^2 \to \mathbb{N}$. Then P_1 looks like this:

$$\begin{aligned} \mathtt{add}(x, 0, x) &\leftarrow \\ \mathtt{add}(x, s(y), z) &\leftarrow \mathtt{add}(x, y, z'), \mathtt{eq}(z, s(z')) \end{aligned}$$

Here we assume that `eq` is a relation which has the following interpretation: $\mathtt{eq}(t_1, t_2)$ holds if and only if t_1 and t_2 represent the same natural number. So `eq` allows us to deal with equality. A BDD for this program is shown in figure 13.
Now consider the multiplication function $\mathtt{mult} : \mathbb{N}^2 \to \mathbb{N}$. The definition of `mult` uses the definition of `add` as we see now:

$$\begin{aligned} \mathtt{mult}(x, 0, 0) &\leftarrow \\ \mathtt{mult}(x, 1, x) &\leftarrow \\ \mathtt{mult}(x, s(y), z) &\leftarrow \mathtt{mult}(x, y, z'), \mathtt{add}(x, z', z) \end{aligned}$$

A BDD for this program P_2 is shown in figure 14.
However, if we want the program to work as intended, we have to add the definition of `add` to the definition of `mult`. This yields the program $P = P_1 \cup P_2$. And this has also effects on the BDD for the multiplication function. Now we have $B_P = B_{P_1} \wedge B_{P_2}$. The resultant BDD is shown in figure 15
Let size(B) be the number of nonterminal nodes in a BDD B. Then the following theorem holds:

Algorithm 6 Checking-Algorithm for $P \models \psi$, final version

Input: Program P, query ψ

1: $t \leftarrow \emptyset$ /* Table for $\alpha-$values */
2: Let $P = \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$
3: Let $\psi = \psi_1 \wedge \cdots \wedge \psi_m$
4: $\varphi \leftarrow \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \bigvee_{i=1}^{m} \neg\psi_i$
5: skolemize φ
6: call SATTEST_4(φ)

proc SATTEST_4

1: $B \leftarrow R(B_\varphi)$
2: $v \leftarrow \alpha(B)$
3: **if** $v \in t$ **then**
4: **for** $i = 1, \ldots, m$ **do**
5: **if** ψ_i is found in B **then**
6: Stop! Output: $P \not\models \psi$
7: **end if**
8: **end for**
9: **end if**
10: $t \leftarrow t \cup \{v\}$
11: **while** true **do**
12: RED_4(B)
13: $B \leftarrow R(C(B))$
14: **end while**

proc RED_4

1: **if** $B = B_\mathrm{f}$ **then**
2: Stop! Output: *unsatisfiable*
3: **else**
4: $S \leftarrow$ RELEVANT_UNIFIERS(B)
5: **if** $S = \emptyset$ **then**
6: **for** $i = 1, \ldots, m$ **do**
7: **if** ψ_i is found in B **then**
8: Stop! Output: $P \not\models \psi_i, \psi_i \in \mathrm{FFS}(P)$
9: **end if**
10: **end for**
11: **else**
12: **for** all $\theta \in S$ **do**
13: call RED_4($R(U_\theta(B))$)
14: **end for**
15: **end if**
16: **end if**

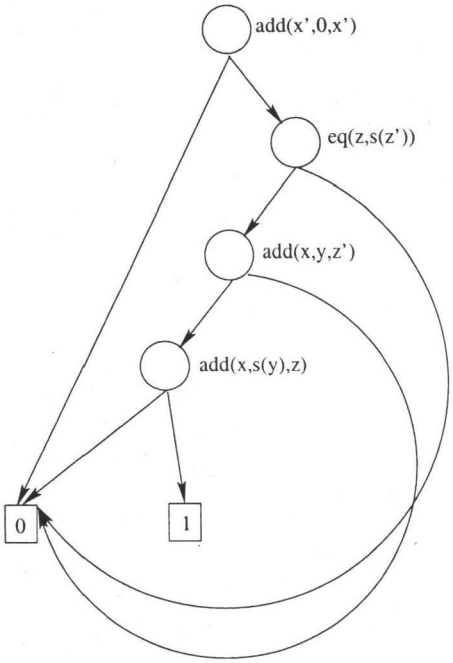


Figure 13: B_{P_1} : realization of the addition function

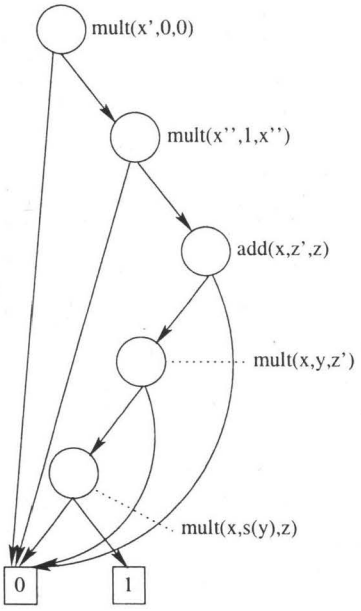


Figure 14: B_{P_2} : realization of the multiplication function

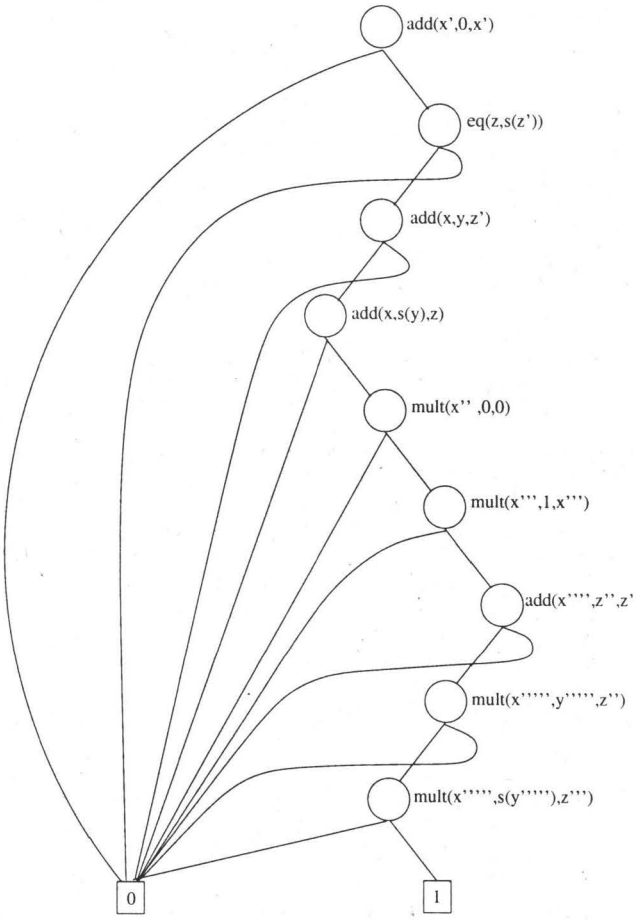


Figure 15: $B_P = B_{P_1} \wedge B_{P_2}$

Theorem 6 *If P_1 and P_2 have no variables in common and no two nodes n_1, n_2 with $n_1 \in B_{P_1}, n_2 \in B_{P_2}$ are labeled with the same ground literal, then*

$$size(B_{P_1 \wedge P_2}) = size(B_{P_1}) + size(B_{P_2})$$

Proof. The construction is straightforward. Let s_1 and s_2 be the starting nodes of B_{P_1} and B_{P_2}. The construction of $B_{P_1} \wedge B_{P_2}$ redirects every edge from a node $p \in B_{P_1}$ with $p \xrightarrow{\mathsf{f}} 1$ or $p \xrightarrow{\mathsf{t}} 1$ to $p \xrightarrow{\mathsf{f}} s_2$ ($p \xrightarrow{\mathsf{t}} s_2$). Since P_1 and P_2 have no variables and ground literals in common, none of the operators $N_p, J_{p,q}, M_p^{\mathsf{f}}, M_p^{\mathsf{t}}, S_p^{\mathsf{f}}$ and S_p^{f} can be applied. So the result is a BDD with size(B_{P_1}) + size(B_{P_2}) nodes. Furthermore $B_{P_1} \wedge B_{P_2}$ is reduced if B_{P_1} and B_{P_2} are reduced. ♦

Now assume that we allow *variable arities* for function and predicate symbols. This means that at least one symbol $p \in Pr$ (or $f \in F$) has no fixed arity. This is a ratonal assumption because in many practical problems the situation requires such symbols. Consider the example of summation: given n numbers $n_i \in \mathbb{N}$ for an arbitrary value of n, calculate the sum $\sum_{i=1}^{n} n_i$. Using only symbols of fixed arity we would have something like this in our program:

$$\begin{aligned}
\mathtt{sum}_1(x_1^{(1)}, x_1^{(1)}) &\leftarrow \\
\mathtt{sum}_2(x_1^{(2)}, x_2^{(2)}, z^{(2)}) &\leftarrow \mathtt{add}(x_1^{(2)}, x_2^{(2)}, z^{(2)}) \\
&\ldots \quad \ldots \quad \ldots \\
\mathtt{sum}_n(x_1^{(n)}, \ldots, x_n^{(n)}, z^{(n)}) &\leftarrow \mathtt{sum}_{n-1}(x_1^{(n)}, \ldots, x_{n-1}^{(n)}, z'), \mathtt{add}(z', x_n^{(n)}, z^{(n)})
\end{aligned}$$

So we need n function symbols for the realization of a program which calculates the sum of n numbers. However, if we allow variable arities, the program is much simpler:

$$\begin{aligned}
\mathtt{sum}(x_1^{(1)}, x_1^{(1)}) &\leftarrow \\
\mathtt{sum}(x_1^{(2)}, x_2^{(2)}, z^{(2)}) &\leftarrow \mathtt{add}(x_1^{(2)}, x_2^{(2)}, z^{(2)}) \\
&\ldots \quad \ldots \quad \ldots \\
\mathtt{sum}(x_1^{(n)}, \ldots, x_n^{(n)}, z^{(n)}) &\leftarrow \mathtt{sum}(x_1^{(n)}, \ldots, x_{n-1}^{(n)}, z'), \mathtt{add}(z', x_n^{(n)}, z^{(n)})
\end{aligned}$$

Although the number of clauses is the same, the program has a much simpler structure because in a SLD-refutation for a goal there are not so many literals which have to be checked for unifiability. In analogy to the program for summation, we can construct a program for multiplication. Here `prod` represents a predicate which is true if and only if the product of the first n arguments is equal to the $n+1-$st argument:

$$\begin{aligned}
\mathtt{prod}(x_1^{(1)}, x_1^{(1)}) &\leftarrow \\
\mathtt{prod}(x_1^{(2)}, x_2^{(2)}, z^{(2)}) &\leftarrow \mathtt{mult}(x_1^{(2)}, x_2^{(2)}, z^{(2)}) \\
&\ldots \quad \ldots \quad \ldots \\
\mathtt{prod}(x_1^{(n)}, \ldots, x_n^{(n)}, z^{(n)}) &\leftarrow \mathtt{prod}(x_1^{(n)}, \ldots, x_{n-1}^{(n)}, z'), \mathtt{mult}(z', x_n^{(n)}, z^{(n)})
\end{aligned}$$

Now assume that P is the program which consists of the definition of `add`, the definition of `mult` and the definition of the symbols `sum` and `mult` for an

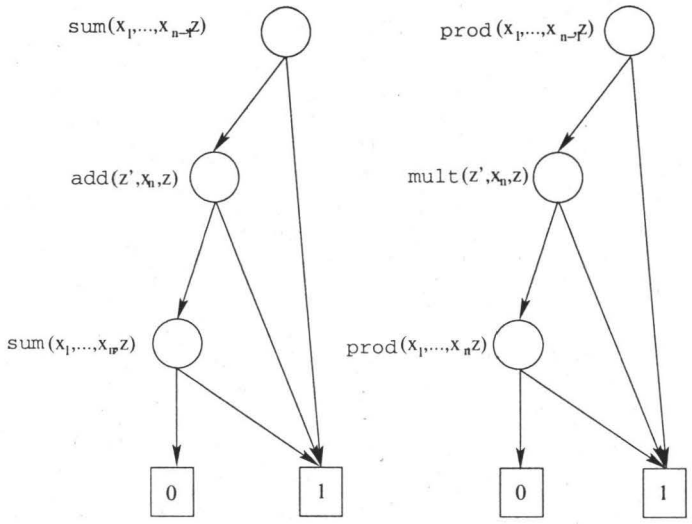


Figure 16: BDDs for P_n and S_n

arbitrary but fixed value of n. Then P is a program which can do addition and multiplication of up to n numbers. It is easily seen that all clauses in P can be standardized apart (i.e. no two different clauses in P have variables in common). So theorem 6 is applicable. We will use the following two abbreviations:

$$S_n = \mathtt{sum}(x_1^{(n)}, \ldots, x_n^{(n)}, z^{(n)}) \leftarrow \mathtt{sum}(x_1^{(n)}, \ldots, x_{n-1}^{(n)}, z'), \mathtt{add}(z', x_n^{(n)}, z^{(n)})$$
$$P_n = \mathtt{prod}(x_1^{(n)}, \ldots, x_n^{(n)}, z^{(n)}) \leftarrow \mathtt{prod}(x_1^{(n)}, \ldots, x_{n-1}^{(n)}, z'), \mathtt{mult}(z', x_n^{(n)}, z^{(n)})$$

First we will prove the following lemma:

Lemma 7 *For every $n \in \mathbb{N}$, the clauses S_n and P_n can be represented by BDDs of size 3.*

Proof. We have

$$\begin{aligned} S_n &= \mathtt{sum}(x_1, \ldots, x_n, z) \leftarrow \mathtt{sum}(x_1, \ldots, x_{n-1}, z'), \mathtt{add}(z', x_n, z) \\ &\equiv \neg\mathtt{sum}(x_1, \ldots, x_{n-1}, z') \vee \neg\mathtt{add}(z', x_n, z) \vee \mathtt{sum}(x_1, \ldots, x_n, z) \\ P_n &= \mathtt{prod}(x_1, \ldots, x_n, z) \leftarrow \mathtt{prod}(x_1, \ldots, x_{n-1}, z'), \mathtt{prod}(z', x_n, z) \\ &\equiv \neg\mathtt{prod}(x_1, \ldots, x_{n-1}, z') \vee \neg\mathtt{mult}(z', x_n, z) \vee \mathtt{prod}(x_1, \ldots, x_n, z) \end{aligned}$$

These formulas can be represented by the BDDs in figure 16. Their size is 3. So the claim is proved. ♦

Let P_1 and P_2 be the programs on page 5 which contain the definitions of `add` and `mult` and let $P^{\mathtt{add}} = \bigcup_{i=3}^{n} S_i$, $P^{\mathtt{mult}} = \bigcup_{i=3}^{n} P_i$. The following lemma is immediately.

Lemma 8 *Let $P_{1,1} = \{\mathtt{sum}(x, x) \leftarrow\}$, $P_{1,2} = \{\mathtt{sum}(x, y, z) \leftarrow \mathtt{add}(x, y, z)\}$, $P_{2,1} =$*

$\{\boldsymbol{prod}(x,x) \leftarrow\}$ *and* $P_{2,2} = \{\boldsymbol{prod}(x,y,z) \leftarrow \boldsymbol{mult}(x,y,z)\}$. *Then it is:*

$$\begin{aligned} \boldsymbol{size}(B_{P_{1,1}}) &= 1 \\ \boldsymbol{size}(B_{P_{1,2}}) &= 2 \\ \boldsymbol{size}(B_{P_{2,1}}) &= 1 \\ \boldsymbol{size}(B_{P_{2,2}}) &= 2 \end{aligned}$$

Therefore we have:

$$\begin{aligned} \text{size}(B_P) &= \text{size}(B_{P_1 \wedge P_2 \wedge P^{\text{add}} \wedge P^{\text{mult}}}) \\ &= \text{size}(B_{P_1}) + \text{size}(B_{P_2}) + \text{size}(B_{P^{\text{add}}}) + \text{size}(B_{P^{\text{mult}}}) \\ &= \text{size}(B_{P_{1,1}}) + \text{size}(B_{P_{1,2}}) + \text{size}(B_{P_{2,1}}) + \text{size}(B_{P_{2,2}}) \\ &\quad + \text{size}(B_{P^{\text{add}}}) + \text{size}(B_{P^{\text{mult}}}) \\ &= 6 + \text{size}(B_{P^{\text{add}}}) + \text{size}(B_{P^{\text{mult}}}) \\ &= 6 + \sum_{i=3}^{n} 3 + \sum_{i=3}^{n} 3 \\ &= 6 + 6(n-2) = 6n - 6 \in \mathcal{O}(n) \end{aligned}$$

So the size of the BDD for a program which realizes an $n-$ary adder/multiplier grows linear with the number of addends/multiplicand. This shows one of the advantages of the BDD for PL1 approach against the classical BDD approach, where it holds that the size of BDDs for $n-$Bit multipliers grows exponentially in the number of bits (see [3], there the lower bound $2^{\frac{n}{8}}$ for the BDD-size of an $n-$Bit multiplier is introduced; although exponential, the growth rate is not that bad for *realistic* sizes).
Let us now turn our view on another topic. As an immediate result of theorem 6 we can reason about the *complexity* of programs. Let P be a program and r a predicate symbol appearing in P. Then the *Definition* of r in P is the set $\text{DEF}_P(r)$ containing all clauses with r in their head. If S is a set of predicate symbols in P then we defint $\text{DEF}_P(S) := \bigcup_{r \in S} \text{DEF}_P(r)$. If we can ensure that the definitions of two or more relations are disjoint, the BDDs for these definitions can be constructed separately. For this let PRED_P be the set of all predicate symbols r with the property that $\text{DEF}_P(r) \neq \emptyset$.

Definition 10 *Let P be a program. We call P*

1. *basic if for every r it holds that $\text{DEF}_P(r)$ contains only unit clauses,*
2. *modular if there is a partition S_1, S_2 with $S_1 \cup S_2 = \text{DEF}_P(\text{PRED}_P)$ and $\text{PRED}_{S_1} \cap \text{PRED}_{S_2} = \emptyset$,*
3. *strong modular if there is a partition $S_1, \ldots, S_n$ with $n > 2$ and $\bigcup_{i=1}^{n} S_i = \text{DEF}_P(\text{PRED}_P)$ and $\text{PRED}_{S_i} \cap \text{PRED}_{S_j} = \emptyset$ for all i, j with $i \neq j$*
4. *hierarchical if there exists a mapping $\gamma : \mathbb{P}(Pr, F, V) \to \mathbb{N}$ (a level mapping) with the property that for all clauses $B \leftarrow A_1, \ldots, A_n$ $\gamma(A_i) < \gamma(B), i \in \{1, \ldots, n\}$.*

Some results follow immediately from this definition.

Lemma 9 *Every basic program is modular.*

Lemma 10 *Every strong modular program is modular.*

Proof. Let P be strong modular. Then there is a partition $S_1, \ldots, S_n$ with $n > 2$ and $\bigcup_{i=1}^{n} S_i = \text{DEF}_P(\text{PRED}_P)$. Set $S := \bigcup_{i=1}^{n-1} S_i, T := S_n$. Then (S,T) is a partition with $S \cup T = \text{DEF}_P(\text{PRED}_P)$. So P is modular. ♦

Lemma 11 *Every hierarchical program in which the predicates $r_1, \ldots, r_l$ occur with $\text{DEF}_P(r_i) \neq \emptyset$ for $i \in \{1, \ldots, l\}$ is modular.*

Proof. Let P be a hierarchical program. Then there is a level mapping γ as stated in definition 10. Due to the fact that P is a finite set of clauses and every clause is a finite set of literals, there is a finite set $M \subseteq \mathbb{N}$ with $M = \text{Ran}(\gamma)$. We define the partition $S_1, \ldots, S_m$ with $m = |M|$ by

$$C = B \leftarrow A_1, \ldots, A_n \in S_k \text{ iff } \gamma(B) = k \text{ for } k = 1, \ldots, m$$

We now show that $\bigcup_{i=1}^{m} S_i = \text{DEF}_P(\text{PRED}_P)$. Assume that this equality does not hold. Then we distinguish three cases.

Case 1 $\bigcup_{i=1}^{m} S_i \subset \text{DEF}_P(\text{PRED}_P)$: Then there is a clause $C = B \leftarrow A_1, \ldots, A_n$ with $C \notin S_i$ for all i. But P is hierarchicel, so we have $\gamma(A_j) < \gamma(B)$ for all j. So there is an l with $\gamma(B) = l \geq 1$ and $C \in S_l$. This is a contradiction.

Case 2 $\bigcup_{i=1}^{m} S_i \supset \text{DEF}_P(\text{PRED}_P)$: Then there is a clause $C = B \leftarrow A_1, \ldots, A_n$ with $C \in \bigcup_{i=1}^{m} S_i \setminus \text{DEF}_P(\text{PRED}_P)$. So there is a k with $C \in S_k$. But C is a hornclause, so there is an $r \in Pr$ and $t_1, \ldots, t_l \in \mathbb{T}(F,V)$ such that $B = r(t_1, \ldots, t_l)$. This implies $r(t_1, \ldots, t_l) = B \in \text{DEF}_P(r) \subseteq \text{DEF}_P(\text{PRED}_P)$, which gives the contradiction.

Case 3 $\bigcup_{i=1}^{m} S_i \# \text{DEF}_P(\text{PRED}_P)$: Then there are clauses

$$C_1 = B_1 \leftarrow A_1^{(1)}, \ldots, A_{n_1}^{(1)} \in \bigcup_{i=1}^{m} S_k \setminus \text{DEF}_P(\text{PRED}_P)$$

$$C_2 = B_2 \leftarrow A_1^{(2)}, \ldots, A_{n_2}^{(2)} \in \text{DEF}_P(\text{PRED}_P) \setminus \bigcup_{i=1}^{m} S_i.$$

But C_1 leads to the same contradiction as stated in case 2, while C_2 leads to the contradiction from case 1.

So the assumption, that $\bigcup_{i=1}^{m} S_i \neq \text{DEF}_P(\text{PRED}_P)$ leads to a contradiction. So P is modular. ♦

The reason why we concentrate on (strong) modular programs is quite simple. If a program P is (strong) modular, then B_P can be built incrementaly from the BDDs of the sets in the partition. This is usefull when we compute on machines with low memory size. Let us illustrate this idea: If P is modular, then a partition $S_1, \ldots, S_n$ as stated in definition 10 exists. Let $P_1 = S_1, \ldots, P_n = S_n$. Then we have

$$B_P = B_{P_1 \wedge \cdots \wedge P_n} = \bigwedge_{i=1}^{n} B_{P_i}$$

Many BDD packages create BDDs by using temporarily additional memory which is not needed in the final BDD. So we have to create B_{P_i} for $i = 1, \ldots, n$ and execute a run of a garbage collection-procedure. This minimizes the amount of memory used during the creation of B_P.

Definition 11 *Let $S = \{S_1, \ldots, S_n\}$ and $S' = \{S'_1, \ldots, S'_m\}$ be two partitions of the same program P. We call S' finer than S if $m \geq n$ and for all j there exists an i such that $j \leq i$ and $S'_i \subseteq S_j$. In this case we write $S' \preccurlyeq S$. We write $S' \prec S$ if $S' \preccurlyeq S$ and $S' \neq S$. Similarly we write $S \succcurlyeq S'$ if $S' \preccurlyeq S$ and $S' \succ S$ if $S' \succcurlyeq S$ and $S' \neq S$.*

Due to the fact that every program P is finite, there's a minimal element $S_{\min}$ wrt. $\preccurlyeq$. $S_{\min}$ is given as the partition in which every set S_i contains exactly the $i-$th clause from P.
We will now relate the order on partitions to provability. Therefore we introduce certain sets of clauses which we will call *minimal contradictive* wrt. a given goal. These minimal contradictive sets have the following property: If $\mathcal{U} \subseteq P$ is such a minimal contradictive set which is *unique*, then every partition $S' \prec \{\mathcal{U}, P \setminus \mathcal{U}\}$ contains only sets from which the associated goal cannot be proved. Formally:

Definition 12 *Let P be a program and $\leftarrow G$ be a goal such that $P \cup \{\leftarrow G\} \models \square$. Then a set $\mathcal{U} \subseteq P$ is called minimal contradictive wrt. $\leftarrow G$ if $\mathcal{U} \cup \{\leftarrow G\} \models \square$ and $\mathcal{V} \cup \{\leftarrow G\} \not\models \square$ for all $\mathcal{V} \subseteq P$ with $|\mathcal{V}| < |\mathcal{U}|$.*

Of course such a minimal contradictive set always exists. The more interesting question is: is this set is *always* unique? Or can P and $\leftarrow G$ possess more than one such minimal contradictive set $\mathcal{U}$? The negative answer is given in the following theorem.

Theorem 7 *Let P be a program and $\leftarrow G$ be a goal with $P \cup \{\leftarrow G\} \models \square$. Then a minimal contradictive set $\mathcal{U}$ needs not to be unique.*

Proof. Consider the program P defined by

$$\begin{aligned} p(a) &\leftarrow \\ p(x) &\leftarrow \end{aligned}$$

and the goal $\leftarrow p(a)$. Then clearly $P \cup \{\leftarrow p(a)\} \models \square$ holds. But $\mathcal{U} = \{p(a) \leftarrow\}$ and $\mathcal{V} = \{p(x) \leftarrow\}$ are two different minimal contradictive sets. ♦

Another property of minimal contradictive sets which we might expect is that if $\mathcal{U}$ is minimal contradictive wrt. P and $\leftarrow G$ then $P \setminus \mathcal{U} \cup \{\leftarrow G\} \not\models \square$. However this does not hold as example 4 shows.

Example 4 *Consider the Program P with*

$$\begin{aligned} p(x) &\leftarrow \\ q(a) &\leftarrow \\ p(a) &\leftarrow \end{aligned}$$

and the goal $\leftarrow p(a)$. *Then clearly* $\mathcal{U} = \{p(a) \leftarrow\}$ *is minimal contradictive wrt.* P *and* $\leftarrow p(a)$. *But nevertheless* $P \setminus \mathcal{U} \cup \{\leftarrow p(a)\} \models \square$.

The program P constructed in the proof of theorem 7 has the property that the clause $p(a) \leftarrow$ is an instance of the other clause $p(x) \leftarrow$. When we remove the clause $p(a) \leftarrow$ we get a program P' with the property that $P' \cup \{\leftarrow p(a)\} \models \square$ still holds and the minimal contradictive set $\mathcal{U}$ is unique. This is the motivation of the next definition.

Definition 13 *Let* $P = \{P_1, \ldots, P_n\}$ *be a program. Then* P *is weak relevant if*

- *for all* i, j *with* $i \neq j$ *it holds that* $P_i \neq P_j$ *and*
- *there's no pair* (i, j) *with* $i \neq j$ *and* P_i *is an instance of* P_j.

So P is weak relevant if it contains no two equal clauses and no clause can be derived from another clause by application of a substitution.
Another further characterizations of relevance are given in the following definitions.

Definition 14 *Let* $P = \{P_1, \ldots, P_n\}$ *be a program. Then* P *is called strong relevant if there's no* i *such that* $P \setminus \{P_i\} \vdash_{SLD} P_i$.

Definition 15 *Let* $P = \{P_1, \ldots, P_n\}$ *be a program. Then* P *is called relevant if there's no* i *such that* $P \setminus \{P_i\} \vdash^*_{SLD} P_i$.

We have that relevance implies strong relevance and strong relevance implies weak relevance. But the converse does not hold as the counterexamples below show.

Example 5 *Consider the program* P *given by*

$$\begin{aligned} p(x) &\leftarrow q(x) \\ q(x) &\leftarrow r(x) \\ r(x) &\leftarrow s(x) \\ p(x) &\leftarrow s(x) \end{aligned}$$

Then $P \vdash^*_{SLD} p(x) \leftarrow s(x)$ *as depicted in the SLD-refutation in figure 17 So* $P \setminus \{p(x) \leftarrow s(x)\} \vdash^*_{SLD} p(x) \leftarrow s(x)$, *and hence* P *is not relevant. But the only clause which can be resolved in exactly one step is* $p(x) \leftarrow r(x) \notin P$. *So* P *is strong relevant.*

A simmilar construction proves that weak relevance does not imply strong relevance.

Example 6 *Consider the Program* P *given by*

$$\begin{aligned} p(x) &\leftarrow q(x) \\ q(x) &\leftarrow r(x) \\ p(x) &\leftarrow r(x) \end{aligned}$$

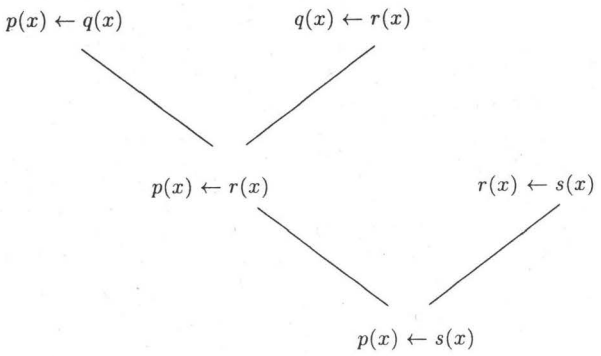


Figure 17: SLD-refutation

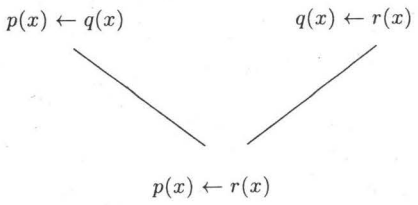


Figure 18: Another SLD-refutation

Then $P \setminus \{p(x) \leftarrow r(x)\} \vdash_{SLD} p(x) \leftarrow r(x)$ *(see figure 18) but no clause in* P *is an instance of another clause in* P*. So* P *is weak relevant but not strong relevant.*

We note that weak relevance is related to incomparabinity with respect to the subsumption order (see [14]). This order is defined as follows.

Definition 16 *Let* C *and* D *be clauses. Then* $C \succcurlyeq_s D$ *iff there is a substitution* σ *such that* $\sigma(C) \subseteq D$*. We write* $C \sim D$ *if* $C \succcurlyeq_s D$ *and* $D \succcurlyeq_s C$ *and* $C \succ_s D$ *if* $C \succcurlyeq_s D$ *and* $D \not\succcurlyeq_s C$*. Finally we write* $C * D$ *if* $C \not\succcurlyeq_s D$ *and* $D \not\succcurlyeq_s C$*.*

However, weak relevance is stronger than the subsumption order, so we need another criterion.

Definition 17 *A clause* C *is called reduced if there is no* $D \subset C$ *such that* $D \sim C$*.*

Reducedness allows us to link subsumption and weak relevance.

Lemma 12 *Let* P *be a program. Then* P *is weak relevant iff every clause in* P *is reduced and for all clauses* P_i, P_j *in* P *it holds that* $P_i * P_j$*.*

Proof.
$\Rightarrow$: clear by definition of weak relevance.
$\Leftarrow$: Assume that every clause in P is reduced and for all P_i, P_j in P we have that $P_i \ast P_j$. Assume further that there are indices i_0, j_0 and a substitution σ such that $\sigma(P_{i_0}) = P_{j_0}$. Then we have $P_{i_0} \succcurlyeq_s P_{j_0}$, which is a contradiction. ♦

If we restrict ourselves to goals which are ground, then we have an easy to prove sufficient condition for the uniqueness of the minimal contradictive set $\mathcal{U}$.

Lemma 13 *Let P be a program and $\leftarrow G$ a goal with only ground atoms in the body. If P consists only of unit clauses and it holds that for all i, j we have that $i \neq j$ implies $P_i \ast P_j$, then there is a unique minimal contradictive set $\mathcal{U} \subseteq P$.*

Proof. trivial. ♦

However this condition is a very strong restriction on programs P, which is of no use in practical applications. But if we try to relax the conditions just a little bit, we lose uniqueness of the minimal contradictive set as the next example shows.

Example 7 *Let P consist of the following clauses.*

$$\begin{aligned} p(a) &\leftarrow \\ p(b) &\leftarrow \end{aligned}$$

and consider the goal $\leftarrow p(x)$. Then $\mathcal{U}_1 = \{p(a) \leftarrow\}$ and $\mathcal{U}_2 = \{p(b) \leftarrow\}$ are two different minimal contradictive sets.

The above example might look a little bit frustrating but it gives an idea of a better criterion. For this we need the following two definitions.

Definition 18 *Let P be a program and let $\leftarrow G = \leftarrow p(t_1, \ldots, t_n)$ be a goal and let $\mathcal{U} \subseteq P$ be a set of clauses from P. If $P \models G$ then the pair $(P, \leftarrow G)$ is called $U-$determnistic if there is exactly one refutation of $P \cup \{\leftarrow G\} \models \square$ which uses exactly the clauses in $\mathcal{U}$.*

Definition 19 *Let P be a program and let $\leftarrow G = \leftarrow G_1, \ldots, G_n$ be a goal. Then* CANDIDATES(G_i, P) *is the set of clauses in P which have the property that their heads match the literal G_i. Formally:*

$$\text{CANDIDATES}(G_i, P) = \{A \leftarrow B_1, \ldots, B_m \in P | mgu(A, G_i) \text{ exists } \}$$

Using these two definitions we can prove a stronger criterion for uniqueness of minimal contradictive sets.

Theorem 8 *Let P be a program and let $\leftarrow G = \leftarrow p(t_1, \ldots, t_n)$ be a goal with $P \cup \{\leftarrow G\} \models \square$. Further assume that $|\text{CANDIDATES}(L, P)| \leq 1$ for every literal L. Then the minimal contradictive set $\mathcal{U} \subseteq P$ wrt. $(P, \leftarrow G)$ is unique iff $(P, \leftarrow G)$ is $P-$deterministic.*

Proof.
$\Leftarrow$: Let $(P, \leftarrow G)$ be $P-$deterministic. Then ther is at most one refutation for every goal. Since we asume that $P \cup \{\leftarrow G\} \models \square$, we can set $\mathcal{U}$ as the set of clauses which are used in this refutation. Due to the uniqueness of the refutation, $\mathcal{U}$ is also unique.
$\Rightarrow$: clear. ♦

However it is worth noticing that not only this criterion is undecidable. Much more: It is even undecidable if a set $\mathcal{U} \subseteq P$ is minimal contradictive or not. For this let MINCONT denote the problem of deciciding whether $\mathcal{U} \subseteq P$ is minimal contradictive wrt. $(P, \leftarrow G)$.

Theorem 9 MINCONT *is undecidable.*

Proof. Let P and $\leftarrow G$ be given. Assume that MINCONT is decidable. Then there is an algorithm $\mathcal{A}_{P,\leftarrow G}$ which accepts sets $\mathcal{U} \subseteq P$ with the property

$$\mathcal{A}_{P,\leftarrow G}(\mathcal{U}) = \begin{cases} 1 & \Leftrightarrow \mathcal{U} \text{ is minimal contradictive wrt. } (P, \leftarrow G) \\ 0 & \Leftrightarrow \text{else} \end{cases}$$

We use $\mathcal{A}_{P,\leftarrow G}$ to define an algorithm $\mathcal{B}_{P,\leftarrow G}$ which has the property:

$$\mathcal{B}_{P,\leftarrow G}(\mathcal{U}) = \begin{cases} 1 & \Leftrightarrow \mathcal{U} \models G \\ 0 & \Leftrightarrow \text{else} \end{cases}$$

Since the problem of deciding logical implication in first order logic is undecidable, this gives a contradiction.
$\mathcal{B}_{P,\leftarrow G}$ works as follows: First we call $\mathcal{A}_{P,\leftarrow G}(\mathcal{U})$. If $\mathcal{A}_{P,\leftarrow G}$ outputs 1, we output 1. If $\mathcal{A}_{P,\leftarrow G}$ outputs 0, we call $\mathcal{A}_{P,\leftarrow G}(\mathcal{V})$ for all $\mathcal{V} \subseteq \mathcal{U}$ with $|\mathcal{V}| \leq |\mathcal{U}|, \mathcal{V} \neq \mathcal{U}$. If $\mathcal{A}_{P,\leftarrow G}(\mathcal{V}) = 1$ for one such $\mathcal{V}$, we output 1 if $\mathcal{V} \subseteq \mathcal{U}$, 0 else. If $\mathcal{A}_{P,\leftarrow G}$ never outputs 1, we also return 0.
The correctness of $\mathcal{B}_{P,\leftarrow G}$ is immediately, because if $\mathcal{A}_{P,\leftarrow G}(\mathcal{U}) = 1$, we have that $\mathcal{U}$ is minimal contradictive and especially that $\mathcal{U} \cup \{\leftarrow G\} \models \square$. Else we know that $\mathcal{U}$ is not minimal contradictive. Then there are two possible cases:

Case 1 $\mathcal{U} \cup \{\leftarrow G\} \models \square$ but there is a $\mathcal{V}$ with $|\mathcal{V}| \leq |\mathcal{U}|$. This is checked by $\mathcal{B}_{P,\leftarrow G}$ and reported. If $\mathcal{V} \subseteq \mathcal{U}$ is reported to be minimal contradictive, we have $\mathcal{V}\,\sup\{\leftarrow G\} \models \square$, and therefore $\mathcal{U} \cup \{\leftarrow G\} \models \square$ by monotonicity.

Case 2 $\mathcal{U} \cup \{\leftarrow G\} \not\models \square$. Then there is no minimal contradictive subset $\mathcal{V} \subseteq \mathcal{U}$ and $\mathcal{B}_{P,\leftarrow G}$ outputs 0.

So we have $\mathcal{B}_{P,\leftarrow G}(\mathcal{U}) = 1$ iff $\mathcal{U} \cup \{\leftarrow G\} \models \square$ iff $\mathcal{U} \models G$. This is the desired contradiction. So the claim is proved and MINCONT is undecidable. ♦
Now we will try to relate the introduced concepts to the idea of modularity. Of course the following is obvious: Let P and $\leftarrow G$ be given and asume that $P \cup \{\leftarrow G\} \models \square$. If $\mathcal{U} \subseteq$ is unique and minimal contradictive, it suffices to apply algorithm 6 to $B_{\mathcal{U}}$ instead of B_P. Because of $\mathcal{U} \subseteq P$ we have $\text{size}(B_{\mathcal{U}}) \leq \text{size}(B_P)$. But due to theorem 9 this is undecidable. Furthermore P needs not to be (strong) modular, even if $\mathcal{U}$ is unique. The next example proves this.

Example 8 *Let P be the program*

$$\begin{array}{lll} p(a) & \leftarrow & \\ p(x) & \leftarrow & q(x) \\ q(a) & \leftarrow & \end{array}$$

and let the goal $\leftarrow p(a)$ be given. Then clearly $\mathcal{U} = \{p(a)\}$ is minimal contradictive and unique. But it is easily seen that P is not modular.

So now we will concentrate on the combination of the two topics *uniqueness of minimal contradictive sets* and *modularity*. First assume that P has a minimal contradictive set $\mathcal{U} \subseteq P$ wrt. $(G, \leftarrow P)$ for a goal $\leftarrow G$. Then $\mathcal{U} = \{C_1, \ldots, C_n\}$ for some n. Each C_i has the form

$$C_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$$

Therefore the following lemma holds:

Lemma 14 *For a horn-clause C the size B_C grows linear in the number of atoms in the body of C.*

Proof Clearly every horn-clause has the form $C = B \leftarrow A_1, \ldots, A_n \equiv B_i \vee \neg A_1 \vee \cdots \vee \neg A_n$ for some n. A simple proof by induction shows that $\text{size}(B_C) \leq n + 2$. First assume that $n = 0$. Then $C \equiv B$. Clearly the claim holds. Now assume the claim holds for some fixed $n \in \mathbb{N}$. Consider the clause $C = B \leftarrow A_1, \ldots, A_n, A_{n+1} \equiv B \vee \neg A_1 \vee \cdots \vee \neg A_n \vee \neg A_{n+1}$. It also holds that

$$B_C = B_B \wedge \bigwedge_{i=1}^{n+1} B_{\neg A_i} = \left(B_B \wedge \bigwedge_{i=1}^{n} B_{\neg A_i} \right) \wedge B_{\neg A_{n+1}}$$

So we have:

$$\begin{array}{rcl} \text{size}(B_C) & = & \text{size}\left(\left(B_B \wedge \bigwedge_{i=1}^{n} B_{\neg A_i} \right) \wedge B_{\neg A_{n+1}} \right) \\ & = & \text{size}\left(\left(B_B \wedge \bigwedge_{i=1}^{n} B_{\neg A_i} \right) \right) + \text{size}\left(B_{\neg A_{n+1}} \right) \\ & \leq & n + 2 + 1 = (n+1) + 2 \end{array}$$

♦

Using lemma 14 we can give an estimation for the size of $B_{\mathcal{U}}$. We have

$$B_{\mathcal{U}} = B_{C_1 \wedge \cdots \wedge C_n} = B_{C_1} \wedge \cdots \wedge B_{C_n} = \bigwedge_{i=1}^{n} B_{C_i}$$

So we get:

$$\begin{aligned}
\text{size}(B_{\mathcal{U}}) &= \sum_{i=1}^{n} \text{size}\,(B_{C_i}) \\
&\leq \sum_{i=1}^{n} n_i + 2 \\
&\leq \sum_{i=1}^{n} \underbrace{\max\{n_i \mid i = 1, \ldots, n\}}_{=:n_{\max}} + 2 \\
&= n \cdot (n_{\max} + 2) \in \mathcal{O}(n)
\end{aligned}$$

Let us now concentrate on the case that P is both strong modular and the minimal contradictive subset $\mathcal{U}$ is unique wrt. $(P, \leftarrow G)$ for al goal $\leftarrow G$. Then there is a partition $S = \{S_1, \ldots, S_n\}$ such that

- $\bigcup_{i=1}^{n} S_i = \text{DEF}_P(\text{PRED}_P)$
- $\bigcap_{i=1}^{n} \text{PRED}_{S_i} = \emptyset$

Two cases have to be distinguished:

Case 1 $\mathcal{U} \subseteq S_{i_0}$ for some $i_0 \in \{1, \ldots, n\}$. So the minimal contradictive subset is completely contained in one of the elements of the partition. Then clearly it holds that

$$\text{size}\,(B_{\mathcal{U}}) \leq \text{size}\,\left(B_{S_{i_0}}\right) < \text{size}\,(B_P)$$

Case 2 $\mathcal{U} \not\subseteq S_i$ for all $i \in \{1, \ldots, n\}$. Then we are able to chose a minimal set $S_{i_0}, \ldots, S_{i_m}\}$ of elements from S such that $\mathcal{U} \subseteq \bigcup_{j=0}^{m} S_{i_j}$. Unless $\{i_0, \ldots, i_m\} = \{1, \ldots, n\}$ we can still reach an improvement in the size of our BDD because it holds that

$$\text{size}\,(B_{\mathcal{U}}) \leq \sum_{j=0}^{m} B_{S_{i_j}} < \text{size}\,(B_P)$$

This proves the following lemma:

Lemma 15 *Let P be a strong modular program and let $\leftarrow G$ be a goal such that the minimal contradictive set $\mathcal{U} \subseteq P$ wrt. $(P, \leftarrow G)$ is unique. Then*

1. *$B_{\mathcal{U} \wedge \leftarrow G}$ is isomorphic to $B_{\mathfrak{f}}$ and*
2. *$\text{size}\,(B_{\mathcal{U}}) \leq \text{size}\,(B_P)$.*

6 Extensions

Up to now we have only considered *positive information*, i.e. we have presented algorithms which check for $e \in HB_P$ if $P \models e$. In ILP we will also have to deal with the dual case: does $P \not\models e$ hold? In general this is a very difficult task, but because of our restriction to ground objects it holds for every atom e that either $P \models e$ or $P \not\models e$. In the second case we have $P \not\models e$. This will be the basis

for the algorithm described in this section. Assume that a program P and a set of ground atoms $E = \{e_1, \ldots, e_n\}$ is given. We want to check if none of these atoms is a logical consequence of P. That means: we want to check if $P \not\models e_j$ holds for all j. Because all e_j are ground, this is equivalent to the question if for all j $P \models \neg e_j$ holds. So in total we have to check if $P \not\models e_1 \vee \cdots \vee e_n$ holds. We have:

$$\begin{aligned} & P \not\models e_1 \vee \cdots \vee e_n \\ \text{iff} \quad & P \models \neg (e_1 \vee \cdots \vee e_n) \\ \text{iff} \quad & P \models \neg e_1 \wedge \cdots \wedge \neg e_n \\ \text{iff} \quad & P \cup \{e_1 \vee \cdots \vee e_n\} \models \square \end{aligned}$$

So we see that we can apply the algorithms introduced in section 4. The only difference is that the BDD which is attached to B_P is now not a conjunction, but a disjunction of atoms.
So all theorems from the last sections also hold in this situation. Especially theorems 4 and 6.

7 Applications in ILP

As we have already mentioned in the introduction, we want to show how the algorithm described above can be used in ILP-systems. Most of the runtime of an ILP-system is consumed by testing the current theory for accuracy. That is: Does the current theory T imply all positive examples and does T imply no negative example? In the case of a positive answer, we have nothing to do. In the case of a negative answer, the theory has to be changed. Either it has to be strenghtened (when at least one negative example is implied) or it has to be weakened (when not every positive example is implied).
Assume now that at a positive example is not implied. Then we have the following situation: Our theory is T, our set of positive examples is $E^+ = \{e_1^+, \ldots, e_n^+\}$. Due to our assumption, the goal $\psi = \bigwedge_{i=1}^n e_i^+$ cannot be proved, i.e. $T \not\models \psi$. Since ψ is a conjunction of literals, at least one of these literals cannot be proved. So $\exists i : T \not\models e_i^+$.
Due to the construction of our algorithm it returns the index of such a literal which cannot be proved (in the case of finite failure). This information can be used in a refinement operator. A refinement operator is an operator which constructs a new theory from a given one, where the new theory is either stronger or weaker than the original one. Let us assume that such a refinement operator Θ is given. Θ has the following inputs: a theory T to be refined and some *hint* about the reason why T shall be refined. We will assume that these hints look as follows: a positive integer i shows that the $i-$th positive example is not implied while a negative integer $-j$ shows that the $j-$th negative example is implied. So if our refinement operator is accurate, we get $T' = \Theta(T, i)$ with the property that $T' \models e_i^+$ for all i and therefore $T' \models \bigwedge_{i=1}^n e_i^+ = \psi$. The algorithm below exploits this idea.

Algorithm 7 Checking-Algorithm for $P \not\models e_1 \vee \cdots \vee e_m$

Input: Program P, query $e_1, \ldots, e_m$

1: $t \leftarrow \emptyset$ /* Table for $\alpha-$values */
2: Let $P = \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$
3: Let $\psi = \psi_1 \wedge \cdots \wedge \psi_m$
4: $\varphi \leftarrow \left(\bigwedge_{i=1}^{n} B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \bigvee_{i=1}^{m} e_i$
5: skolemize φ
6: call SATTEST_5(φ)

proc SATTEST_5

1: $B \leftarrow R(B_\varphi)$
2: $v \leftarrow \alpha(B)$
3: **if** $v \in t$ **then**
4: **for** $i = 1, \ldots, m$ **do**
5: **if** e_i is found in B **then**
6: Stop! Output: $P \models e$
7: **end if**
8: **end for**
9: **end if**
10: $t \leftarrow t \cup \{v\}$
11: **while** true **do**
12: RED_5(B)
13: $B \leftarrow R(C(B))$
14: **end while**

proc RED_5

1: **if** $B = B_\mathrm{f}$ **then**
2: Stop! Output: *unsatisfiable*
3: **else**
4: $S \leftarrow$ RELEVANT_UNIFIERS(B)
5: **if** $S = \emptyset$ **then**
6: **for** $i = 1, \ldots, m$ **do**
7: **if** e_i is found in B **then**
8: Stop! Output: $P \models e_i$
9: **end if**
10: **end for**
11: **else**
12: **for** all $\theta \in S$ **do**
13: call RED_5($R(U_\theta(B))$)
14: **end for**
15: **end if**
16: **end if**

Algorithm 8 Refinement operator

Input: Program P, sets $\{e_1^+, \ldots, e_{m_1}^n\}, \{e_1^-, \ldots, e_{m_2}^-\}$ and a refinement operator Θ as described above.
We asume that $P = \{P_1, \ldots, P_n\}$ with $P_i = B_i \leftarrow A_1^{(i)}, \ldots, A_{n_i}^{(i)}$ for all i.

1: $\varphi_1 \leftarrow \left(\bigwedge_{i=1}^n B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \bigwedge_{i=1}^{m_1} e_i^+$
2: $\varphi_2 \leftarrow \left(\bigwedge_{i=1}^n B_i \vee \bigvee_{j=1}^{n_i} \neg A_j^{(i)}\right) \wedge \bigvee_{i=1}^{m_2} e_i^i$
3: apply FINAL_PROVER to B_{φ_1}
4: **if** FINAL_PROVER returns $i \in \mathbb{N}$ **then**
5: $\quad T \leftarrow \Theta(T, i)$
6: **end if**
7: apply FINAL_PROVER to B_{φ_2}
8: **if** FINAL_PROVER returns $j \in \mathbb{N}$ **then**
9: $\quad T \leftarrow \Theta(T, -j)$
10: **end if**
11: return T

8 Conclusions

We have presented some BDD-based algorithms for theorem proving in the case of horn-clause programs and ground goals. We have pointed out possibilities to improve their behaviour and runtime. We were able to present a theory refinement operator based on these algorithms.
What has not yet been captured is the question which ordering on the literals should be chosen. This question will be adapted in a future work.

References

[1] K.R. Apt, M.H. van Emden. *Contributions to the Theory of Logic Programming.* Journal of the ACM 29(3), 1984, pp. 841-862

[2] K.S. Brace, R.L. Rudell, R.E. Bryant. *Efficient Implementation of a BDD Package.* 27th ACM/IEEE Design Automation Conference 1990, pp. 40-45

[3] R.E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers 35(8), 1986, pp. 677-691

[4] M.H. van Emden, R.A. Kowalski. *The Semantics of Predicate Logic as a Programming Language.* Journal of the ACM 23(4), 1976, pp. 733-742

[5] K.L. Clark. *Negation as failure.* H. Gallaire, J. Minker (Eds.): Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'etudes et de recherches de Toulouse, 1977. Advances in Data Base Theory, Plemum Press, New York, 1978

[6] R. Drechsler, B. Becker. *Graphenbasierte Funktionsdarstellung.* Teubner Verlag, 1998

[7] J.F. GROOTE. *Binary decision diagrams for first order predicate logic.* Technical Report, Logic Group Preprint Series No. 127, Utrecht University, 1995.

[8] G. JÄGER, R.F. STÄRK. *The defining power of stratified and hierarchical logic programs.* Journal of Logic Programming, 15(1&2), 1993, pp. 55-77

[9] R.A. KOWALSKI, D. KUEHNER. *Linear Resolution with Selection Function.* Artificial Intelligence 2(3/4), 1971, pp. 227-260

[10] J.W. LLOYD.*Foundations of Logic Programming.* 2nd Edition, Springer Verlag 1987

[11] S. MUGGLETON, L. DE RAEDT. *Inductive Logic Programming: Theory and Methods.* Journal of Logic Programming 19/20, 1994, pp. 629-679

[12] S.-H. NIENHUYS-CHENG, R. DE WOLF. *Foundations of Inductive Logic Programming.* Springer Verlag, 1997

[13] M.S. PATERSON, M.N. WEGMAN. *Linear Unification.* Journal of Computer and System Sciences, 16, 1978, pp. 158-167

[14] G.D. PLOTKIN. *A Note on Inductive Generalization.* Machine Intelligence,5, 1970, pp. 153-163

[15] J.A. ROBINSON. *A Machine-Oriented Logic Based on the Resolution Principle.* Journal of the ACM 12(1), 1965, pp. 23-41

[16] H. ROGERS. *Theory of recursive functions and effective computability.* MIT Press, 1992

[17] H. SCHOLZ, G. HASENJAEGER. *Grundzüge der mathematischen Logik.* Springer Verlag, 1961

[18] F. SOMENZI. *CUDD: CU Decision Diagram Package Release 2.3.1.* University of Colorado at Boulder, 2001

[19] M. WIDERA, C. BEIERLE. *A Term Rewriting Scheme for Function Symbols with Variable Arity.* Informatik Berichte 280, FernUniversität Hagen, 2001