A non-ground realization of the stable and well-founded semantics

Georg Gottlob\textsuperscript{a},\textsuperscript{*}, Sherry Marcus\textsuperscript{b}, Anil Nerode\textsuperscript{c}, Gernot Salzer\textsuperscript{d}, V.S. Subrahmanian\textsuperscript{e}

\textsuperscript{a} Christian Doppler Labor für Expertensysteme, Technische Universität Wien, Paniglgasse 16, 1040 Wien, Austria
\textsuperscript{b} Mathematical Sciences Institute, Cornell University, Ithaca, NY 14853, USA
\textsuperscript{c} Department of Mathematics and Mathematical Sciences Institute, Cornell University, Ithaca, NY 14853 USA
\textsuperscript{d} Institut für Computersprachen, Technische Universität Wien, Karlsplatz 13, 1040 Wien, Austria
\textsuperscript{e} Institute for Advanced Computer Studies, Institute for Systems Research and Department of Computer Science, University of Maryland, College Park, MD 20742, USA

Received January 1994; revised June 1995
Communicated by H. Katsuno

Abstract

The declarative semantics of nonmonotonic logic programming has largely been based on propositional programs. However, the ground instantiation of a logic program may be very large, and likewise, a ground stable model may also be very large. We develop a non-ground semantic theory for non-monotonic logic programming. Its principal advantage is that stable models and well-founded models can be represented as sets of atoms, rather than as sets of ground atoms. A set $SI$ of atoms may be viewed as a compact representation of the Herbrand interpretation consisting of all ground instances of atoms in $SI$. We develop generalizations of the stable and well-founded semantics based on such non-ground interpretations $SI$. The key notions for our theory are those of covers and anticovers. A cover as well as its anticover are sets of substitutions - non-ground in general - representing all substitutions obtained by ground instantiating some substitution in the (anti)cover, with the additional requirement that each ground substitution is represented either by the cover or by the anticover, but not by both. We develop methods for computing anticovers for a given cover, show that membership in so-called optimal covers is decidable, and investigate the complexity in the Datalog case.

\footnote{Corresponding author. E-mail: gottlob@vexpert.dbai.tuwien.ac.at.}

\footnote{G. Gottlob's and G. Salzer's research was supported by the Christian Doppler Lab. The other authors were supported by the Army Research Office under grants DAAG-29-85-C-0018 and DAAL-03-92-G-0225, by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065 and by NSF grants IRI-91-09755 and IRI-93-57756.}

0304-3975/96/$15.00 \copyright$ 1996—Elsevier Science B.V. All rights reserved
SSDI 0304-3975(95)00207-3
1. Introduction

The declarative semantics of nonmonotonic logic programming methods has largely been based on propositional programs. For example, the stable models of a logic program are defined as certain Herbrand models of the ground instantiation of the program. The Gelfond–Lifschitz transform – playing a key rôle in defining both the stable semantics and the well-founded semantics for logic programming [2, 17] – works on ground instantiations as well. From a theoretical point of view, these semantics are satisfactory as they present a simple and clear way of describing the meaning of logic programs including various kinds of negation. When it comes to effectively computing these semantics e.g. in the context of databases, however, they prove to be inadequate. The reason is the combinatorial explosion taking place when forming all ground instances of a program, as required by both the stable and the well-founded semantics. As a consequence, the models derived from these ground programs may become very large.

Example 1. Let $P$ be the logic program

$$\begin{align*}
p(X, Y) & \leftarrow q(X, Y) \& \text{not}(r(b, Y)) \\
q(a, X) & \leftarrow \\
q(b, X) & \leftarrow \\
r(a, Z) & \leftarrow q(a, Z)
\end{align*}$$

and suppose we have a total of $n$ constants in the language (i.e., we have $a$, $b$ and another $(n - 2)$ constants). Then the ground version of the program contains $(n^2 + 3n)$ ground clauses. It has one stable model containing $5n$ atoms: all ground instances of the five atoms $r(a, X)$, $q(a, Y)$, $q(b, Z)$, $p(a, V)$ and $p(b, W)$ – and only those – are true in it ($X$, $Y$, $Z$ and $W$ are variable symbols).

As we see in the example, these large models can be characterized by just a few non-ground atoms. What are the advantages of such a non-ground representation? First, observe that we just need to store five non-ground atoms to capture the stable model instead of $5n$. As we will demonstrate later, this set of five atoms is a stable model in a new sense which allows non-ground atoms.

Second, this stable model can be computed directly from $P$ without grounding it. We only need to deal with the four clauses in $P$ (and a few more) rather than with $(n^2 + 3n)$ ground instances.

Last but not the least, if at a later point in time a new constant is introduced into the language, the non-ground representation of the stable model will not change, unless this new constant is known to affect one or more of the relations $p$, $q$ or $r$. The reason for this is that every instance of $p(b, W)$ is true in every stable model of $P$, independently of exactly what constant symbols occur in the language.
In this paper we use the work on S-semantics [7, 15] to develop a non-ground version of the Gelfond–Lifschitz transform. This enables us to define a non-ground stable model semantics and a non-ground well-founded semantics. In both cases, when we restrict our interest to ground instantiations of programs, it turns out that the resulting semantics coincides with the existing semantics for such programs. Furthermore, we show that these non-ground semantics have a number of nice properties, paralleling similar properties enjoyed by their ground counterparts. We also report on some computational aspects of various problems arising out of non-ground computations.

The organization of this paper is as follows. In Section 2, we briefly recall the ground Gelfond–Lifschitz transform as well as stable and well-founded models based on it; furthermore, we describe the S-semantics of Falaschi et al. [7] and set up basic notations concerning substitutions. In Section 3, we develop our non-ground version of the Gelfond–Lifschitz transform based on covers, leading in Section 4 to the introduction of non-ground stable models and the non-ground well-founded semantics. Section 5 gives another characterization of the non-ground Gelfond–Lifschitz transform based on bad sets and anticovers, which is – from a computational point of view – preferable to the first one. Properties of covers and anticovers are investigated in Section 6. This section also defines the notion of an optimal anticover and proves that optimal anticovers are recursive sets. Section 7 contains details on an improved anticover computation algorithm. In Section 8, we show how the methods in the previous section can, with minor changes, be used to develop a sound and complete algorithm for the computation of anticovers in the Datalog case. Furthermore, it is proved that in the Datalog case there is no output-polynomial algorithm for computing anticovers unless \( P = NP \). Our work on anticovers is closely related to the notion of disunification [15], which is discussed in Section 9. Finally, in Section 10, we show how the methods described in this paper apply to extended logic programs (i.e., logic programs containing both explicit as well as non-monotonic modes of negations).

2. Preliminaries

A logic program is a finite set of universally closed formulas

\[ A \leftarrow A_1 \land \cdots \land A_n \land \text{not}(B_1) \land \cdots \land \text{not}(B_m) \]

where each of \( A, A_1, \ldots, A_n, B_1, \ldots, B_m \) is an atom. The connective not stands for negation by failure. We use \( \text{grd}(P) \) to denote the set of all ground instances of clauses in \( P \). If \( C \) is a clause, \( C^+ \) denotes the negation-free clause obtained from \( C \) by deleting all atoms prefixed by not. Unless stated otherwise, the language associated with a logic program \( P \) consists of the smallest first-order language built from the constant, function, and predicate symbols occurring in \( P \).

We assume the reader to be familiar with standard notions in logic programming such as Herbrand interpretations, the \( T_P \) operator, iterations of the \( T_P \) operator, etc. (see [13]).
2.1. Stable and well-founded semantics

Suppose \( I \) is an Herbrand interpretation, i.e., \( I \) is a set of ground atoms. The Gelfond-Lifschitz transform (GL-transform) of a logic program \( P \) w.r.t. \( I \) is defined as

\[
G(P, I) = \{ D^+ | D \in \text{grd}(P) \text{ and none of the negated atoms in } D \text{ occurs in } I \}.
\]

Given a logic program \( P \), the associated operator \( F_P \) maps Herbrand interpretations to Herbrand interpretations:

\[
F_P(I) = \Upsilon_{G(P, I)} \uparrow \omega.
\]

In other words, \( F_P(I) \) is the set of ground atoms provable from the negation-free program \( G(P, I) \). It is well-known \([2, 17]\) that \( F_P \) is anti-monotonic w.r.t. \( \subseteq \), and hence \( F_P^2 \) has a least and a greatest fixpoint. An Herbrand interpretation \( I \) is called a stable model of \( P \) iff \( I = F_P(I) \). A ground atom \( A \) is true in the well-founded semantics of \( P \) iff \( A \in \text{lfp}(F_P^2) \), and false iff \( A \notin \text{gfp}(F_P^2) \).

2.2. S-semantics

Falaschi et al. \([7]\) define the concept of a non-ground fixpoint semantics for logic programming. An S-interpretation \( S_I \) is a collection of not necessarily ground atoms. An atom \( A \) is S-satisfied by \( S_I \) iff there is an atom \( A' \in S_I \) subsuming \( A \). A clause \( A \leftarrow A_1 \& \cdots \& A_n \) is S-satisfied by \( S_I \) iff for each tuple \((B_1, \ldots, B_n) \in S_I^n\) such that \( \theta \) is a most general simultaneous unifier (mgsu) of \((A_1, \ldots, A_n)\) and \((B_1, \ldots, B_n)\), the atom \( A\theta \) is S-satisfied by \( S_I \).\(^2\) An S-interpretation \( S_I \) can be thought of as a non-ground representation of the Herbrand interpretation

\[
\text{grd}(S_I) = \{ A | A \text{ is a ground instance of some atom } A' \in S_I \}.
\]

Given a logic program \( P \), Falaschi et al. define an operator \( W_P \) mapping S-interpretations to S-interpretations:

\[
W_P(S_I) = \{ A\theta | A \leftarrow A_1 \& \cdots \& A_n \text{ is in } P \text{ and there exist } B_1, \ldots, B_n \in S_I \text{ such that } \theta \text{ is an mgsu of } (A_1, \ldots, A_n) \text{ and } (B_1, \ldots, B_n) \}.
\]

An ordering \( \preceq^a \) can be defined on S-interpretations by \( S_I \preceq^a S_I' \) iff for all \( A \in S_I \) there is an \( A' \in S_I' \) such that \( A = A' \lambda \) for some substitution \( \lambda \). It turns out that under suitable conditions, \( W_P \) is continuous w.r.t. \( \preceq^a \) and \( S_I \) S-satisfies \( P \) iff \( W_P(S_I) \preceq^a S_I \).

2.3. Substitutions

A substitution \( \sigma \) is a mapping from variables to terms such that the domain of \( \sigma \), \( \text{dom}(\sigma) \), is finite, where \( \text{dom}(\sigma) = \{ v | \sigma(v) \neq v \} \). The range of \( \sigma \), denoted by \( \text{rg}(\sigma) \),

\(^2\) We assume wlog that S-interpretations and clauses as well as any two atoms in an S-interpretation share no variables.
is the set \( \{ \sigma(v) \mid v \in \text{dom}(\sigma) \} \). As usual, \( \sigma \) is written as \( \{ x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n) \} \) for \( \text{dom}(\sigma) = \{ x_1, \ldots, x_n \} \).

Substitutions are extended homomorphically from variables to terms, thus mapping terms to terms. The result of applying \( \sigma \) to a term \( t \) is written in postfix notation, i.e., as \( t \sigma \). The composition of substitutions \( \sigma \) and \( \sigma' \) is defined as their functional composition, written as \( \sigma \sigma' \) with the understanding that the application of \( \sigma \sigma' \) is equivalent to first applying \( \sigma \) and then \( \sigma' \).

Let \( V \) be a set of variables. \( \sigma \) and \( \sigma' \) are equal on \( V \), denoted as \( \sigma \equiv_V \sigma' \), iff \( \sigma(v) = \sigma'(v) \) for all variables \( v \in V \). \( \sigma \) is a \( V \)-instance of \( \sigma' \) iff there is a substitution \( \lambda \) such that \( \sigma \equiv_V \sigma' \lambda \); we also say that \( \sigma \) is less general than \( \sigma' \) on \( V \), written as \( \sigma \leq V \sigma' \). \( \sigma \) and \( \sigma' \) are \( V \)-variants iff they are \( V \)-instances of each other, denoted as \( \sigma \equiv_V \sigma' \).

The restriction of a substitution \( \sigma \) to \( V \), \( \sigma|_V \), is defined by \( \sigma|_V(x) = \sigma(x) \) for \( x \in V \) and \( \sigma|_V(x) = x \) for \( x \notin V \). The image of \( V \) under \( \sigma \), denoted by \( \text{img}_V(\sigma) \), is the multiset \( \{ x \sigma \mid x \in V \} \).

**Example 2.** Let \( \sigma_1 = \{ X \mapsto f(Y) \} \), \( \sigma_2 = \{ X \mapsto f(a), Z \mapsto a \} \) and \( \sigma_3 = \{ X \mapsto f(Z) \} \).

Furthermore, let \( V = \{ X \} \). \( \sigma_2 \) as well as \( \sigma_3 \) are \( V \)-instances of \( \sigma_1 \), since we have \( \sigma_2 \equiv_V \{ Y \mapsto a \} \) and \( \sigma_3 \equiv_V \{ Y \mapsto Z \} \). But \( \sigma_1 \) is also a \( V \)-instance of \( \sigma_3 \), since \( \sigma_1 \equiv_V \sigma_3 \) \( \{ Z \mapsto Y \} \), i.e., \( \sigma_1 \) and \( \sigma_3 \) are \( V \)-variants. Note that the three substitutions are unrelated to each other when including the variables \( Y \) and \( Z \) into \( V \).

The relations \( \leq_V \) and \( \equiv_V \) can be extended to sets \( \Sigma, \Sigma' \) of substitutions. \( \Sigma \preceq_V \Sigma' \) iff for every \( \sigma \in \Sigma \) there is a substitution \( \sigma' \in \Sigma' \) such that \( \sigma \leq_V \sigma' \). The reflexive and transitive relation \( \preceq_V \) induces an equivalence relation, \( \sim_V \), on sets of substitutions: \( \Sigma \sim_V \Sigma' \) iff \( \Sigma \preceq_V \Sigma' \) and \( \Sigma' \preceq_V \Sigma \).

The ordering on (sets of) substitutions can be defined similarly for (sets of) atoms. Let \( A \) and \( A' \) be atoms. Then \( A \preceq A' \) iff there is a substitution \( \lambda \) such that \( A = A' \lambda \); in this case, \( A \) is said to be an instance of \( A' \). If \( A \preceq A' \) and \( A' \preceq A \) then \( A \) and \( A' \) are variants of each other, written as \( A \equiv A' \). For sets \( SI \) and \( SI' \) of atoms, \( SI \preceq SI' \) iff for every \( A \in SI \) there is an \( A' \in SI' \) such that \( A \preceq A' \); if \( SI \preceq SI' \) and \( SI' \preceq SI \) then \( SI \) and \( SI' \) are variants, written as \( SI \sim SI' \).

Let \( C = A \leftarrow A_1 & \cdots & A_n \) and \( C' = A' \leftarrow A'_1 & \cdots & A'_m \) be clauses. \( C \) is subsumed by \( C' \), denoted as \( C \preceq C' \), iff \( A = A' \lambda \) and \( \{ A'_1 \lambda, \ldots, A'_m \lambda \} \subseteq \{ A_1, \ldots, A_n \} \) for some substitution \( \lambda \). The relation \( \equiv \) as well as the extensions of \( \leq \) and \( \equiv \) to sets of clauses (= programs) is defined in the same way as for atoms.

The set of all variables occurring in an object \( O \) – i.e., in a term, clause, (multi)set of terms or in a program – is denoted by \( \text{var}(O) \).

For a finite set \( V \) of variables, let \( \text{GS}_V \) denote the set of all ground substitutions with domain \( V \). For a substitution \( \sigma \) and a set, \( \Sigma \), of substitutions we define
\[
\text{GS}_V(\sigma) = \{ \theta \in \text{GS}_V \mid \theta \text{ is a } V\text{-instance of } \sigma \} ,
\]
\[
\text{GS}_V(\Sigma) = \bigcup_{\sigma \in \Sigma} \text{GS}_V(\sigma) .
\]
Simply put, $\text{GS}_V(\Sigma)$ contains exactly those ground substitutions of $\text{GS}_V$, which are $V$-instances of some substitution in $\Sigma$. $\Sigma$ can be regarded as a non-ground representation of $\text{GS}_V(\Sigma)$.

**Example 3.** Suppose our language contains just one constant symbol $a$ and one unary function symbol $f$. Let $V = \{X, Y\}$. For the set of all ground substitutions with domain $V$ we obtain

$$\text{GS}_V = \{\{X \mapsto f^i(a), Y \mapsto f^j(a)\} \mid i, j \geq 0\}.$$  

Let $\Sigma = \{\sigma_1, \sigma_2\}$ where $\sigma_1 = \{X \mapsto a, Z \mapsto a\}$ and $\sigma_2 = \{X \mapsto f(Y)\}$. Then $\text{GS}_V(\Sigma)$ is the union of

$$\text{GS}_V(\sigma_1) = \{\{X \mapsto a, Y \mapsto f^j(a)\} \mid j \geq 0\},$$  

$$\text{GS}_V(\sigma_2) = \{\{X \mapsto f^{i+1}(a), Y \mapsto f^j(a)\} \mid j \geq 0\}.$$  

Note that the component $Z \mapsto a$ in $\sigma_1$ is irrelevant since $Z \notin V$.

$V$ is intended to comprise all variables contained in the program or clause under consideration, while variables occurring in $S$-interpretations are excluded from $V$. If $V$ is clear from context we omit the index or prefix $V$, writing simply $\sigma = \sigma'$, $\sigma \leq \sigma'$, $\text{GS}(\Sigma)$, instance, variant, etc.

3. A non-ground Gelfond–Lifschitz transform

The aim of this section is to define a generalized GL-transform, genG. It takes as input a logic program $P$ and an $S$-interpretation $SI$, and returns a negation-free program $P'$ such that $P'$ is a (non-ground) representation of $G(P, I)$, where $I = \text{grd}(SI)$. More formally, $P'$ has to meet two requirements:

- **Completeness:** $G(P, I) \subseteq \text{grd}(P')$. For every (ground) clause $D$ in $G(P, I)$ there is some clause in $P'$ having $D$ as an instance.
- **Correctness:** $\text{grd}(P') \subseteq G(P, I)$. Each ground instance of a clause in $P'$ is in $G(P, I)$.

Our first step is to reformulate the definition of the GL-transform in terms of ground substitutions instead of ground clauses:

$$G(P, I) = \{ (C\theta)^+ \mid C \in P, \theta \in \text{GS}_{\text{var}(C)}, \text{and none of the negated atoms in } C\theta \text{ occurs in } I \}. \quad (\text{GL}')$$

Obviously, this definition is equivalent to Eq. (GL) in Section 2.1. Next we replace the ground interpretation $I$ by an $S$-interpretation $SI$ obtaining

$$G'(P, SI) = \{ (C\theta)^+ \mid C \in P, \theta \in \text{GS}_{\text{var}(C)}, \text{and none of the negated atoms in } C\theta \text{ is an instance of an atom in } SI \}.$$
It is not hard to see that \( G'(P, SI) = G(P, \text{grd}(SI)) \). Before proceeding further, we partition \( \text{GS}_{\text{var}(C)} \) into two classes \( \mathcal{G} \) and \( \mathcal{B} \) containing the "good" and the "bad" substitutions, respectively:

\[
\mathcal{G}_{C,SI} = \{ \theta \in \text{GS}_{\text{var}(C)} \mid \text{none of the negated atoms in } C \theta \text{ is an instance of an atom in } SI \} ,
\]

\[
\mathcal{B}_{C,SI} = \{ \theta \in \text{GS}_{\text{var}(C)} \mid \text{some negated atom in } C \theta \text{ is an instance of an atom in } SI \} .
\]

Using \( \mathcal{G}_{C,SI} \), transformation \( G' \) can be more concisely written as

\[
G'(P, SI) = \{(C \theta)^+ \mid C \in P, \theta \in \mathcal{G}_{C,SI}\} .
\]

The final step towards a generalized GL-transform is to represent \( \mathcal{G}_{C,SI} \) by a set of non-ground substitutions, by a so-called cover.

**Definition 1 (Cover).** Let \( V \) be a finite set of variables and \( \mathcal{G} \) be a set of ground substitutions with domain \( V \). A set \( \Sigma \) of substitutions is a \( V \)-cover of \( \mathcal{G} \) iff \( \text{GS}_V(\Sigma) = \mathcal{G} \). \( \Sigma \) is a maximal \( V \)-cover of \( \mathcal{G} \) iff for all \( V \)-covers \( \Sigma' \) of \( \mathcal{G} \), \( \Sigma \preceq_V \Sigma' \) implies \( \Sigma \sim_V \Sigma' \). Furthermore, \( \Sigma \) is an optimal \( V \)-cover iff it is maximal and for any two substitutions \( \sigma, \tau \in \Sigma \), \( \sigma \leq_V \tau \) implies \( \sigma = \tau \).

In Section 6.1 we show that the optimal cover always exists and is unique up to variable renaming. An optimal cover is minimal among all maximal covers when cardinality is considered: it contains neither variants nor subsumed substitutions.

**Definition 2 (Generalized/Non-Ground GL-transform).** Let \( P \) be a logic program and \( SI \) an \( S \)-interpretation. A generalized GL-transform of \( P \) w.r.t. \( SI \) is a negation-free program

\[
\text{genG}(P, SI) = \{(C \theta)^+ \mid C \in P, \theta \in \Sigma_{C,SI}\} ,
\]

where \( \Sigma_{C,SI} \) is a \( \text{var}(C) \)-cover of \( \mathcal{G}_{C,SI} \) for all \( C \in P \). If \( \Sigma_{C,SI} \) is an optimal \( V \)-cover for all \( C \) then \( \text{genG}(P, SI) \) is the non-ground GL-transform of \( P \) w.r.t. \( SI \), denoted by \( \text{ngG}(P, SI) \).\(^3\)

A non-ground GL-transform is just one particular kind of generalized GL-transforms, obtained by choosing optimal covers only. As proved in Section 6.1, optimal covers are unique up to variants, hence it is justified to speak of the non-ground GL-transform of a program w.r.t. an \( S \)-interpretation. Furthermore, since optimal covers are maximal and contain no redundancies, the non-ground GL-transform yields the most general

\(^3\)To be precise, \( \text{genG}(P, SI) \) ought to have an additional parameter, viz. a function \( f \) mapping every clause \( C \in P \) to the particular \( \Sigma_{C,SI} \) chosen as cover of \( \mathcal{G}_{C,SI} \). Usually the choice of \( f \) is of no relevance, therefore we omit this additional parameter for the sake of better readability. In the case of \( \text{ngG} \) the choice is completely irrelevant since optimal covers are unique up to renaming.
representation of the ground GL-transform. Note that the ground GL-transform itself is just a special case of the generalized GL-transform: if we choose $\mathcal{G}_{C,SI}$ as covers in the above definition, then $\text{genG}(P, SI) = G'(P, SI) = G(P, \text{grd}(SI))$.

**Example 4.** Suppose our language contains one constant symbol $a$ and one unary function symbol $f$. Let $P$ be the program consisting of the single clause

$$C = p(x,y) \leftarrow q(x) \& \neg(r(a,x)) \& \neg(r(y,y))$$

and let $SI$ be the $S$-interpretation $\{r(U,a), r(f(f(V)), W)\}$. $\mathcal{G}_{C,SI}$ is the set of all ground substitutions $\theta$ with domain $V = \text{var}(C) = \{x, y\}$ such that neither $r(a, x)\theta$ nor $r(y, y)\theta$ is an instance of any atom in $SI$. We obtain

$$\mathcal{G}_{C,SI} = \{\{x \mapsto f(i(a)), y \mapsto f(a)\} \mid i \geq 1\}$$

The optimal cover of $\mathcal{G}_{C,SI}$ is the singleton set containing the substitution $\{x \mapsto f(Z), y \mapsto f(a)\}$. Thus the non-ground GL-transform of $P$ w.r.t. $SI$ is the program

$$\text{ngG}(P, SI) = \{p(f(Z), f(a)) \leftarrow q(f(Z))\}.$$ 

Note that the ground GL-transform, $G(P, \text{grd}(SI))$, is a set containing an infinite number of ground clauses, which are exactly the ground instances of the clause in $\text{ngG}(P, SI)$.

The following theorem shows that $\text{genG}$ captures our intentions: every ground instance of a clause in $\text{genG}(P, SI)$ occurs in the (ground) Gelfond–Lifschitz transform of $P$ w.r.t. $\text{grd}(SI)$, and conversely, for every clause $C$ in $G(P, \text{grd}(SI))$ there is a clause in $\text{genG}(P, SI)$ subsuming $C$. In other words, $\text{genG}$ is both complete and correct.

**Theorem 1.** Let $P$ be a logic program and $SI$ an $S$-interpretation. Then

$$\text{grd}(\text{genG}(P, SI)) = G(P, \text{grd}(SI)).$$

**Proof.** It is sufficient to show $\text{grd}(\text{genG}({\{C\}, SI})) = G({\{C\}, \text{grd}(SI)})$ for each clause $C \in P$. Let $\Sigma$ be the cover of $\mathcal{G}_{C,SI}$ used in the generalized GL-transform.

$$D \in \text{grd}(\text{genG}({\{C\}, SI})) \Rightarrow D \in G({\{C\}, \text{grd}(SI)})$$

By definition of $\text{genG}$, there is a substitution $\sigma \in \Sigma$ such that $D$ is a ground instance of $(C\sigma)^+$, i.e., $D = (C\sigma)^+\theta = (C\sigma_\theta)^+$ for some substitution $\theta \in \text{GS}_{\text{var}(C)}$. Since $\Sigma$ is a cover of $\mathcal{G}_{C,SI}$, none of the negated atoms in $C\sigma\theta$ is an instance of an atom in $SI$, i.e., none of them occurs in $\text{grd}(SI)$. Hence, by the definition of the GL-transform, $(C\sigma_\theta)^+ = D$ is in $G({\{C\}, \text{grd}(SI)})$.

$$D \in G({\{C\}, \text{grd}(SI)}) \Rightarrow D \in \text{grd}(\text{genG}({\{C\}, SI}))$$

Since $D$ is in the GL-transform of $C$, there is a substitution $\theta \in \text{GS}_{\text{var}(C)}$ such that $D = (C\theta)^+$, and none of the negated atoms in $C\theta$ is an instance of an atom in $SI$. Hence $\theta$ belongs to the good substitutions, $\mathcal{G}_{C,SI}$, which are covered by $\Sigma$. Therefore $\Sigma$ contains a substitution $\sigma$ having $\theta$ as an instance: $\theta = \sigma\lambda$ for some substitution $\lambda$. We obtain $D = (C\theta)^+ = (C\sigma\lambda)^+ = (C\sigma)^+\lambda$, i.e., $D$ is a ground instance of $(C\sigma)^+$, which is a clause in $\text{genG}({\{C\}, SI})$. □
The non-ground GL-transform is anti-monotonic w.r.t. to S-interpretations. The proposition below states a slightly stronger version with \( \text{grd}(SI) \) replacing \( SI \). The weaker version is given as a corollary.

**Proposition 2** (Anti-monotonicity of \( \text{ngG} \)). For a program \( P \) and S-interpretations \( SI \) and \( SI' \), \( \text{grd}(SI) \subseteq \text{grd}(SI') \) implies \( \text{ngG}(P, SI') \preceq \text{ngG}(P, SI) \).

**Proof.** It is sufficient to show the implication for a single clause, i.e., \( P = \{C\} \). Let \( V = \text{var}(C) \). We start by observing that \( \text{grd}(SI) \subseteq \text{grd}(SI') \) implies \( \mathcal{G}_{C,SI'} \subseteq \mathcal{G}_{C,SI} \), which in turn implies \( \Sigma' \preceq_V \Sigma \), where \( \Sigma \) and \( \Sigma' \) are optimal covers of \( \mathcal{G}_{C,SI} \) and \( \mathcal{G}_{C,SI'} \), respectively. For each clause \( D' \in \text{ngG}(\{C\}, SI') \) there is a substitution \( a' \in \Sigma' \) such that \( D' = C a' \). Because of \( \Sigma' \preceq_V \Sigma \), \( \Sigma \) contains a substitution \( \sigma \) with \( \sigma' \preceq_V \sigma \). Thus we have \( D' = C a' \preceq_C C \sigma = D \) with \( D \in \text{ngG}(P, SI) \). We conclude that \( \text{ngG}(\{C\}, SI') \preceq \text{ngG}(\{C\}, SI) \). \( \square \)

The first of the following corollaries states the non-monotonicity of \( \text{ngG} \) in its weaker form. The second one shows that the result of \( \text{ngG} \) is independent of the particular representation, \( SI \) or \( SI' \), chosen for the Herbrand interpretation \( \text{grd}(SI) = \text{grd}(SI') \). Both corollaries will be needed in subsequent proofs.

**Corollary 3.** (a) \( SI \preceq SI' \) implies \( \text{ngG}(P, SI') \preceq \text{ngG}(P, SI) \).

(b) \( \text{grd}(SI) = \text{grd}(SI') \) implies \( \text{ngG}(P, SI) \sim \text{ngG}(P, SI') \).

4. A non-ground stable and well-founded semantics

Using the non-ground GL-transform of the last section and the work of Falaschi et al. [7] on S-semantics, we can now define a non-ground stable semantics and a non-ground well-founded semantics. As we will show below in Theorems 4 and 7, these non-ground semantics are proper generalizations of their ground counterparts.

**Definition 3** (\( \text{ngF} \)). For a logic program \( P \), the operator \( \text{ngF}_P \) maps S-interpretations to S-interpretations and is defined as

\[
\text{ngF}_P(SI) = \text{lfp}(W_{\text{ngG}(P, SI)})
\]

In other words, the function \( \text{ngF}_P \), when applied to an S-interpretation \( SI \), returns as output the least S-model of the negation-free program \( \text{ngG}(P, SI) \). The analogues of the ground and the non-ground case are summed up in Table 1.

**Definition 4** (Non-ground stable model). An S-interpretation \( SI \) is a non-ground stable model of \( P \) iff \( SI \sim \text{ngF}_P(SI) \).
The analogues in the ground and the non-ground case

<table>
<thead>
<tr>
<th></th>
<th>Ground case</th>
<th>Non-ground case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>( \text{grd}(P) )</td>
<td>( P )</td>
</tr>
<tr>
<td>Fixpoint operator</td>
<td>( T_P )</td>
<td>( W_P )</td>
</tr>
<tr>
<td>GL-transformation</td>
<td>( G(P, I) )</td>
<td>( \text{ngG}(P, SI) )</td>
</tr>
<tr>
<td>GL-operator</td>
<td>( F_P )</td>
<td>( \text{ngF}_P )</td>
</tr>
</tbody>
</table>

\( SI \) is a non-ground stable model of \( P \) iff every atom in \( SI \) is subsumed by an atom in \( \text{ngF}_P(SI) \), and conversely, every atom in \( \text{ngF}_P(SI) \) is subsumed by an atom in \( SI \). The following theorem states that the notion of a non-ground stable model captures the standard notion of a ground stable model.

**Theorem 4.** Let \( P \) be a logic program. Then:

(a) \( \text{grd}(\text{ngF}_P(SI)) = F_P(\text{grd}(SI)) \) for all S-interpretations \( SI \).
(b) If \( SI \) is a non-ground stable model of \( P \), then \( \text{grd}(SI) \) is a stable model of \( P \).
(c) If \( I \) is a stable model of \( P \), then there is a non-ground stable model \( SI \) of \( P \) such that \( \text{grd}(SI) = I \).

**Proof.** (a) By definition, \( \text{ngF}_P(SI) = \text{lfp}(W_{\text{ngG}(P, SI)}) = \bigcup_{n \geq 0} SI_n \) where \( SI_0 = \emptyset \) and \( SI_{n+1} = W_{\text{ngG}(P, SI)}(SI_n) \). Hence we obtain

\[
\text{grd}(\text{ngF}_P(SI)) = \bigcup_{n \geq 0} \text{grd}(SI_n),
\]

where

\[
\text{grd}(SI_0) = \emptyset,
\]
\[
\text{grd}(SI_{n+1}) = \text{grd}(W_{\text{ngG}(P, SI)}(SI_n)).
\]

For an arbitrary negation-free program \( P' \) and an S-interpretation \( SI' \) we have

\[
\text{grd}(W_{P'}(SI')) = T_{P'}(\text{grd}(SI')) = T_{\text{grd}(P')}(\text{grd}(SI')).
\]

Choosing \( P' = \text{ngG}(P, SI) \) and \( SI' = SI_n \) we obtain

\[
\text{grd}(W_{\text{ngG}(P, SI)}(SI_n)) = T_{\text{ngG}(P, SI)}(\text{grd}(SI_n)).
\]

By Theorem 1, \( \text{grd}(\text{ngG}(P, SI)) = G(P, \text{grd}(SI)) \), i.e.,

\[
T_{\text{ngG}(P, SI)}(\text{grd}(SI_n)) = T_{G(P, \text{grd}(SI))}(\text{grd}(SI_n)).
\]

Setting \( I_n = \text{grd}(SI_n) \), the recursion for \( \text{grd}(SI_n) \) takes the form

\[
I_0 = \emptyset, \quad I_{n+1} = T_{G(P, \text{grd}(SI))}(I_n).
\]

Summarizing we have

\[
\text{grd}(\text{ngF}_P(SI)) = \bigcup_{n \geq 0} I_n = \text{lfp}(T_{G(P, \text{grd}(SI))}) = F_P(\text{grd}(SI)).
\]
(b) By definition, $SI \sim^a ngF_p(SI)$, and therefore $grd(SI) = grd(ngF_p(SI))$. By the first part of this theorem, $grd(ngF_p(SI)) = F_p(grd(SI))$. We conclude that $grd(SI) = F_p(grd(SI))$, i.e., $grd(SI)$ is a stable model of $P$.

(c) We show that $SI = ngF_p(I)$ is a non-ground stable model of $P$ such that $grd(SI) = I$. We have

$$
grd(SI) = grd(ngF_p(I))$$

$$= F_p(grd(I)) \quad \text{(by Theorem 4(a))}$$

$$= F_p(I) \quad \text{(I is ground)}$$

$$= I \quad \text{(I is a stable model)} .$$

Furthermore, $SI$ is a non-ground stable model of $P$ since

$$ngF_p(SI) = lfp(ngG(P, SI))$$

$$\sim^a lfp(ngG(P, I))$$

$$= ngF_p(I)$$

$$= SI .$$

The step from the first to the second line is an application of Corollary 3(b): $ngG(P, SI) \sim^c ngG(P, I)$ since $grd(SI) = I = grd(I)$. □

**Definition 5** (Non-ground well-founded semantics). Let $P$ be a logic program and $A$ be some atom (not necessarily ground). $A$ is true in the non-ground well-founded semantics of $P$ iff it is an instance of some atom in $lfp(ngF_p)$; $A$ is false iff it is not an instance of any atom in $gfp(ngF_p)$. Note that $A$ being true (false) in the well-founded semantics implies that all ground instances of $A$ are true (false).

The non-ground well-founded semantics is well defined only if the fixpoints $lfp(ngF_p)$ and $gfp(ngF_p)$ exist. Therefore we show that $ngF$ is anti-monotonic w.r.t. the ordering $\preceq^a$ on S-interpretations.

**Proposition 5** (Anti-monotonicity of $ngF$). For a program $P$ and S-interpretations $SI$ and $SI'$, $SI \preceq^a SI'$ implies $ngF_p(SI') \preceq^a ngF_p(SI)$.

**Proof.** Because of the anti-monotonicity of $ngG$ (Corollary 3(a)) we have $Q' = ngG(P, SI') \preceq^c ngG(P, SI) = Q$. By the monotonicity of $W$ w.r.t. negation-free programs (see Lemma 6 below), $W_{Q'}(SI) \preceq^a W_Q(SI)$ for all S-interpretations $SI$, and hence $lfp(W_{Q'}) \preceq^a lfp(W_Q)$. We conclude that $ngF_p(SI') \preceq^a ngF_p(SI)$. □

The following lemma shows that the $W$-operator is monotonic w.r.t. negation-free programs, i.e., if every clause in a program $P$ is subsumed by a clause in a program $P'$, then for an arbitrary S-interpretation $SI$ every atom in $W_P(SI)$ is an instance of some atom in $W_{P'}(SI)$. This result is needed in the proof of Proposition 5 above.
Lemma 6. Let P and P' be definite logic programs. Then $P \preceq^c P'$ implies $W_P(SI) \preceq^a W_{P'}(SI)$ for all S-interpretations SI.

Proof. Let $D$ be an atom in $W_P(SI)$. Then there has to be a clause $C = A \leftarrow A_1 \& \cdots \& A_n$ in P and atoms $B_1, \ldots, B_n$ in SI such that the tuples $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_n)$ are simultaneously unifiable via a most general unifier $\theta$ and $D = A\theta$. Because of $P \preceq^c P'$, $C$ is subsumed by some clause $C' \in P'$, i.e., $P'$ contains a clause $C' = A' \leftarrow A'_1 \& \cdots \& A'_m$ such that $A = A'\lambda$ and $\{A'_1, \ldots, A'_m\} \subseteq \{A_1, \ldots, A_n\}$ for some substitution $\lambda$. It is not hard to see that $\lambda\theta$ is a simultaneous unifier of the $m$-tuple $(A'_1, \ldots, A'_m)$ and an $m$-tuple consisting of (some of the) $B_i$'s. Therefore there exists a most general unifier $\theta'$ of these $m$-tuples, and $\lambda\theta = \theta'\lambda'$ for some substitution $\lambda'$. Thus $D = A\theta = A'\lambda\theta = A'\theta'\lambda' = D'\lambda'$, and $D'$ is an atom in $W_{P'}(SI)$. This completes the proof. 

The following theorem shows that our non-ground well-founded semantics accurately generalizes the ground well-founded semantics.

Theorem 7. Let P be a logic program. Then:

(a) $\text{grd}(\text{lfp}(\text{ngF}_2^2)) = \{A | A \text{ is true in the well-founded sem. of } \text{grd}(P)\}$.
(b) $\text{grd}(\text{gfp}(\text{ngF}_2^2)) = \{A | A \text{ is false in the well-founded sem. of } \text{grd}(P)\}$.

Proof. (a) From Theorem 4(a) we obtain by transfinite induction that for all ordinals $\gamma$, $F_{\text{grd}}^2 \uparrow \gamma = \text{grd}(\text{ngF}_2^2 \uparrow \gamma)$. By results in [2, 17], the set

$$\{A | A \text{ is true in the well-founded semantics of } \text{grd}(P)\}$$

is equal to $F_{\text{grd}}^2 \uparrow \lambda$ for some ordinal $\lambda$. Now the assertion of the theorem follows immediately.

(b) Analogous to the proof of the first part. 

Remark. The operator $\text{ngF}$ and thus both semantics defined in this section are based on the non-ground GL-transform, i.e., on optimal covers. In principle, one could also use other kinds of generalized GL-transforms, leading to different non-ground representations of programs and interpretations. For the sake of simplicity we have restricted our discussion here to the uniquely defined operator $\text{ngG}$. However, we would like to emphasize that the above results can be extended to other kinds of generalized GL-transforms.

5. Bad sets and anticovers

Generalized GL-transforms are based on covers for sets of “good” substitutions. From a computational point of view this definition is not quite satisfactory: starting from a non-ground S-interpretation SI, one has to compute – at least in principle –
the ground set $G_{C,S_I}$ in order to find a non-ground cover for it. Both of these sets are infinite in general, even if $S_I$ is finite.

In this section we show how the covers needed in the generalized GL-transform can be characterized via the so-called bad sets and anticovers. This approach has the advantage of being one step nearer towards an effective algorithm. A bad set is a certain kind of cover representing the "bad" substitutions of a clause. If $S_I$ is finite, it is finite, too, and can be immediately computed from the clause and $S_I$. The covers for the generalized GL-transform are anticovers of these bad sets, i.e., they represent exactly those ground substitutions not covered by the bad sets. Moreover, for an arbitrary substitution there is an effective method for testing whether it belongs to some anticover or not.

**Definition 6 (Weak unifiability).** Let $V$ be a set of variables. Two terms $s$ and $t$ are weakly unifiable w.r.t. $V$ iff there are substitutions $\mu$ and $\eta$ satisfying

1. $\eta$ is a renaming substitution such that $\text{var}(\eta) \cap \text{var}(s) = \emptyset$ and $\text{var}(\eta) \cap V = \emptyset$,

and

2. $s\mu = t\eta\mu$, i.e., $\mu$ is a unifier of $s$ and the renamed version of $t$.

$\mu$ is called a weak unifier of $s$ and $t$ (w.r.t. $V$). If $\mu$ is a most general unifier of $s$ and $\eta$, then it is called a most general weak unifier of $s$ and $t$.

Two substitutions $\sigma$ and $\tau$ are weakly unifiable w.r.t. $V$, where $V = \{v_1, \ldots, v_n\}$ is finite, iff for some arbitrary $n$-ary function symbol $f$ the terms $s = f(v_1, \ldots, v_n)\sigma$ and $t = f(v_1, \ldots, v_n)\tau$ are weakly unifiable w.r.t. $V$. $\mu$ is a (most general) weak unifier of $\sigma$ and $\tau$ w.r.t. $V$ iff it is one of $s$ and $t$.

Note that unifiability implies weak unifiability, but not vice versa. Furthermore, it is easy to see that weak unifiability is decidable and that a weak most general unifier can be effectively computed, provided it exists. Weak unification is investigated in [5].

**Lemma 8.** The most general weak unifier of two terms or substitutions w.r.t. $V$ is unique up to $V$-variants.

**Proof.** By the definition above, the assertion concerning substitutions reduces to the one concerning terms. Let $s$ and $t$ be two terms, and let $\mu, \mu'$ be most general weak unifiers of $s$ and $t$ w.r.t. $V$. Let $\eta, \eta'$ be renaming substitutions such that $s\mu = t\eta\mu$ and $s\mu' = t\eta'\mu'$. We show that $\mu' \leq V \mu$. For reasons of symmetry we then also have $\mu \leq V \mu'$, and thus $\mu \equiv V \mu'$.

We start by observing that $t\eta$ and $t\eta'$ are equal up to renaming, i.e., there is a renaming substitution $\rho$ such that $t\eta' = t\rho \eta$ and $\text{dom}(\rho) \subseteq \text{var}(\eta)$. Because of $\text{var}(\eta) \cap \text{var}(s) = \emptyset$ and $\text{var}(\eta) \cap V = \emptyset$ we also have $\text{dom}(\rho) \cap (V \cup \text{var}(s)) = \emptyset$. Therefore $s\rho = s$ and $\rho \mu' = V \mu'$. Putting all together we obtain $s\rho \mu' = s\mu' = t\eta' \mu' = t\rho \mu'$, i.e., $\rho \mu'$ is a unifier of $s$ and $\eta$. By definition, $\mu$ is a most general unifier of $s$ and $\eta$ and thus is more general than $\rho \mu' = V \mu'$. We conclude that $\mu' \leq V \mu$. $\square$
Because of this lemma it is justified to speak of the most general weak unifier of \( s \) and \( t \) w.r.t. \( V \), denoted by \( \text{mgwu}_V(s,t) \), in spite of the various choices for the renaming substitution \( \eta \).

**Example 5.** The terms \( s = f(X,a) \) and \( t = f(b,X) \) are not unifiable, but weakly unifiable. All weak unifiers, which are also most general in this case, have the form \( \{X \mapsto b, v \mapsto a\} \) where \( v \) is some variable different from \( X \). The renaming substitution \( \eta \) is given by \( \{X \mapsto v\} \). Clearly, all weak unifiers are equivalent to each other on any set \( V \) not containing \( v \).

**Definition 7 (Bad set).** Let \( C = A \leftarrow A_1 \land \cdots \land A_n \land \neg(B_1) \land \cdots \land \neg(B_m) \) be a clause and \( SI \) be an \( S \)-interpretation. The bad set of \( C \) w.r.t. \( SI \) is the set of all most general weak unifiers computable from negated atoms in \( C \) and atoms in \( SI \), i.e.,

\[
\text{bad}(C, SI) = \{\text{mgwu}_{\text{var}(C)}(B_i,D) \mid 1 \leq i \leq m, D \in SI\}.
\]

**Example 4 (continued).** The negated atoms in \( C \) are \( \neg(a, X) \) and \( \neg(Y, Y) \), the atoms in \( SI \) are \( r(U, a) \) and \( r(f(f(V)), W) \). We obtain three weak unifiers, leading to the bad set

\[
\text{bad}(C, SI) = \{\{U \mapsto a, X \mapsto a\},
\{U \mapsto a, Y \mapsto a\},
\{W \mapsto f(f(V)), Y \mapsto f(f(V))\}\}.
\]

Note that we only need the components concerning variables in \( V \). We may drop the components \( U \mapsto a \) and \( W \mapsto f(f(V)) \); the resulting substitutions still form a cover of the set of bad substitutions.

**Lemma 9.** Let \( C \) be a clause and \( SI \) be an \( S \)-interpretation. Then \( \text{bad}(C, SI) \) is a \( \text{var}(C) \)-cover of \( B_{C,SI} \). Furthermore, if \( SI \) is finite then \( \text{bad}(C, SI) \) is finite, too.

**Proof.** Since the number of negated atoms in \( C \) is finite, the finiteness of \( SI \) trivially implies the finiteness of \( \text{bad}(C, SI) \). It remains to show that \( \text{bad}(C, SI) \) is a \( \text{var}(C) \)-cover of \( B_{C,SI} \), i.e., that \( \text{GS}_{\text{var}(C)}(\text{bad}(C, SI)) = B_{C,SI} \).

\( \theta \in \text{GS}_{\text{var}(C)}(\text{bad}(C, SI)) \Rightarrow \theta \in B_{C,SI} \): \( \theta \) is a \( \text{var}(C) \)-instance of a substitution in \( \text{bad}(C, SI) \), which is the most general weak unifier of some negated atom \( B \) in \( C \) and some atom \( D \in SI \). Hence some negated atom in \( C \theta \), viz. \( B \theta \), is an instance of \( D \), which by definition of \( B_{C,SI} \) implies \( \theta \in B_{C,SI} \).

\( \theta \in B_{C,SI} \Rightarrow \theta \in \text{GS}_{\text{var}(C)}(\text{bad}(C, SI)) \): \( \theta \) is in \( B_{C,SI} \) iff \( \theta \in \text{GS}_{\text{var}(C)} \) and some negated atom in \( C \theta \), say \( B' \), is an instance of an atom \( D \in SI \). Let \( B \) be a negated atom in \( C \) such that \( B \theta = B' \). \( B \theta \) being an instance of \( D \) implies that \( B \) and \( D \) are unifiable via a most general weak unifier \( \sigma \), which by the definition of bad is in \( \text{bad}(C, SI) \). Since \( \sigma \) is most general, the ground substitution \( \theta \) is a \( \text{var}(C) \)-instance of \( \sigma \), hence \( \theta \in \text{GS}_{\text{var}(C)}(\text{bad}(C, SI)) \). \( \square \)
Note that bad(C,SI) may be redundant in many ways, and thus need not be an optimal cover. For instance, an atom in SI may be an instance of another atom in SI, which leads to two different unifiers with one subsuming the other.

**Definition 8 (Anticover).** Let V be a finite set of variables and Σ, Σ' be sets of substitutions. Σ' is a V-anticover of Σ iff GS_V(Σ) and GS_V(Σ') are disjoint and together cover GS_V, i.e., iff

1. GS_V(Σ') ∪ GS_V(Σ) = GS_V,
2. GS_V(Σ') ∩ GS_V(Σ) = ∅.

**Lemma 10.** Let C be a clause and SI be an S-interpretation. A set of substitutions is a var(C)-cover of G_C,SI iff it is a var(C)-anticover of bad(C,SI).

**Proof.** The lemma follows immediately from Lemma 9 and the facts that G_C,SI ∪ B_C,SI = GS_var(C) and G_C,SI ∩ B_C,SI = ∅. □

**Example 4 (continued).** Let Σ be the bad set of C w.r.t. SI with all redundant components removed, i.e., Σ = {{X ↦ a}, {Y ↦ a}, {Y ↦ f(f(g))}}. The ground substitutions not covered by Σ are those which are not an instance of any substitution in Σ. Since f(a) is the only term which is an instance neither of a nor of f(f(V)) they are of the form {X ↦ f(i(a)), Y ↦ f(a)} for i ≥ 1. Obviously, the set consisting of the single substitution {X ↦ f(Z), Y ↦ f(a)} is a cover of these substitutions, and thus an anticover of Σ.

The definition of bad sets depends on weak unification. To see why we cannot use the simpler concept of ordinary unification instead, consider the following example.

**Example 6.** Let our language consist of just two constant symbols, a and b, and let C be the clause p(X) ← not(q(a)). We compute the non-ground GL-transform of {C} w.r.t. the S-interpretation SI = {q(X)}.

Suppose bad sets were defined via ordinary unification. Unifying q(a) and q(X) we would obtain the set {{X ↦ a}} for bad(C,SI). For V = var(C) = {X}, a V-anticover is given by {{X ↦ b}} leading to the 'non-ground GL-transform' {p(b) ←}. Unfortunately, the ground GL-transform of {C} w.r.t. grd(SI) = {q(a), q(b)} is just the empty set, since the body of the clause is always false in the interpretation.

The reason for the divergence between ground and 'non-ground GL-transform' lies in the wrong treatment of X. Each atom in SI as well as each clause has to be regarded as being universally quantified. Therefore, X occurring in SI is different from X occurring in C. Weak unification takes this difference into account: variables in SI are renamed prior to unification. Computing bad(C,SI) according to Definition 7 we obtain {{X' ↦ a}}; its unique V-anticover is the empty set. Now the non-ground GL-transform of C w.r.t. SI is correctly obtained as the empty set.
Note also the importance of considering only $V$-anticovers, where $V = \text{var}(C)$. If in our example $V$ also contained $X'$, a $V$-anticover of the bad set would be given by $\{\{X' \mapsto b\}\}$. This anticover, however, leads again to a wrong GL-transform, namely $\{p(X) \leftarrow \}$.

**Definition 9 (incompatible).** Let each of $O_1$ and $O_2$ be either a substitution or a set of substitutions. $O_1$ and $O_2$ are incompatible on $V$ iff $\text{GS}_V(O_1) \cap \text{GS}_V(O_2) = \emptyset$; otherwise they are compatible on $V$.

By definition, every $V$-anticover of $\Sigma$ is incompatible with $\Sigma$ on $V$. Similarly, if $\sigma$ belongs to any anticover of $\Sigma$ then it is incompatible with $\Sigma$. Provided that the sets involved are finite, (in)compatibility can be effectively tested.

**Lemma 11.** Let $V$ be a finite set of variables. Two substitutions $\sigma$ and $\tau$ are compatible on $V$ iff they are weakly unifiable on $V$.

**Proof.** If $\sigma$ and $\tau$ are compatible on $V$ then there is a substitution $\theta$ with $\theta \in (\text{GS}_V(\sigma) \cap \text{GS}_V(\tau))$. $\theta$ is a $V$-instance of both $\sigma$ and $\tau$, or equivalently, of $\sigma$ and $v\eta$ where $\eta$ is a renaming substitution such that $\text{img}_V(\sigma)$ and $\text{img}_V(\tau\eta)$ share no variables. Therefore $\sigma\lambda = \theta$ and $v\eta\lambda' = \theta$ for some substitutions $\lambda, \lambda'$, i.e., $v\sigma\lambda = v\tau\eta\lambda'$ for all $v \in V$. Since $\text{var}(\text{img}_V(\sigma)) \cap \text{var}(\text{img}_V(\tau\eta)) = \emptyset$, the domains of $\lambda$ and $\lambda'$ are disjoint and $\mu = \lambda \cup \lambda'$ is a well defined substitution. Now we obtain $v\tau\eta\mu = v\sigma\mu$, i.e., $s = f(v_1, \ldots, v_n)\sigma$ and $t = f(v_1, \ldots, v_n)\tau$ are weakly unifiable.

Conversely, suppose $\mu$ is a weak unifier of $s$ and $t$, i.e., $v\sigma\mu = v\tau\eta\mu$ for all $v \in V$ and some renaming substitution $\eta$. Let $\lambda$ be a substitution such that $v\sigma\mu\lambda$ is ground for all $v \in V$. Now consider the substitution $\theta = (\sigma\mu\lambda)_{/V}$. $\theta$ is a ground substitution in $\text{GS}_V$ and a $V$-instance of both $\sigma$ and $\tau$. Therefore, we have $\theta \in (\text{GS}_V(\sigma) \cap \text{GS}_V(\tau))$, i.e., $\sigma$ and $\tau$ are compatible. \(\square\)

**Corollary 12.** If $V$ and $\Sigma$ are finite then it can be effectively tested whether some substitution belongs to some $V$-anticover of $\Sigma$ or not.

**Proof.** $\sigma$ belongs to some anticover of $\Sigma$ iff it is incompatible with $\Sigma$, iff it is incompatible with every substitution in $\Sigma$. The corollary now follows from the finiteness of $\Sigma$ and Lemma 11. \(\square\)

### 6. Cover stories: properties of (anti)covers

The key concept in defining both the non-ground stable semantics as well as the non-ground well-founded semantics is the notion of a cover. Hence, this concept needs to be more carefully investigated. In this and in the next section, we study various computational properties related to covers. In particular, we will develop methods to compute (maximal) covers, we will study issues concerning the finiteness of covers, we
will characterize the size of a cover, and will define certain *optimal* covers, studying their properties, too.

6.1. Uniqueness of optimal covers

In general, a set of ground substitutions has several covers that may differ noticeably from each other. Before studying further computational issues, it is thus useful to see if we can formally identify an *optimal* cover for each given set of substitutions. In this section we show that such an optimal cover always exists and is unique up to variants.

Recall the following definitions from Sections 2.3 and 3. For two substitutions $\sigma$ and $\sigma'$, we write $\sigma <_v \sigma'$ iff there exists a substitution $\lambda$ such that $\sigma =_v \sigma' \lambda$; we write $\sigma \equiv_v \sigma'$ iff $\sigma <_v \sigma'$ and $\sigma' <_v \sigma$. For two sets $\Sigma$ and $\Sigma'$ of substitutions, we write $\Sigma \preceq_v \Sigma'$ iff for every $\sigma \in \Sigma$ there is a substitution $\sigma' \in \Sigma'$ such that $\sigma <_v \sigma'$; we write $\Sigma \sim_v \Sigma'$ iff $\Sigma \preceq_v \Sigma'$ and $\Sigma' \preceq_v \Sigma$. Let $\mathcal{G}$ be a set of ground substitutions with domain $V$. $\Sigma$ is a cover of $\mathcal{G}$ iff $\text{GS}_v(\Sigma) = \mathcal{G}$; it is a maximal cover iff for every cover $\Sigma'$ with $\Sigma \preceq_v \Sigma'$, $\Sigma \sim_v \Sigma'$. $\Sigma$ is an optimal cover of $\mathcal{G}$ iff it is maximal and for any two substitutions $\sigma, \tau \in \Sigma$, $\sigma <_v \tau$ implies $\sigma = \tau$.

**Definition 10.** Let $\mathcal{G}$ be a set of ground substitutions with domain $V$. A substitution $\sigma$ is called covering w.r.t. $V$ iff $\text{GS}_v(\sigma) \subseteq \mathcal{G}$. The set of all covering substitutions is denoted by $\text{CS}(\mathcal{G})$. The subset of all $<_v$-maximal elements in $\text{CS}(\mathcal{G})$ is denoted by $\text{maxCS}(\mathcal{G})$. The quotient set $\text{maxCS}(\mathcal{G})/\equiv_v$ is $\text{maxCS}(\mathcal{G})$ partitioned into equivalence classes w.r.t. $\equiv_v$.

The set of all $V$-covers of $\mathcal{G}$ is denoted by $\text{COV}(\mathcal{G})$. The covers in $\text{COV}(\mathcal{G})$ fall into equivalence classes modulo $\sim_v$; the set of all classes is denoted by $\text{COV}(\mathcal{G})/\sim_v$. The partial order $\preceq_v$ can be extended to these equivalence classes in the usual way.

Note that $\mathcal{G} \in \text{COV}(\mathcal{G})$ and $\mathcal{G} \subseteq \text{CS}(\mathcal{G})$; more generally, every cover of $\mathcal{G}$ is a subset of $\text{CS}(\mathcal{G})$.

**Lemma 13.** Let $V$ be a set of variables. For each set $\mathcal{G}$ of ground substitutions with domain $V$ there exists precisely one $\preceq_v$-maximal equivalence class in the quotient set $\text{COV}(\mathcal{G})/\sim_v$.

**Proof.** Let $\Sigma$ be a set of substitutions obtained by choosing one element from each class in $\text{maxCS}(\mathcal{G})/\equiv_v$. Obviously, $\Sigma$ is a cover of $\mathcal{G}$. Moreover, each substitution in $\text{CS}(\mathcal{G})$ is an instance of some substitution in $\Sigma$. Therefore, each cover $\Sigma'$ of $\mathcal{G}$ fulfills $\Sigma' \preceq_v \Sigma$, i.e., $\Sigma$ is a maximal cover.

Let $\Sigma$ be the equivalence class of $\Sigma$ in $\text{COV}(\mathcal{G})/\sim_v$. We show that $\Sigma$ is the unique $\preceq_v$-maximal class in $\text{COV}(\mathcal{G})/\sim_v$. The maximality of $\Sigma$ follows immediately from the maximality of $\Sigma$. Now suppose $\Sigma'$ is another maximal class. Then it has to contain a maximal cover $\Sigma'$. But since $\Sigma' \preceq_v \Sigma$ we have $\Sigma' \sim_v \Sigma$ and thus $\Sigma' = \Sigma$. \[\square\]
We have thus shown that modulo \( \sim_v \) there is a unique maximal cover for each set of ground substitutions with domain \( V \). However, maximality of covers is not fully satisfactory. Maximal covers may contain a great deal of useless information such as thousands of variants or instances of the same substitution. In fact, observe that the union of two or more maximal covers is again a maximal cover. Therefore we now turn to optimal covers. The next proposition states an interesting characterization of optimal covers.

**Proposition 14.** A \( V \)-cover \( \Sigma \) of \( \mathcal{H} \) is optimal iff it is an exact hitting set of the family of sets \( \text{maxCS}(\mathcal{H})_{\equiv_v} \), i.e., iff each substitution in \( \Sigma \) occurs in some class of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \) and each class of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \) has exactly one element in common with \( \Sigma \).

**Proof.** The if-direction is already implicit in the proof of Lemma 13. If \( \Sigma \) is an exact hitting set of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \), then it is a maximal cover. But then, since \( \Sigma \) does not contain any substitution being an instance of another one in \( \Sigma \), it is also optimal.

To show the converse, let \( \Sigma \) be an optimal anticover. First assume that \( \Sigma \) contains some substitution \( \sigma \in \text{CS}(\mathcal{H}) \) not belonging to any class of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \). Since \( \sigma \) is not maximal, there must exist a class \( \bar{\tau} \) in \( \text{maxCS}(\mathcal{H})_{\equiv_v} \) and a substitution \( \tau \in \bar{\tau} \) such that \( \sigma \) is a \( V \)-instance of \( \tau \). Now consider the cover \( \Sigma' = (\Sigma - \{\sigma\}) \cup \{\tau\} \). Clearly \( \Sigma \preceq_v \Sigma' \). However, \( \Sigma' \not\preceq_v \Sigma \) for otherwise \( \Sigma \) would contain a substitution \( \lambda \) more general than \( \tau \) and thus also more general than \( \sigma \), contradicting optimality. But from \( \Sigma \preceq_v \Sigma' \) and \( \Sigma' \not\preceq_v \Sigma \) it follows that \( \Sigma \) is not maximal and hence cannot be optimal. Contradiction. Therefore, each substitution in \( \Sigma \) belongs to some class of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \).

We now prove that each optimal cover \( \Sigma \) must contain at least one element from each class in \( \text{maxCS}(\mathcal{H})_{\equiv_v} \). Assume there is a class \( \bar{\tau} \) in \( \text{maxCS}(\mathcal{H})_{\equiv_v} \) such that \( \bar{\tau} \cap \Sigma = \emptyset \). Let \( \tau \) be an arbitrary element of \( \bar{\tau} \). Let \( \Sigma' = \Sigma \cup \{\tau\} \). Again we have \( \Sigma \preceq_v \Sigma' \) but \( \Sigma' \not\preceq_v \Sigma \) contradicting the maximality of \( \Sigma \). Therefore \( \Sigma \) contains at least one element from each class in \( \text{maxCS}(\mathcal{H})_{\equiv_v} \).

Moreover, since \( \Sigma \) is optimal it cannot contain a pair of variants. Hence \( \Sigma \) contains precisely one element from each class in \( \text{maxCS}(\mathcal{H})_{\equiv_v} \). Thus \( \Sigma \) is an exact hitting set of \( \text{maxCS}(\mathcal{H})_{\equiv_v} \). □

In order to compare two covers at a fine level of granularity, we need the following definition.

**Definition 11.** Let \( V \) be a set of variables and \( \Sigma \) and \( \Sigma' \) be sets of substitutions. \( \Sigma \) and \( \Sigma' \) are equal up to \( V \)-variants iff there is a bijection \( f : \Sigma \rightarrow \Sigma' \) such that for each \( \sigma \in \Sigma \), \( \sigma \) and \( f(\sigma) \) are \( V \)-variants. 4

4Note that it then also follows that for each \( \sigma' \in \Sigma' \), \( f^{-1}(\sigma') \) and \( \sigma' \) are \( V \)-variants.
If two covers are equal up to variants, they are almost identical. In particular, they are of the same cardinality.

**Theorem 15.** For each set \( \mathcal{G} \) of ground substitutions with domain \( V \), all optimal covers of \( \mathcal{G} \) are equal up to variants. In other words, there is a unique optimal cover for \( \mathcal{G} \) up to variants.

**Proof.** This theorem follows immediately from our characterization of optimal covers in Proposition 14. In fact, by that theorem, two optimal covers of \( \mathcal{G} \) may differ only insofar as they may contain different representatives of some equivalence classes of \( \text{maxCS}(\mathcal{G}) / \equiv_{V} \). But, by the definition of \( \text{maxCS}(\mathcal{G}) / \equiv_{V} \), all representatives of an equivalence class are \( V \)-variants. \( \square \)

Let us have a closer look at variants. Actually, it turns out that if \( \sigma \) and \( \sigma' \) are variants, then \( \sigma \) can be obtained from \( \sigma' \) by a mere renaming of variables, and vice versa.

**Definition 12 (Variable renaming).** Let \( V, W \) be sets of variables. A substitution \( \lambda \) is a variable renaming on \( W \) iff
1. for all \( v \in W \), \( v\lambda \) is variable (not necessarily in \( W \)), and
2. for all \( v, v' \in W \), \( v \neq v' \) implies \( v\lambda \neq v'\lambda \).

Two substitutions \( \sigma \) and \( \sigma' \) are equal on \( V \) up to variable renaming iff there is variable renaming \( \lambda \) on the set \( W = \text{var}(\text{img}_V(\sigma)) \) such that \( \sigma\lambda = \sigma'\).

Two sets \( \Sigma, \Sigma' \) of substitutions are equal on \( V \) up to variable renaming iff there is a bijection \( f: \Sigma \rightarrow \Sigma' \) such that for each \( \sigma \in \Sigma \), \( \sigma \) and \( f(\sigma) \) are equal on \( V \) up to variable renaming.

**Lemma 16.** If two substitutions are \( V \)-variants then they are equal on \( V \) up to variable renaming.

**Proof.** If \( \sigma \) and \( \sigma' \) are \( V \)-variants then, by definition, there are substitutions \( \lambda, \lambda' \) such that \( \sigma = \sigma'\lambda' \) and \( \sigma\lambda = \sigma' \). We show that \( \lambda \) is a variable renaming on \( W = \text{var}(\text{img}_V(\sigma)) \).

From \( \sigma = \sigma'\lambda' \) and \( \sigma\lambda = \sigma' \) we obtain \( \sigma = \sigma\lambda\lambda' \), i.e., \( \lambda\lambda' \) acts as identity on the variables in \( \text{img}_V(\sigma) \). Now suppose that for some variable \( v \in W \), \( v\lambda \) is no variable, i.e., \( v\lambda = f(\cdot) \) where \( f \) is a constant or a function symbol. But then \( \lambda' \) has to satisfy \( f(\cdot)\lambda' = v \) which clearly is impossible. Hence \( v\lambda \) has to be a variable for all \( v \in W \).

The second criterion that \( \lambda \) has to fulfill in order to be a variable renaming is that for all \( v, v' \in W \), \( v \neq v' \) implies \( v\lambda \neq v'\lambda \). Suppose that \( v \neq v' \) but \( v\lambda = v'\lambda = w \). This means that \( \lambda' \) has to satisfy \( w\lambda' = v \) and \( w\lambda' = v' \) at the same time, which only is

---

5 Remember that \( \text{img}_V(\sigma) \) is the multiset \( \{x\sigma \mid x \in V \} \) (see also Section 2.3).
possible for \( v = v' \). Contradiction. We conclude that \( \lambda \) is a variable renaming on \( W \), and thus \( \sigma \) and \( \sigma' \) are equal on \( V \) up to variable renaming. \( \square \)

**Corollary 17.** Each set of ground substitutions with domain \( V \) has an optimal \( V \)-cover unique up to variable renaming.

Note that even optimal covers may be rather large in the worst case. Actually, even in the function-free case it is possible that the size of a smallest possible cover is of a magnitude comparable to the size of the ground instantiation of a program, i.e., exponential. This is shown by the following example.

**Example 7.** Let the function-free vocabulary consist of constant symbols \( a_1, \ldots, a_c \) where \( c > n \), and let \( V = \{ X_1, \ldots, X_n \} \). Consider the set \( E \) of substitutions defined by \( E = \{ \{ X_i \leftarrow a_i \} \mid 1 \leq i \leq n \} \), which could be the bad set of a clause with negated atoms \( p_i(X_i) \), \( 1 \leq i \leq n \), w.r.t. to the \( S \)-interpretation \( \{ p_i(a_i) \mid 1 \leq i \leq n \} \). The unique optimal anticover \( A \) of \( E \) is the trivial anticover, i.e., it is the set of all substitutions \( \{ X_i \leftarrow t_i \mid 1 \leq i \leq n \} \) such that \( t_i \in \{ a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_c \} \). Note that the cardinality of \( A \) is \((c - 1)^n\). Thus, the size of the anticover is exponential in the number of substitutions having as base the number of constants. The anticover can be effectively computed in time \( pol(c^n) \) for some polynomial \( pol \), and thus the complexity is determined up to a polynomial function.

### 6.2. Infinite anticovers

As we have seen in Section 5, it is possible to construct effectively a finite cover for the bad substitutions of some clause – provided the \( S \)-interpretation under consideration is finite – by computing the bad set of the clause w.r.t. the \( S \)-interpretation. A natural question to ask is whether finite bad sets possess finite anticovers. This question is of practical relevance since the anticovers of bad sets are nothing but covers for the good substitutions, which are needed for the computation of generalized GL-transforms.

However, as the example below shows, even for singleton sets of substitutions there may be only anticovers of infinite cardinality. Of course, this is not surprising since it is well known that in the presence of function symbols there may be uncountably many stable models.

**Example 8.** Consider a vocabulary with two constants \( a \) and \( b \), a single unary function symbol \( f \), and infinitely many variable symbols \( \{ X, Y, Z, \ldots \} \). Let \( SL = \{ p(Z, f(Z)) \} \) and let \( C = q(X, Y) \leftarrow \neg p(X, Y) \). The set of bad substitutions is given by

\[
\mathcal{B}_{C, SL} = \{ \{ X \leftarrow t, Y \leftarrow f(t) \} \mid t = f^n(a) \text{ or } t = f^n(b), n \geq 0 \}.
\]

\( \text{bad}(C, SL) \) consists of the single substitution \( \{ X \leftarrow Z, Y \leftarrow f(Z) \} \), which is the mgwu of \( p(X, Y) \) and \( p(Z, f(Z)) \). Obviously, \( \text{bad}(C, SL) \) is a finite \( V \)-cover of \( \mathcal{B}_{C, SL} \) where \( V = \text{var}(C) = \{ X, Y \} \).
All anticovers of $\text{bad}(C, SI)$ are of infinite cardinality. To see this assume that some anticover, $\mathcal{A}$, is finite. $\mathcal{A}$ has to cover all substitutions in

$$\mathcal{B}_{C, SI} = \mathcal{B}_C \setminus \mathcal{A}_{C, SI}$$

$$= \{ \{X \mapsto t_1, Y \mapsto f(t_2)\} \mid t_1, t_2 \text{ are ground terms with } t_1 \neq t_2\}$$

$$\cup \{\{X \mapsto t_1, Y \mapsto t_2\} \mid t_1 \text{ is a ground term and } t_2 \in \{a, b\}\}.$$  

In particular, $\mathcal{A}$ has to cover $\theta_n = \{X \mapsto f^n(a), Y \mapsto f(f^n(b))\}$ for all $n \geq 0$. Since $\mathcal{A}$ is finite there must be some substitution $\sigma \in \mathcal{A}$ and some infinite set $I$ of natural numbers such that $\lambda_i$ is a $V$-instance of $\sigma$ for all $i \in I$. But this implies that $\lambda_i = \{X \mapsto f(U), Y \mapsto f(f(V))\}$ is a $V$-instance of $\sigma$, too, where $U$ and $V$ are new variables. Moreover, every ground instance of $\lambda_i$ has to be an instance of $\sigma$, including those where $U$ and $V$ are replaced by the same ground term. These latter substitutions, however, are also instances of $\text{bad}(C, SI)$, i.e., $\mathcal{B}_V(\mathcal{A}) \cap \mathcal{B}_V(\text{bad}(C, SI)) \neq \emptyset$. This contradicts the assumption that $\mathcal{A}$ is an anticover of the bad set. We conclude that $\mathcal{A}$ cannot be finite.

Note that the anticovers remain infinite even if $SI$ contains no function symbols. Take for instance $SI = \{p(Z, Z)\}$. It is easy to see that our arguments from above still apply.

The above example suggests that difficulties with the finiteness of anticovers arise in cases where some variable occurs twice in the range of a substitution, like $Z$ in $\text{bad}(C, SI)$. The next theorem clarifies this intuition.

**Definition 13 (Linear substitutions).** Let $V$ be a set of variables. A substitution $\sigma$ is called **linear on $V$** iff no variable occurs more than once in $\text{img}_V(\sigma)$. A set of substitutions is linear iff each of its elements is linear.

**Example 9.** Let $V = \{X, Y\}$. The substitution $\sigma_1 = \{U \mapsto Z, Y \mapsto f(Z)\}$ is linear since $Z$ occurs only once in $\text{img}_V(\sigma_1) = [X, f(Z)]$. On the other hand, neither of the three substitutions $\sigma_2 = \{X \mapsto Z, Y \mapsto f(Z)\}$, $\sigma_3 = \{Y \mapsto g(Z, Z)\}$ and $\sigma_4 = \{Y \mapsto f(X)\}$ is linear. In the first two cases $Z$ occurs twice in $\text{img}_V(\sigma_2) = [Z, f(Z)]$ and $\text{img}_V(\sigma_3) = [X, g(Z, Z)]$, respectively. In the last case the culprit is variable $X$ since it occurs twice in $\text{img}_V(\sigma_4) = [X, f(X)]$.

**Theorem 18.** Let the set of function and constant symbols in the underlying language be finite and contain at least one function and one constant symbol. Furthermore, let $V$ be a set of variables and $\sigma$ be a substitution. Then $\{\sigma\}$ has a finite $V$-anticover iff it is linear on $V$.

**Proof.** The fact that $\{\sigma\}$ has a finite $V$-anticover if it is linear on $V$ is a consequence of Lemma 28 in Section 7. It remains to show the converse. Suppose that $\sigma$ is not linear on $V$. We show that every $V$-anticover of $\{\sigma\}$ has to be infinite. Wlog we assume
dom(σ) ⊆ V. In the following, f and a denote a function and a constant symbol, respectively.

Since σ is not linear there is some variable z occurring at least twice in \( \text{img}_V(σ) \). Let x be a variable such that \( xσ \) contains z, say at position p. Let t be a term identical to \( xσ \), except that the subterm of t at position p is a new variable \( z' \) – occurring neither in V nor in σ – instead of z. Note that t may still contain the variable z, since only one of its occurrences is replaced by \( z' \). Furthermore, let \( η = \{ v \mapsto a \mid v \in \text{var}(\text{img}_V(σ)) - \{z\} \} \).

We define a new substitution by

\[
λ(v) = \begin{cases} 
  tη & \text{for } v = x \\
  vση & \text{otherwise}
\end{cases}
\]

λ can be obtained from σ by first replacing every variable except z by the constant a and then replacing one occurrence of z by \( z' \). In other words, \( λ \{z' \mapsto z\} = vση \).

Now consider substitutions of the form \( λ_1,λ_2 = \{Z \mapsto t_1, Z \mapsto t_2\} \).

If \( t_1 = t_2 \) then \( λ_1,λ_2 \) is an instance of σ since

\[
λ \{z \mapsto t_1, z' \mapsto t_2\} = λ \{z \mapsto t_1\} \{z \mapsto t_1\} = vση \{z \mapsto t_1\}.
\]

If, additionally, \( t_1 \) and \( t_2 \) are ground then \( λ_1,λ_2 \in \text{GS}_V(σ) \). If, on the other hand, \( t_1 \) and \( t_2 \) are ground terms with \( t_1 \neq t_2 \) then \( λ_1,λ_2 \in \text{GS}_V - \text{GS}_V(σ) \). In particular, for each \( n > 0 \) the substitution

\[
θ_n = λ_{f^n(a)},f^{2n}(a) = \{z \mapsto f^n(a), z' \mapsto f^{2n}(a)\}
\]

is in \( \text{GS}_V - \text{GS}_V(σ) \). Assume that there is a finite \( V \)-anticover \( A \) of \( \{σ\} \). By definition, \( A \) covers all substitutions in \( \text{GS}_V - \text{GS}_V(σ) \). Since \( A \) is finite, there has to be a substitution \( τ \in A \) and an infinite set \( I \) of positive integers such that for each \( i \in I \), \( θ_i \) is a \( V \)-instance of \( τ \). Let \( j \) and \( j' \) be two elements of \( I \) with \( j < j' \). The fact that \( θ_j \) and \( θ_{j'} \) both are instances of \( τ \) implies that their least common generalization, \( λ_{f^j(u)},f^{j'}(w) \), is a \( V \)-instance of \( τ \), too, where u and w are new variables. Moreover, every ground instance of this substitution has to be an instance of \( τ \), including that one where u is replaced by \( f^j(a) \) and w by a, i.e., including the substitution \( λ \{z \mapsto f^j(a), z' \mapsto f^{j'}(a)\} \). However, as we have seen above, this substitution is also an instance of \( σ \), i.e., \( \text{GS}_V(AC) \cap \text{GS}_V(\{σ\}) \neq \emptyset \). This contradicts the assumption that \( A \) is an anticover of \( \{σ\} \). We conclude that \( A \) cannot be finite. ∎

Theorem 18 deals with singleton sets \( \{σ\} \) of substitutions. Does this result carry over to the case of general sets \( \{σ_1, \ldots, σ_n\} \) of substitutions? Interestingly, as we will show in the next two examples, it does not. The reason is that non-linear substitutions may – in a sense – be overruled by other substitutions, and thus loose their detrimental effects. Therefore, there exist sets of substitutions having finite anticovers even though they contain non-linear substitutions.

**Example 10.** Let \( V = \{X, Y\} \), \( σ_1 = \{X \mapsto Z, Y \mapsto Z\} \), and \( σ_2 = \{\} \). Furthermore, let \( \Sigma = \{σ_1, σ_2\} \). Clearly, any substitution whatsoever is an instance of \( σ_2 \). Therefore, a
A V-anticover of $\Sigma$ is given by the empty set. Thus $\Sigma$ has a finite anticover even though $\sigma_1$ is non-linear on $V$.

Note that $\Sigma$ in the above example is equivalent to $\{\sigma_2\}$ containing solely the empty substitution which is linear. This simple example may suggest that it could suffice to eliminate subsumed substitutions and then apply our linearity criterion in order to decide whether the set has a finite or an infinite anticover. However, we must disappoint the reader again: As the following example shows, there exist sets containing non-linear non-subsumed substitutions having a finite anticover.

**Example 11.** Consider a language whose set of constant symbols is $\{a, b\}$ and whose set of function symbols is $\{f\}$. Let $V = \{X, Y\}$, $\sigma_1 = \{X \mapsto f(Z), Y \mapsto U\}$, $\sigma_2 = \{X \mapsto U, Y \mapsto f(Z)\}$, and $\sigma_3 = \{X \mapsto Z, Y \mapsto Z\}$. Furthermore, let $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$. Obviously, $\sigma_3$ is not linear. Moreover, $\sigma_3$ is not a $V$-instance of any other substitution in $\Sigma$. Notwithstanding, the set $\mathcal{A} = \{\tau_1, \tau_2\}$, where $\tau_1 = \{X \mapsto a, Y \mapsto b\}$ and $\tau_2 = \{X \mapsto b, Y \mapsto a\}$, is a finite $V$-anticover of $\Sigma$.

Although the presence of a non-linear substitution usually impedes finite anticovers, the above example shows that there exist sophisticated cases where the presence of a non-subsumed and non-linear substitution does not imply the infiniteness of the anticover. A precise characterization of these cases is outside the scope of this paper and will be carried-out elsewhere. On the other hand, as we will show in Section 7, sets of linear substitutions always have finite anticovers.

### 6.3. Recursiveness of anticovers

As we have seen above, even when the bad set is finite, all of its anticovers may be of infinite cardinality. This section shows that each finite set of substitutions has at least a recursive optimal $V$-anticover, provided that $V$ is finite. In other words, one can specify an optimal anticover $\mathcal{A}$ by supplying an algorithm which decides for each substitution $\sigma$ whether $\sigma$ belongs to $\mathcal{A}$ or not. We will develop such an algorithm in the present section.

Given a finite set $\Sigma$ of substitutions and a substitution $\sigma$, the algorithm returns true if $\sigma$ is in a specific optimal anticover of $\Sigma$ called canonical anticover, and false otherwise. Basically, the algorithm generates substitutions up to the size of $\sigma$ and tests whether they subsume $\sigma$. What we need to formulate the algorithm is the following:

- an appropriate definition of the size of a substitution;
- an effective way of constructing representatives of all substitutions smaller than $\sigma$;
- a definition of canonical anticover.

In Section 6.1 we have shown that optimal covers are unique up to variants, or equivalently, up to variable renaming. In this section we identify one particular optimal

---

6 This is no restriction since usually $V$ is the set of variables occurring in a clause or program, and thus is finite.
cover, the so-called canonical cover. For this purpose we introduce an ordering on \( V \)-
variants based on a total ordering on variables.

**Definition 14.** Let \( \sqsubseteq \) be a total ordering on all variables, and let \( V = \{v_1, \ldots, v_n\} \)
be a finite set of variables where \( v_i \sqsubseteq v_{i+1} \) for \( 1 \leq i \leq n - 1 \). We extend \( \sqsubseteq \) to an
ordering on substitutions in the following way. For all substitutions \( \sigma, \tau \) such that
\( \sigma \) and \( \tau \) are \( V \)-variants and \( (\text{dom}(\sigma) \cup \text{dom}(\tau)) \subseteq V \), let \( \sigma \sqsubseteq \tau \) iff \((v_1\sigma, \ldots, v_n\tau)\) is
smaller than \((\sigma \sigma, \ldots, v_n\tau)\) when interpreting the tuples as strings and comparing them
lexicographically based on the ordering on variables.\(^7\) Additionally, \( \sigma/\tau \sqsubseteq \sigma \) for all
substitutions \( \sigma \) with \( \sigma/\tau \neq \sigma \).

**Lemma 19.** For all substitutions \( \sigma \) and all finite sets \( V \) of variables, there is a unique
\( V \)-variant of \( \sigma \) which is minimal w.r.t. \( \sqsubseteq \). Moreover, for a given substitution this
minimal variant can be computed in time \( O(lm) \), where \( l \) is the total number of
occurrences of constant, function and variable symbols in \( \text{img}_V(\sigma) \) and \( m \) is the
number of different variables occurring in \( \text{img}_V(\sigma) \).

**Proof.** For an arbitrary substitution \( \sigma \), let \( \sigma' \) be the restriction of \( \sigma \) to \( V \). Obviously
\( \sigma' \) is a \( V \)-variant of \( \sigma \) satisfying \( \text{dom}(\sigma') \subseteq V \) and either \( \sigma' = \sigma \) or \( \sigma' \sqsubseteq \sigma \). Now
consider the set of all variants \( \tau \) of \( \sigma' \) fulfilling \( \text{dom}(\tau) \subseteq V \). The ordering \( \sqsubseteq \) is total
on this set; it inherits this property from the lexicographic ordering on tuples. Hence
the minimal variant exists and is unique.

The minimal variant of \( \sigma \) can be computed by the following algorithm. Let \( V = \{v_1, \ldots, v_n\} \). Mark all variables in the tuple \((v_1\sigma, \ldots, v_n\sigma)\) as not-yet-processed. Scanning
it from left to right find the first marked variable, say \( v \). Replace all marked occurrences
of \( v \) by the smallest variable w.r.t. to the ordering on variables. This new variable
remains unmarked. Now repeat the process: scan the changed tuple from left to right,
find the next marked variable and replace all its marked occurrences by the second
smallest variable, and so on. After \( m \) iterations all marked variables have been replaced
by new variables. The resulting tuple represents the minimal variant. In each cycle we
have to scan the entire tuple which has a length proportional to the total number of
symbol occurrences. Hence the algorithm is of time complexity \( O(lm) \). \( \square \)

**Example 12.** Let \( V = \{X, Y, Z\} \), \( \sigma = \{X \mapsto f(W, U), Z \mapsto W, U \mapsto a\} \), and let \( \sqsubset \) be
(partially) defined by \( U \sqsubset X \sqsubset V \sqsubset Y \sqsubset W \sqsubset Z \sqsubset \cdots \).

We start with the tuple \((f(W^*, U^*), Y^*, W^*)\) where all variables have been marked by \(*\).
Starting from the left, the first marked variable is \( W^* \), which we replace by the smallest
variable, viz. \( U \). We obtain \((f(U, U^*), Y^*, U)\). In the next iteration all occurrences of \( U^* \)
are replaced by the second smallest variable, \( X \); the last iteration replaces the remaining
marked variable, \( Y^* \), by \( V \). We end up with the unmarked tuple \((f(U, X), V, U)\), which

\(^7\) Note that the tuples only differ in places where they contain variables; hence there is no need to define an
ordering on the other symbols.
represents the substitution \( \sigma' = \{X \mapsto f(U, X), Y \mapsto V, Z \mapsto U \} \). \( \sigma' \) is a \( V \)-variant of \( \sigma \), since
\[
\sigma' =_V \sigma \{U \mapsto X, W \mapsto U, Y \mapsto V\}
\]
and
\[
\sigma =_V \sigma' \{U \mapsto W, V \mapsto Y, X \mapsto U\}.
\]
Furthermore, \( \sigma' \) is minimal w.r.t. \( \subset_V \).

**Definition 15** (Canonical cover). The canonical cover of a set \( \mathcal{G} \) of ground substitutions with domain \( V \), denoted by \( \text{CanC}_V(\mathcal{G}) \), is that one among the optimal covers of \( \mathcal{G} \), where all substitutions are minimal w.r.t. \( \subset_V \). For a set \( \Sigma \) of substitutions, let \( \text{CanAC}_V(\Sigma) \) denote the canonical anticover of \( \Sigma \), i.e., \( \text{CanAC}_V(\Sigma) = \text{CanC}_V(GS_V - GS_V(\Sigma)) \).

Our next step towards the algorithm is a proper definition of the size of a substitution.

**Definition 16** (Size of substitutions). Let \( V \) be a finite set of variables and \( \sigma \) be a substitution. The size of \( \sigma \) w.r.t. \( V \), denoted by \( |\sigma|_V \), is defined as \( l - m \), where \( l \) is the total number of occurrences of constant, function and variable symbols in \( \text{img}_V(\sigma) \) and \( m \) is the number of different variables occurring in \( \text{img}_V(\sigma) \).

**Example 13.** Let \( V = \{X, Y, Z\} \) and \( \sigma = \{X \mapsto f(W, U), Z \mapsto W, U \mapsto a\} \). The image of \( V \) under \( \sigma \) is given by the multiset \([f(W, U), Y, W]\). For the total number of symbols we obtain \( l = 5 \), for the number of different variables \( m = 3 \). Hence \( |\sigma|_V = 5 - 3 = 2 \).

**Lemma 20.** For all finite sets of variables and all substitutions \( \sigma \), \( |\sigma|_V \geq 0 \). Furthermore, for every \( V \)-instance \( \tau \) of \( \sigma \), \( |\tau|_V = |\sigma|_V \) where \( |\tau|_V = |\sigma|_V \) holds iff \( \tau \) is a \( V \)-variant of \( \sigma \).

**Definition 17.** Let \( V \) be a finite set of variables. The set \( \text{SUBS}_V^{(k)} \), \( k \geq 0 \), consists of all substitutions having size \( k \) w.r.t. \( V \). The set \( \text{BASE}_V^{(k)} \) consists of all substitutions in \( \text{SUBS}_V^{(k)} \) which are minimal w.r.t. \( \subset_V \).

**Example 14.** Let \( V = \{X, Y\} \) and let \( U \subset V \subset \cdots \subset X \subset Y \subset \cdots \). Furthermore, assume that there are only the unary function symbol \( f \) and the constant symbol \( a \). We obtain:

\[
\text{BASE}_V^{(0)} = \{\{X \mapsto U, Y \mapsto V\}\}
\]

\[
\text{BASE}_V^{(1)} = \{\{X \mapsto a, Y \mapsto U\}, \{X \mapsto U, Y \mapsto a\}, \{X \mapsto U, Y \mapsto U\}, \{X \mapsto f(U), Y \mapsto V\}, \{X \mapsto U, Y \mapsto f(V)\}\}
\]

\[
\text{BASE}_V^{(2)} = \{\{X \mapsto a, Y \mapsto a\}, \{X \mapsto a, Y \mapsto f(U)\}, \{X \mapsto f(U), Y \mapsto a\}, \{X \mapsto U, Y \mapsto f(a)\}, \{X \mapsto f(a), Y \mapsto U\}, \{X \mapsto f(U), Y \mapsto f(V)\}, \{X \mapsto U, Y \mapsto f(f(V))\}, \{X \mapsto f(f(U)), Y \mapsto V\}, \{X \mapsto U, Y \mapsto f(U)\}, \{X \mapsto f(U), Y \mapsto U\}\}
\]
Table 2
Function testing membership in the canonical anticover

```plaintext
function InCanAC (σ: substitution;
    Σ: finite set of substitutions;
    V: finite set of variables;
    □: total ordering on V): boolean;

(* Returns true, if σ belongs to CanAC(Σ), *)
(* and false otherwise. *)
If σ is compatible with Σ then return false;
If G is not minimal w.r.t. Ev then return false;
for i := 0 to Icrlw-1 do
    for all
        BASE~ ) do
            if z is incompatible with Z and
            is a V-instance of v then return false;
    return true;
end function;
```

Note that BASE~ ) contains no redundancies, since all substitutions in it are minimal variants.

**Lemma 21.** For finite V and k ≥ 0, BASE~ ) is finite, can be effectively constructed, and contains for each substitution in SUBS~ ) its minimal variant.

**Proof.** By definition, the size k of a substitution is computed as l - m, where l is the total number of symbols occurring in the multiset img_v, and m is the number of different variables in img_v. Since m cannot be smaller than 0, a substitution of size k contains at most k function and constant symbols. The number of different variables is bounded by the maximal number of function symbols, their maximal arity and the cardinality of img_v which is that one of V. BASE~ ) contains only minimal substitutions, i.e., the variables used therein are the minimal ones w.r.t. □ and the domain of the substitutions is a subset of V. Hence the substitutions in BASE~ ) are built of a finite vocabulary, i.e., BASE~ ) is finite. According to Lemma 19 the minimal variant of a given substitution can be effectively computed, hence BASE~ ) can be effectively constructed, too. Finally, by Lemma 20, all variants of a substitution are of the same size, hence SUBS~ ) contains for every substitution of size k also its minimal variant, which by definition belongs to BASE~ ).

**Corollary 22.** For each set G of ground substitutions and each σ ∈ CanC(Σ), σ is in BASE~ ) where k = |σ|_V.

We now have all ingredients for our algorithm, which is listed in Table 2.

**Theorem 23 (Correctness of InCAC).** Let σ be a substitution, Σ be a finite set of substitutions, and V be a finite set of variables totally ordered by □. Then InCanAC(σ, Σ, V, □) terminates and returns true iff σ is in the canonical V-anticover of Σ.
Proof. The termination of InCanAC(σ, Σ, V, □) follows from the finiteness of all sets involved and from Corollary 12 and Lemmas 19 and 21.

σ is in an anticover of Σ iff it is incompatible with Σ. Furthermore, by the definition of canonical anticover, each of its members has to be minimal w.r.t. □V. Hence the algorithm may return false if σ does not meet any of these two conditions.

It remains to check that σ is maximal, i.e., that it is not a proper V-instance of some other substitution in an anticover. By Lemma 20 it suffices to check substitutions which are strictly smaller in size. By Lemma 21, for every substitution of size k some V-variant is in the finite set BASE_k^V. Therefore the algorithm generates BASE_k^V for all sizes k smaller than |σ|_V and picks all substitutions in BASE_k^V which are incompatible with Σ, i.e., which belong to some anticover of Σ. If any of these substitutions subsumes σ, the algorithm returns false since σ is not maximal and thus cannot belong to CanAC_V(Σ). Otherwise, at the end of both for-loops, it may return true. □

Corollary 24. CanAC_V(Σ) is a recursive set.

The next theorem shows that the membership problem for CanAC_V(Σ) lies in co-NP.

Theorem 25. Checking σ ∈ CanAC_V(Σ) is in co-NP.

Proof. We show that the complementary problem, σ ∉ CanAC_V(Σ), is in NP. By our remarks in the proof of Theorem 23, it is immediate that the following NP-procedure can be used to show that σ ∉ CanAC_V(Σ).

First check whether σ is compatible with some substitution in Σ. If yes, σ ∉ CanAC_V(Σ) holds. Otherwise, compute the minimal variant of σ. If it is different from σ, σ ∉ CanAC_V(Σ) holds. Both tests can be done in polynomial time; see [14] and Lemma 19, respectively.

Finally, check whether there exists a substitution in CanAC_V(Σ) which is strictly more general than σ. To check this, guess a substitution τ of size smaller than σ, which is minimal w.r.t. □V as well as incompatible with Σ, such that σ is a V-instance of τ. All these properties of τ can be checked in polynomial time. The problem is thus in NP. □

It is currently open whether this problem is NP-complete. Note that the following, related problem is also in co-NP: decide whether a substitution σ is an element of some optimal anticover of Σ. In fact, it suffices to compute the minimal variant σ′ of σ, which can be done in polynomial time, and then check whether σ′ ∈ CanAC(Σ).

7. An improved anticover computation technique

In this section we develop a more efficient strategy to compute anticovers. The strategy is sound for all logic programs, but is complete only for a fragment of logic
programs. Fortunately, this fragment is quite large and seems adequate for a wide variety of knowledge representation problems.

To see how our algorithm works, consider a set $\Sigma$ containing a single substitution $\sigma$. Each substitution $\tau$ belonging to a $V$-anticover of $\Sigma$ is incompatible with $\sigma$ on $V$, i.e., $\sigma$ and $\tau$ are not weakly unifiable (see Section 5). Consequently, an anticover of $\Sigma$ can be obtained by first finding, for each $v \in V$, a set of terms not weakly unifiable with $v\sigma$ such that the entire spectrum of non-unifiable terms is covered, and then combining these sets appropriately. This intuition is formalized in the definitions below. In the sequel, we will assume that the constant symbols in our language are $a_1, \ldots, a_k$ and that the function symbols are $f_1, \ldots, f_m$. The arity of a function symbol $f$ will be denoted by $\text{ar}(f)$. The set of base terms is the set

$$\text{BTS} = \{a_1, \ldots, a_k, f_1(\vec{y}_1), \ldots, f_m(\vec{y}_m)\}$$

where $\vec{y}_i = (x_1, \ldots, x_{\text{ar}(f_i)})$ is a vector containing $\text{ar}(f_i)$ distinct variables. For every function symbol $f$, BTS contains only one term $f(\vec{y})$, which can be viewed as representing all of its variants.

**Definition 18 (Anti-term set).** The anti-term set associated with a term $t$, denoted by $\text{ATS}(t)$, is inductively defined as:

- $\text{ATS}(v) = \emptyset$ for a variable $v$,
- $\text{ATS}(a) = \text{BTS} - \{a\}$ for a constant symbol $a$,
- $\text{ATS}(f(t_1, \ldots, t_{\text{ar}(f)})) = (\text{BTS} - \{f(\vec{y})\}) \cup \bigcup_{i=0}^{\text{ar}(f)} A_i$

where

$$A_i = \{f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_{\text{ar}(f)}) \mid t \in \text{ATS}(t_i)\}$$

(The $x_j$ are variables different from each other not occurring in $\text{ATS}(t_i)$.)

Anti-term sets are only unique up to renaming. Nevertheless, we speak of the anti-term set associated with some term, since each element in the anti-term set will always be treated as representative of the class of all its variants. Note that, by construction, no variable occurs more than once in a particular element of the anti-term set.

**Example 15.** Suppose our language contains the constant symbols $a$, $b$ and $c$ and the function symbols $f$, $g$ and $h$ of arities 1, 2, and 3, respectively. For the set of base terms we obtain $\text{BTS} = \{a, b, c, f(X), g(X,Y), h(X,Y,Z)\}$.

The anti-term set of the constant symbol $a$ is the set BTS with $a$ removed. It is easy to see that each ground term not weakly unifiable with $a$ is an instance of some term in $\text{ATS}(a)$. Furthermore, none of the instances of any term in $\text{ATS}(a)$ is weakly unifiable with $a$. 
Now consider the term $g(b, x)$. $ATS\,(g(b, x))$ is the union of $(BTS - \{g(X, Y)\})$, $A_1$ and $A_2$, where

$$A_1 = \{g(a, u), g(c, u), g(f(x), u), g(g(X, Y), u), g(h(X, Y, Z), u)\}$$

and

$$A_2 = \emptyset .$$

$A_2$ is empty since $ATS(X) = \emptyset$. As in the previous example, $g(b, x)$ and the terms in $ATS(g(b, x))$ have no common ground instances, and every ground term non-unifiable with $g(b, x)$ is an instance of some term in $ATS(g(b, x))$.

As a final example consider the term $g(X, X)$. Its anti-term set is just $BTS - \{g(X, Y)\}$ since $A_1 = A_2 = ATS(X) = \emptyset$. Again $g(X, X)$ and $ATS(g(X, X))$ share no common ground instances. Contrary to the examples above, however, there are ground terms which are neither an instance of $g(X, X)$ nor of $ATS(g(X, X))$, like the term $g(a, b)$.

The difference between the first two terms, $a$ and $g(b, x)$, and the term in the last example, $g(X, X)$, is that the latter contains two occurrences of the same variable. This leads us to the following definition.

**Definition 19** (Linear terms). A term $t$ is linear iff no variable symbol occurs more than once in $t$.

**Lemma 26.**
(a) For every term $t$, if $s \in ATS(t)$, then $s$ is not weakly unifiable with $t$.
(b) Let $s$ and $t$ be linear terms. If $s$ is not weakly unifiable with $t$, then there is a term $s' \in ATS(t)$ such that $s$ is an instance of $s'$.

**Proof.** By induction on the depth of $t$. The depth of a variable or a constant symbol is defined to be zero, and the depth of a functional term $f(t_1, \ldots, t_n)$ is $(d + 1)$ where $d$ is the maximum of the depths of $t_1, \ldots, t_n$.

(a) **Base case.** The depth of $t$ being zero implies that $t$ is either a variable or a constant symbol. In the first case the assertion trivially holds since $ATS(t) = \emptyset$. In the second case $t = a$ for some constant symbol $a$, and $ATS(t) = BTS - \{a\}$. All terms in $ATS(t)$ are headed by a constant or function symbol different from $a$, hence none of them is weakly unifiable with $t$.

**Inductive case.** The depth of $t$ being greater than zero implies that $t = f(t_1, \ldots, t_n)$ for some function symbol $f$. By definition, $s \in ATS(t)$ either belongs to $BTS - \{f(\vec{y})\}$ or it occurs in $A_i$ for some $i$. In the first case $s$ is headed by a function symbol different from $f$, and thus is not weakly unifiable with $t$. In the second case $s$ is of the form $f(\ldots, x_{i-1}, t', x_{i+1}, \ldots)$ where $t' \in ATS(t_i)$. Since the depth of $t_i$ is smaller than that of $t$, we conclude by the induction hypothesis that $t'$ is not weakly unifiable with $t_i$. Therefore the same holds for $s$ and $t$.

(b) **Base case.** $t$ being of depth zero and the precondition that $s$ and $t$ are not weakly unifiable imply that $t$ is a constant symbol and that $s$ is either a constant symbol different from $t$ or a functional term. It is not hard to see that $s$ is subsumed by some base term in $ATS(t) = BTS - \{t\}$.
Inductive case. $t$ being of depth greater than zero implies that $t = f(t_1, \ldots, t_n)$ for some function symbol $f$. Since $s$ and $t$ both are linear there are only two possibilities for $s$ in order to be not weakly unifiable with $t$: either $s$ is headed by a constant or function symbol different from $f$, or $s$ is of the form $s = f(s_1, \ldots, s_n)$ and for some $i$, $s_i$ is not weakly unifiable with $t_i$. In the first case it is not hard to see that $s$ is subsumed by some base term in BTS-$\{f(\bar{y})\}$. In the second case we note that $t_i$ is smaller in depth than $t$, and therefore, by the induction hypothesis, there is a term $s'_i \in$ ATS($t_i$) such that $s_i$ is an instance of $s'_i$. Now consider the term $s' = f(x_1, \ldots, x_{i-1}, s'_i, x_{i+1}, \ldots, x_{ar(f)})$, where the $x_i$ are variables different from each other. Each of the variables $x_j, j \neq i$, is more general than $s_j$, and, as stated above, $s'_i$ is more general than $s_i$. Hence $s$ is an instance of $s'$. But $s'$ (or one of its variants) is also in the anti-term set of $t$, which concludes the proof. \[ \Box \]

To see that the restriction to linear terms in the second part of the theorem is necessary, consider the following example.

**Example 16.** Let $t = f(X, X)$ and $s = f(a, b)$. Clearly, $s$ and $t$ are not weakly unifiable. However, the anti-term set ATS($t$) does not contain any term headed by $f$, hence there is no $s' \in$ ATS($t$) subsuming $s$.

The following lemmas show how anti-term sets are related to anticovers.

**Lemma 27.** Let $\sigma$ be a substitution. Then each substitution of the form $\{v \mapsto s\}$, where $s \in$ ATS($v\sigma$), is incompatible with $\sigma$ on any (finite) set of variables containing $v$.

**Proof.** Assume the contrary, i.e., assume that $\sigma$ is compatible with $\tau = \{v \mapsto s\}$ on $V = \{v_1, \ldots, v_n\}$. Wlog we assume $v = v_1$. Then, by Lemma 11, the tuples $(v_1\sigma, \ldots, v_n\sigma)$ and $(v_1\tau, \ldots, v_n\tau)$ have to be weakly unifiable. This implies that the terms $v_1\sigma = v\sigma$ and $v_1\tau = s$ are weakly unifiable, too. Contradiction to Lemma 26(a). \[ \Box \]

For a finite set $V$ of variables and a substitution $\sigma$, we define

$$\text{linAC}_V(\sigma) = \{\{v \mapsto s\} \mid v \in V, s \in \text{ATS}(v\sigma)\}.$$ 

One could hope that the collection of all substitutions $\{v \mapsto s\}$, i.e. $\text{linAC}_V(\sigma)$, would form an anticover of $\sigma$. However, this is not true in general.

**Example 17.** Suppose our language contains two constant symbols $a, b$ and a unary function symbol $f$. Let $\sigma = \{X \mapsto Z, Y \mapsto f(Z)\}$ and $V = \{X, Y\}$. Since ATS($Z$) = $\emptyset$ and ATS($f(Z)$) = $\{a, b\}$, we obtain $\text{linAC}_V(\sigma) = \{\tau_1, \tau_2\}$ where $\tau_1 = \{Y \mapsto a\}$ and $\tau_2 = \{Y \mapsto b\}$. Both substitutions are incompatible with $\sigma$, but they form no complete $V$-anticover; the ground substitution $\{X \mapsto a, Y \mapsto f(b)\}$ is neither a $V$-instance of $\sigma$, nor of $\tau_1$, nor of $\tau_2$. 
The matter is different as long as we restrict ourselves to linear substitutions.

**Lemma 28.** Let \( V \) be a finite set of variables, and let \( \sigma \) be a substitution linear on \( V \). Then \( \text{linAC}_V(\sigma) \) is a finite \( V \)-anticover of \( \{\sigma\} \).

**Proof.** According to Lemma 27, each of the substitutions \( \{v \mapsto s \} \) is incompatible with \( \sigma \) on \( V \), i.e., \( \text{GS}_V(\{v \mapsto s \}) \cap \text{GS}_V(\sigma) = \emptyset \). It remains to show that every ground substitution \( \theta \in \text{GS}_V \), which is no \( V \)-instance of \( \sigma \), is a \( V \)-instance of some substitution \( \{v \mapsto s\} \).

Since \( \sigma \) is linear, \( \theta \) being no \( V \)-instance of \( \sigma \) implies that there is some variable \( v \in V \) such that \( v\theta \) is no instance of \( v\sigma \). Both terms, \( v\theta \) as well as \( v\sigma \), are linear: the first one because it is ground, and the second one because \( \sigma \) is linear on \( V \). Furthermore, it is not hard to see that \( v\theta \) and \( v\sigma \) are not weakly unifiable; otherwise \( v\theta \) would be an instance of \( v\sigma \). Hence, by Lemma 26(b), there is a term \( s' \in \text{ATS}(v\sigma) \) having \( v\theta \) as instance. Clearly, this implies that \( \theta \) is a \( V \)-instance of \( \{v \mapsto s'\} \).

The finiteness of the anticover follows from the finiteness of \( V \) and from the finiteness of \( \text{ATS}(t) \) for arbitrary \( t \). □

The method for computing the anticover of a singleton set can be used to find anticovers for a set \( \Sigma \) containing more than one substitution. The idea is the following. Each ground substitution \( \theta \) represented by the anticover is incompatible with every substitution in \( \Sigma \), and therefore has to be an instance of any anticover of \( \{\sigma\} \) for each \( \sigma \in \Sigma \). What we thus need is some kind of intersection of these latter anticovers. The elements of anticovers are non-ground substitutions; as intersection of two such elements we use the most general substitution which is an instance of both. Variables occurring in the range of both substitutions have to be treated as different. Hence the process of finding the most general substitution is nothing but weak unification.

Let \( \circ_V \) denote an operator on substitutions defined by \( \sigma \circ_V \tau = \sigma \mu \) where \( \mu \) is a weak most general unifier of \( \sigma \) and \( \tau \) w.r.t. \( V \). In other words, \( \sigma \circ_V \tau \) is obtained by first renaming the variables in \( t = f(v_1, \ldots, v_n)\tau \), where \( V = \{v_1, \ldots, v_n\} \), such that \( t \) and \( s = f(v_1, \ldots, v_n)\sigma \) have no variables in common, then computing a most general unifier of \( s \) and the renamed version of \( t \), and finally applying it to \( \sigma \). If no (weak) most general unifier exists then \( \sigma \circ_V \tau \) is undefined.

Note that the result of \( \circ_V \) is only unique up to \( V \)-variants. However, when viewing \( \circ_V \) as operating on equivalence classes modulo \( \equiv_V \) rather than on single substitutions, it is a well-defined function which is commutative, associative and idempotent.

**Lemma 29.** A substitution \( \theta \) is a \( V \)-instance of all \( \tau_i \), \( 1 \leq i \leq n \), iff it is a \( V \)-instance of \( \tau_1 \circ_V \cdots \circ_V \tau_n \).

**Definition 20 (Linear anticover).** Let \( V \) be a finite set of variables and \( \Sigma = \{\sigma_1, \ldots, \sigma_n\} \) be a finite set of substitutions. The linear anticover of \( \Sigma \) is defined as

\[
\text{linAC}_V(\Sigma) = \{\tau_1 \circ_V \cdots \circ_V \tau_n \mid \tau_i \in \text{linAC}_V(\sigma_i) \text{ for } 1 \leq i \leq n\}.
\]
Example 18. Suppose our language contains one constant symbol \( a \) and one binary function symbol \( f \). Let \( \Sigma = \{ \sigma_1, \sigma_2 \} \) where \( \sigma_1 = \{ X \mapsto f(a,Y) \} \) and \( \sigma_2 = \{ X \mapsto f(Y,a) \} \), and let \( V = \{ X \} \). The anti-term set of \( f(a,Y) \) is given by \( \text{ATS}(f(a,Y)) = \{ a, f(f(X,Y),Z) \} \). Therefore we obtain

\[
\text{linAC}_V(\sigma_1) = \{ \{ X \mapsto a \}, \{ X \mapsto f(f(X,Y),Z) \} \}
\]

and similarly,

\[
\text{linAC}_V(\sigma_2) = \{ \{ X \mapsto a \}, \{ X \mapsto f(X,f(Y,Z)) \} \}.
\]

To compute \( \text{linAC}_V(\Sigma) \) we have to combine each substitution in \( \text{linAC}_V(\sigma_1) \) with each one in \( \text{linAC}_V(\sigma_2) \) using the operator \( \odot_V \). Since \( a \) is not unifiable with any term headed by \( f \), only two of the four combinations contribute to the final set:

\[
\text{linAC}_V(\Sigma) = \{ \{ X \mapsto a \}, \{ X \mapsto f(f(X,Y),f(U,V)) \} \}
\]

It is not hard to see that \( \text{linAC}_V(\Sigma) \) is a (finite) \( V \)-anticover of \( \Sigma \).

Theorem 30. Let \( V \) be a finite set of variables, and let \( \Sigma \) be a finite set of substitutions all of which are linear on \( V \). Then \( \text{linAC}_V(\Sigma) \) is a finite \( V \)-anticover of \( \Sigma \).

Proof. The finiteness of \( \text{linAC}_V(\Sigma) \) follows from the finiteness of all sets participating in the construction of \( \text{linAC}_V(\Sigma) \).

By definition, every substitution \( \lambda \) in \( \text{linAC}_V(\Sigma) \) is an instance of some substitution in \( \text{linAC}_V(\sigma) \), for all \( \sigma \in \Sigma \). In other words, \( \lambda \) is incompatible with every \( \sigma \in \Sigma \). Therefore every ground instance of \( \lambda \) with domain \( V \) is incompatible with \( \Sigma \). We thus have \( \text{GS}_V(\text{linAC}_V(\Sigma)) \cap \text{GS}_V(\Sigma) = \emptyset \).

It remains to show that every ground substitution \( \theta \), which is incompatible with all substitutions in \( \Sigma = \{ \sigma_1, \ldots, \sigma_n \} \), is represented by \( \text{linAC}_V(\Sigma) \), i.e., that \( \text{GS}_V(\text{linAC}_V(\Sigma)) \cup \text{GS}_V(\Sigma) = \text{GS}_V \). Since \( \theta \) is incompatible with \( \sigma_i \), there has to be a substitution \( \tau_i \in \text{linAC}_V(\sigma_i) \) such that \( \theta \) is an instance of \( \tau_i \). But then, by Lemma 29, \( \theta \) is also an instance of \( \tau_1 \odot_V \cdots \odot_V \tau_n \). By definition, the latter is in \( \text{linAC}_V(\Sigma) \). \( \square \)

As we have seen above, linear substitutions are particularly well-behaved concerning finite anticovers. This raises the question: “When computing the stable and/or well-founded semantics of a logic program, under what circumstances will we encounter linear substitutions? Is there a class of programs and/or S-interpretations such that \( \text{linAC} \) can be used to compute anticovers?” Below, we identify conditions on programs and S-interpretations guaranteeing linearity for the substitutions encountered during the computation of generalized GL-transforms.

Definition 21 (Linear atoms). An atom is linear iff no variable symbol occurs more than once in it. An S-interpretation is linear iff each atom in it is linear.

Lemma 31. Let \( A \) and \( B \) be atoms. If \( B \) is linear then \( \text{mgw}_V(A,B) \) is linear on \( V \).
Proof. The most general weak unifier is defined to be the most general unifier \( \mu \) of \( A \) and \( B\eta \) for an appropriate renaming substitution \( \eta \). Let \( F \) be the frontier of \( A \) and \( B\eta \).\(^8\) \( F \) is a set of pairs having the form \((u, t)\) or \((s, v)\), where \( u, v \) are variables and \( s, t \) stand for arbitrary terms. \( u \) and \( s \) are subterms of \( A \), whereas \( t \) and \( v \) are the corresponding subterms of \( B\eta \) in the same position as \( u \) and \( s \), respectively. \( \mu \) is a most general unifier of \( A \) and \( B\eta \) iff it is a most general unifier of all these pairs.

First consider pairs of the form \((s, v)\). Since \( B\eta \) is linear, there is at most one such pair for each variable \( v \) in the frontier, i.e., \((s, v), (s', v) \in F \) implies \( s = s' \). Hence these variables do not introduce additional variable bindings by requiring the unification of \( s \) and \( s' \). Furthermore, we have \( v \notin V \) by the definition of \( \eta \). As a consequence, we may disregard this kind of pairs when investigating the linearity of \( \mu \), since linearity is defined via the image of \( \mu \), and obviously \( s \notin \text{img}_\varphi(\mu) \).

Now consider pairs of the form \((u, t)\). There may be several pairs with the same variable \( u \), say \((u, t_1), \ldots, (u, t_n)\), since \( A \) is not required to be linear. These pairs lead to the unification of \( t_1, \ldots, t_n \); let \( \sigma \) be their most general unifier. Note that the effects of this unification are purely local: because of the linearity of \( B \) each variable occurs at most once in at most one of the terms. Hence each variable in the domain of \( \sigma \) occurs in exactly one of the \( t_i \) and nowhere else in \( F \). Thus each variable \( u \) – provided it belongs to \( V \) – contributes a term of the form \( t_1 \sigma \) to \( \text{img}_\varphi(\mu) \). We conclude that each variable occurs at most once in \( \text{img}_\varphi(\mu) \), i.e., \( \mu \) is linear on \( V \). \( \square \)

**Corollary 32.** If \( SI \) is a linear \( S \)-interpretation then for any clause \( C \), \( \text{bad}(C, SI) \) is linear on \( \text{var}(C) \).

The corollary guarantees that it suffices to check whether \( SI \) is linear when computing the generalized Gelfond–Lifschitz transform of a clause \( C \) w.r.t. \( SI \). If this is the case, then the linear anticover of \( \text{bad}(C, SI) \) can be used as cover for the good substitutions, \( G_{C, SI} \).

Going one step further one may ask under which circumstances the linearity of \( S \)-interpretations is preserved when computing the fixpoint of a program.

**Definition 22 (Linear clauses).** A clause is linear, iff the atom in its head is linear. A program is linear iff each of its clauses is linear.

**Proposition 33.** Let \( SI \) be a linear \( S \)-interpretation.

(a) If \( C \) is a linear clause, then any generalized GL-transform of \([C]\) w.r.t. \( SI \) based on a linear cover is a linear (negation-free) program.

(b) If \( P \) is a linear, negation-free program, then \( W_P(SI) \) is a linear \( S \)-interpretation.

---

\(^8\)The frontier of two terms \( s \) and \( t \) is the set of all pairs of corresponding subterms in \( s \) and \( t \) where at least one of the subterms is a variable. E.g., the frontier of \( f(g(a), g(x)) \) and \( f(g(y), Z) \) is the set \( \{(a, y), (g(x), Z)\} \). The frontier can be considered as the repeated application of the decomposition rule; it is undefined if there are any clashes of function symbols. For a formal definition see [14].
Now the wheel comes full circle. Starting from a linear S-interpretation \( SI \) and a linear program \( P \), we know by Corollary 32 that for each clause in \( P \) its bad set is linear. By Theorem 30, \( \text{linAC} \) can be used to compute anticovers for linear bad sets, which themselves are linear by the definition of \( \text{linAC} \). Using these linear anticovers, Proposition 33(a) guarantees that the generalized GL-transform of a linear program (with negation) is a linear program (without negation). Finally, the W-operator corresponding to the latter program again yields a linear S-interpretation when applied to \( SI \) (Proposition 33(b)).

However, the observant reader will note that there is a gap concerning the stable and well-founded semantics of Section 4. Both are based on the operator \( \text{ngF} \), which is defined via the W-operator applied to the non-ground GL-transform of a program. In other words, it is not enough to consider generalized GL-transforms based on arbitrary linear covers, rather we have to investigate the linearity of optimal covers forming the basis of non-ground GL-transforms. The question to answer is: "Given an arbitrary linear (anti)cover, is the corresponding optimal (anti)cover linear, too?" Unfortunately, the answer is negative, even in the function-free case as the following example shows.

**Example 19.** Suppose our language contains just two constant symbols, a and b, and no function symbols. Let \( V = \{X, Y\} \) and \( \Sigma = \{\{X \leftarrow a, Y \leftarrow b\}, \{X \leftarrow b, Y \leftarrow a\}\} \). The linear \( V \)-anticover of \( \Sigma \) is given by

\[
\text{linAC}_V(\Sigma) = \{\{X \leftarrow a, Y \leftarrow a\}, \{X \leftarrow b, Y \leftarrow b\}\},
\]

which is not optimal. \( \mathcal{A} = \{\{X \leftarrow Y\}\} \) covers the same ground substitutions as \( \text{linAC}_V(\Sigma) \), but is more general. In fact, it is not hard to see that it is an optimal anticover of \( \Sigma \). However, \( \mathcal{A} \) is not linear since \( Y \) occurs twice in \( \text{img}_V(\{X \leftarrow Y\}) = [Y, Y] \).

One way out is to base the definition of \( \text{ngF} \) on generalized GL-transforms using non-optimal covers, using e.g. linear covers. For appropriate classes of covers and the corresponding \( \text{genG} \)-operator the results of Section 4 will still hold. In this way it is possible to trade non-optimality for the effective computability of (anti)covers.

Another possibility is presented in the next section for datalog programs. By definition, these programs contain no function symbols. We will describe a method of computing finite anticovers for non-linear, but function-free sets of substitutions.

**8. The datalog case**

In this section, we show how anticovers may be computed when dealing with datalog programs (for an overview of datalog, the reader may consult Ullman [16] or Ceri et al. [4]). In general, there are many substitutions that are not linear, and hence \( \text{linAC} \) as defined in the last section cannot be applied to find a complete anticover. This is even true when considering a datalog language. However, because of the lack of
function symbols, we will be able to characterize the missing substitutions not captured by \linAC.

By definition, datalog languages contain no function symbols. As a consequence, all substitutions are of the form \( \sigma = \{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \} \) where the \( t_i \) are either constant symbols or variables. Let \( \mathcal{C} \) be the set of all constants in the language. The anti-term set of a term \( t \) in a datalog language is particularly simple. If \( t \) is a variable then \( \text{ATS}(t) = \emptyset \); if it is a constant symbol then \( \text{ATS}(t) = \mathcal{C} - \{ t \} \). Thus we obtain
\[
\text{linAC}_V(\sigma) = \{ \{ v \mapsto c \} \mid v \in V, v \sigma \in \mathcal{C}, c \in (\mathcal{C} - \{ v\sigma \}) \}.
\]

If \( \sigma \) is linear we know from the last section that \( \text{linAC}(\sigma) \) is an anticover of \( \{ \sigma \} \). Now suppose \( \sigma \) is non-linear, i.e., some variable occurs twice in \( \text{img}_V(\sigma) \). Since the language contains no function symbols, this means that \( \sigma \) is of the form \( \{ \ldots, x_i \mapsto v, x_j \mapsto v, \ldots \} \) where \( x_i \neq x_j \) and \( v \) is some variable. Obviously all ground substitutions assigning different constant symbols to \( x_i \) and \( x_j \) cannot be instances of \( \sigma \), and thus should be represented by the anticover of \( \{ \sigma \} \). This leads us to the set
\[
\text{nlincAC}_V(\sigma) = \{ \{ v_1 \mapsto c_1, v_2 \mapsto c_2 \} \mid v_1, v_2 \in V, v_1 \neq v_2, v_1 \sigma, v_2 \sigma \in V, v_1 \sigma = v_2 \sigma, c_1, c_2 \in \mathcal{C}, c_1 \neq c_2 \}.
\]

An anticover of \( \sigma \) in a function-free language is now obtained by taking the union of these two sets:
\[
\text{ffAC}_V(\sigma) = \text{linAC}_V(\sigma) \cup \text{nlincAC}_V(\sigma).
\]

**Lemma 34.** Let \( V \) be a finite set of variables, and let \( \sigma \) be a substitution in a function-free language. Then \( \text{ffAC}_V(\sigma) \) is a finite \( V \)-anticover of \( \{ \sigma \} \).

**Proof.** The finiteness of \( \text{ffAC}_V(\sigma) \) follows from the finiteness of all sets participating in the construction of \( \text{ffAC}_V(\sigma) \). For \( \text{ffAC}_V(\sigma) \) to be an anticover of \( \{ \sigma \} \) it has to satisfy two conditions: each substitution in \( \text{ffAC}_V(\sigma) \) has to be incompatible with \( \sigma \), and each ground substitution \( \theta \) with domain \( V \) has to be a \( V \)-instance either of \( \sigma \) or of some substitution in \( \text{ffAC}_V(\sigma) \). The first condition is satisfied by the definition of \( \text{ffAC}_V(\sigma) \), as can be easily verified: none of the substitutions in \( \text{ffAC}_V(\sigma) \) is weakly unifiable with \( \sigma \).

Now let \( \theta \) be a ground substitution with domain \( V \). \( \theta \) may fail to be an instance of \( \sigma \) for two reasons:

1. For some variable \( v \in V \), \( v \theta = c \) is a constant symbol different from the constant \( v \sigma \). But then, by the definition of \( \text{linAC} \), there is a substitution \( \{ v \mapsto c \} \) in \( \text{ffAC}_V(\sigma) \), which obviously is more general on \( V \) than \( \theta \).

2. For two different variables \( v_1, v_2 \in V \), \( v_1 \theta = c_1 \) and \( v_2 \theta = c_2 \) are distinct constants and \( v_1 \sigma = v_2 \sigma \) is some variable. In this case, by the definition of \( \text{nlincAC} \), there is a substitution \( \{ v_1 \mapsto c_1, v_2 \mapsto c_2 \} \) in \( \text{ffAC}_V(\sigma) \), which clearly subsumes \( \theta \). \( \square \)
In general, \( \text{ffAC}_V(\sigma) \) is no optimal cover of \( \{\sigma\} \). Consider the following example.

**Example 20.** Let \( \emptyset = \{a, b\} \), \( V = \{X, Y\} \) and \( \sigma = \{X \mapsto a, Y \mapsto b\} \). We obtain

\[
\text{linAC}_V(\sigma) = \{\{X \mapsto b\}, \{Y \mapsto a\}\} \quad \text{and} \quad \text{nlinAC}_V(\sigma) = \emptyset,
\]

i.e., \( \text{ffAC}_V(\sigma) = \text{linAC}_V(\sigma) \). Now consider the substitution \( \tau = \{X \mapsto Y\} \). It is incompatible with \( \sigma \), but neither more nor less general than any element of \( \text{ffAC}_V(\sigma) \). In fact it can be shown that \( \text{ffAC}_V(\sigma) \cup \{\tau\} \) is the optimal anticover of \( \{\sigma\} \). This example also shows that an optimal cover need not be optimal concerning the number of its substitutions.

Similar to \( \text{linAC} \), \( \text{ffAC} \) can be easily extended to compute anticovers of sets of substitutions.

**Definition 23 (Function-free anticover).** Let \( V \) be a finite set of variables and \( \Sigma = \{\sigma_1, \ldots, \sigma_n\} \) be a finite set of substitutions. The function-free anticover of \( \Sigma \) is defined as

\[
\text{ffAC}_V(\Sigma) = \{\tau_1 \odot_V \cdots \odot_V \tau_n | \tau_i \in \text{ffAC}_V(\sigma_i) \text{ for } 1 \leq i \leq n\}.
\]

**Theorem 35.** Let \( V \) be a finite set of variables, and let \( \Sigma \) be a finite set of substitutions in a function-free language. Then \( \text{ffAC}_V(\Sigma) \) is a finite \( V \)-anticover of \( \Sigma \).

**Proof.** Analogous to the proof of Theorem 30.

We can compute anticovers for function-free programs. But how efficient can such a computation be? Clearly, since the smallest anticover of a set \( \Sigma \) of substitutions may be exponential in \( |\Sigma| \), the computation of an anticover will require both exponential time and exponential space in general.

In such a setting, another interesting question arises. Is it possible to compute anticovers in output-polynomial time? In other words, is it possible to compute anticovers in time polynomial in the size of the resulting anticover? If this were the case, then we could design an algorithm which behaves efficiently in case the anticover is small and which does exponential work only if the anticover is very large (i.e., exponential in \( |\Sigma| \)). In the following, we show that unless \( \text{NP} = \text{P} \) we cannot find an output-polynomial algorithm computing an anticover for a given set of substitutions.

We first show that it is co-NP complete to decide whether a given set \( \Sigma \) of function-free substitutions has the empty set as anticover. Note that if \( \Sigma \) has the empty set as anticover, then this is the unique anticover of \( \Sigma \); in other words, \( \Sigma \) has a non-empty anticover iff all anticovers of \( \Sigma \) are non-empty.

**Theorem 36.** Let \( V \) be a finite set of variables and \( \Sigma \) be a set of substitutions over a function-free vocabulary. The problem \( \text{EMPTYACOVER} \) which consists in deciding whether \( \Sigma \) has an empty \( V \)-anticover is co-NP complete.
Proof. We will show that the complement of \textsc{emptycover}, which we call \textsc{nonemptycover}, is NP-complete.

The empty set is not a \( V \)-anticover of \( \Sigma \) iff there is some ground substitution with domain \( V \) such that \( \sigma \) is not an instance of any substitution in \( \Sigma \). Clearly, if such a \( \sigma \) exists, it can be guessed and verified in NP time. Therefore \textsc{nonemptycover} is in NP.

Let us now prove NP-hardness. We use a transformation from the well-known NP-complete problem \textsc{hitting string} (see [8], problem SR12). This problem is defined as follows. For any string \( s \) of length \( k \), and for any integer \( 1 \leq i \leq k \), we denote by \( s[i] \) the \( i \)th symbol of \( s \).

**INSTANCE:** A finite set \( A \) of strings over the alphabet \( \{0, 1, \bot\} \), all having the same length \( n \).

**QUESTION:** Is there a string \( h \in \{0, 1\}^* \) with \( |h| = n \) such that for each string \( s \in A \) there is some \( i, 1 \leq i \leq n \), for which \( s[i] = h[i] \)? (If so, \( h \) is called a hitting string of \( A \).)

We transform an instance \( I \) of \textsc{hitting string} into an instance \( I' \) of \textsc{nonemptycover} as follows. The vocabulary of \( I' \) is given by \( V = \{x_1, \ldots, x_n\} \) and \( \emptyset = \{0, 1\} \). The set \( \Sigma \) of substitutions in \( I' \) contains for each string \( s \in A \) a substitution \( \sigma(s) = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \) such that for \( 1 \leq i \leq n \), \( t_i = \theta[i] \) if \( s[i] \in \{0, 1\} \), and \( t_i = x_i \) otherwise.

We now prove that \( \Sigma \) has a non-empty anticover iff \( A \) has a hitting string. By \( \bar{\sigma} \) we denote the ‘complement’ of an element in \( \{0, 1\} \), i.e., \( \bar{0} = 1 \) and \( \bar{1} = 0 \).

**If direction.** Let \( h \in \{0, 1\}^* \) be a hitting string of \( A \). Define the ground substitution \( \theta \) by \( \theta = \{v_i \mapsto \bar{h}[j] \mid 1 \leq i \leq n\} \). Each string \( s \in A \) has a component \( s[j] \) for some \( j \) such that \( s[j] \neq \bar{h}[j] \). As a consequence \( \theta \) is incompatible with each substitution \( \sigma(s) \) in \( \Sigma \). Therefore every anticover of \( \Sigma \) has to represent \( \theta \) and thus cannot be empty.

**Only if direction.** Let \( \mathcal{A} \) be a nonempty \( V \)-anticover of \( \Sigma \) and let \( \theta \) be an arbitrary ground \( V \)-instance of some substitution in \( \mathcal{A} \). Define the string \( h \) of length \( n \) by \( h[i] = x_i \theta \) for \( 1 \leq i \leq n \). Consider any substitution \( \sigma(s) \) from \( \Sigma \). Since \( \theta \) is incompatible with \( \sigma(s) \), there is a \( j \) such that \( v_j \theta \) and \( v_j \sigma(s) \) are different elements from \( \{0, 1\} \), i.e., \( v_j \sigma(s) = \bar{v}_j \theta \); otherwise \( \theta \) would be an instance of \( \sigma(s) \). But this means that for the string \( s \in A \) corresponding to \( \sigma(s) \) we have \( s[j] = h[j] \). Thus \( h \) is a hitting string of \( A \). \( \square \)

**Corollary 37.** There is no output-polynomial algorithm for constructing an anticover for a given set \( \Sigma \) of substitutions unless \( \text{NP} = \text{P} \).

9. Related work

Anticovers are very similar in spirit to the notion of disunification. Turi [15] develops an extension of the S-semantics for logic programs with negation (but not when inter-
interpreted in terms of the stable, well-founded or answer set semantics). In his framework, a constrained S-interpretation is a collection of constrained atoms (e.g., the constrained atom \( p(X)[X \neq a] \) may be roughly read as “\( p(X) \) is true for all \( X \) except \( X = a \)”). Likewise, ordinary substitutions are generalized to constrained substitutions; a constrained substitution consists of an ordinary substitution, together with a set of constraints on variables that are not present in the domain of the substitution. A constrained substitution is a disunifier for a constrained atom \( A \) and a constrained S-interpretation \( I \) iff no constrained atom in \( I \) unifies with \( A \). The purpose of disunifiers is almost identical to the purpose of anticovers; there are two significant differences, however. The first is that anticovers apply to sets of substitutions. The second is that the two concepts are represented differently. Turi’s representation adds constraints to atoms. Thus, if we wish to store a particular stable model, we would need to store it as a set of constrained atoms. In our framework a set of non-constrained atoms would be stored, meaning that we can do so very easily in a standard relational DBMS system such as Oracle or Sybase. This is more difficult in Turi’s model. On the other hand, Turi’s model uses a more compact representation of anticovers. However, this compact representation may not be suitable for defining a stable semantics because evaluating whether two constrained S-interpretations are equivalent involves determining whether certain sets \( S_1, \ldots, S_n \) of constraints jointly imply certain other constraints. Checking this can be extremely complex. Hence, in both cases, certain trade-offs are being made, and it may be difficult to evaluate the effect of these trade-offs without an implementation.

An alternative approach to developing a non-ground stable and well-founded semantics could use Turi’s approach in conjunction with work on solving systems of equations and disequations (see for example [1] and [12]) in the following way. An interpretation is given by a set of constrained atoms. When transforming a clause of a logic program w.r.t. the constrained S-interpretation, negation in the body of the clause is replaced by constraints, thus yielding a negation-free constrained logic program. Intuitively, the original constrained S-interpretation is “stable” iff it is “equivalent” (cf. example below) to the least constrained S-model of this negation-free logic program with constraints. While we cannot explain this approach in full detail here (cf. [6] for more on this subject), we can outline the basic idea using an example below.

**Example 21.** Let \( I \) be the following constrained S-interpretation:

\[
\begin{align*}
\{ & \ p(X, Y) \leftarrow X = Y, \\
& \ p(X, Y) \leftarrow X = f(f(Y)), \\
& \ q(X, Y) \leftarrow X \neq Y \& X \neq f(f(Y)).
\end{align*}
\]

Let \( C \) be the clause \( q(U, V) \leftarrow \neg \top(p(U, V)) \). Consider the first constrained atom in \( I \). This atom is true whenever \( X = Y \), meaning that all instances of \( C \) satisfying \( U = V \) should be “thrown out” when performing the Gelfond–Lifschitz transform. Likewise, considering the second atom in \( I \), all instances satisfying \( U = f(f(V)) \) should be eliminated. On the other hand, all instances not satisfying \( U = V \) or \( U = f(f(V)) \)
should be in the program resulting from the Gelfond–Lifschitz transform, with their
negated atoms deleted. Thus, the Gelfond–Lifschitz transform of \( C \) w.r.t. \( I \) is

\[
q(U, V) \leftarrow \neg(U = V \lor U = f(f(V)))
\]

which is the same as

\[
q(U, V) \leftarrow U \neq V \land U \neq f(f(V))
\]

Note that this clause is a variant of the third constrained fact in \( I \).

An important point to note is that in general, in order to check the intuitive notion
of “equivalence” between constraints, it is necessary to check that all constraints in the
least constrained S-model of the non-ground GL-transform of \( P \) w.r.t. \( I \) are implied
by \( I \), and vice versa. This check requires the ability to check that certain sets of
constraints imply others, which can be done using algorithms for solving systems of
(dis)equations like those described comprehensively in [1].

10. Extended logic programs

The idea that logic programs should have a notion of explicit negation was first
introduced by Blair and Subrahmanian [3]. Subsequently, Gelfond and Lifschitz [10]
showed that logic programs containing two kinds of negation – explicit, as well as
non-monotonic negation – are useful in expressing a wide variety of systems.

An extended logic program is a set of clauses of the form

\[
L \leftarrow L_1 \land \cdots \land L_n \land \text{not}(L_{n+1}) \land \cdots \land \text{not}(L_{n+m})
\]

where each of the \( L, L_1, \ldots, L_{n+m} \) is a literal. Given a set \( X \) of ground literals and an
extended logic program \( \Pi \), the GL-transform of \( \Pi \) w.r.t. \( X \) is the ground extended
logic program

\[
\mathcal{GL}(\Pi, X) = \{ D^+ | D \in \text{grd}(\Pi) \text{ and none of the}
\text{negated literals in } D \text{ occurs in } X \}.
\]

\( X \) is said to be an answer set of \( \Pi \) iff it satisfies two conditions:

1. Whenever \( L \leftarrow L_1 \land \cdots \land L_n \) is in \( \mathcal{GL}(\Pi, X) \) and \( \{L_1, \ldots, L_n\} \subseteq X \), then \( L \in X \).
2. If there is an atom \( A \) such that both \( A \) and \( \neg A \) are in \( X \), then \( X \) is the set of all
ground literals expressible in the language.

It is well-known that explicit negation can be eliminated, and that, mathematically
speaking, answer sets can be reduced to stable models by the following construction,
as has been noted by Gelfond and Lifschitz [10, Section 4, “Reduction to General
Programs”].

Step 1: To each predicate symbol \( p \) in \( \Pi \), associate a new predicate symbol \( p' \) of
the same arity.
Step 2: Replace in II all occurrences of \( \neg p(\overline{r}) \) by \( p'(\overline{r}) \), including those prefixed by not.

Step 3: For each pair \( p, q \) of predicate symbols in II add the four clauses

\[
q(\overline{\gamma}) \leftarrow p(\overline{x}) \land p'(\overline{x}), \quad p(\overline{x}) \leftarrow q(\overline{\gamma}) \land q'(\overline{\gamma}), \\
q'(\overline{\gamma}) \leftarrow p(\overline{x}) \land p'(\overline{x}), \quad p'(\overline{x}) \leftarrow q(\overline{\gamma}) \land q'(\overline{\gamma}),
\]

where \( \overline{x} \) and \( \overline{\gamma} \) are disjoint vectors of variable symbols.

Let \( Tr(II) \) denote the program obtained from II by these three steps. It is easy to show that the following result holds.

**Theorem 38** (Gelfond and Lifschitz [11]). Suppose II is an extended logic program.

(a) If \( X \) is an answer set of II, then

\[
\{ p(\overline{r}) \mid p(\overline{r}) \text{ is an atom in } X \} \cup \{ p'(\overline{r}) \mid \neg p(\overline{r}) \text{ is a literal in } X \}
\]

is a stable model of \( Tr(II) \).

(b) If \( M \) is a stable model of \( Tr(II) \), then

\[
\{ p(\overline{r}) \mid p \text{ is un-primed and } p(\overline{r}) \in M \} \cup \{ \neg p(\overline{r}) \mid p'(\overline{r}) \in M \}
\]

is an answer set of II.

This theorem is very important. It shows that given an extended logic program II, the answer sets of II are essentially just the stable models of \( Tr(II) \). Consequently, all the definitions of non-ground stable and well-founded semantics can be applied to extended logic programs in the following way:

1. Transform the extended logic program II to \( Tr(II) \).

2. Compute the non-ground stable models (resp. well-founded semantics) of \( Tr(II) \).

3. If \( X \) is a non-ground stable model of \( Tr(II) \), then the corresponding non-ground answer set of II is obtained by replacing all occurrences of primed atoms \( p'(\overline{r}) \) by \( \neg p(\overline{r}) \).

4. The set of literals true (resp. false) in the well-founded semantics of the extended logic program II can be similarly obtained from the set of atoms true (resp. false) in the well-founded semantics of \( Tr(II) \) by replacing primed atoms \( p'(\overline{r}) \) by \( \neg p(\overline{r}) \).

### 11. Conclusions

There are essentially two semantics for logic programming with non-monotonic modes of negation – the stable semantics [9], and the well-founded semantics [18]. Both characterize the meaning of a program by a set (or a set of sets) of ground atoms. The Gelfond–Lifschitz transform, which plays a fundamental rôle in defining these semantics [2, 17], transforms the ground version of a logic program w.r.t. a set of ground atoms. Our intention is to avoid grounding altogether by defining a non-ground
version of the stable and the well-founded semantics. These semantics are based on a transform – equivalent to the GL-transform in the sense of Theorem 4 and 7 – where a set of (not necessarily ground) clauses is transformed w.r.t. a set of atoms (not necessarily ground either).

The technical key idea underlying this development is that of an (anti)cover. Given a set \( G \) of ground substitutions, a cover of \( G \) is a set of (non-ground) substitutions such that each element in \( G \) (and only these) can be obtained by instantiating some element in the cover. An anticover of a cover is a cover of the complementary set of \( G \), i.e., every ground substitution is represented either by the cover or the anticover, and no ground substitution is represented by both of them.

Based on the notion of (anti)cover, we have shown how a non-ground version of the Gelfond–Lifschitz transform can be defined. Furthermore, we have studied the decidability aspects as well as some aspects related to the computational complexity of computing anticovers. We have developed methods for computing anticovers and have studied some of their properties. We have shown that the resulting definitions of non-ground well-founded semantics and non-ground stable semantics generalize the corresponding ground definitions. As grounding of logic programs can often lead to an explosion in the size of the program, our framework provides a method to circumvent such grounding. At the same time, it leads to a compact representation of stable models through the use of non-ground atoms.

References


