A General Framework for Preferences in Answer Set Programming

Gerhard Brewka\textsuperscript{a}, James Delgrande\textsuperscript{b}, Javier Romero\textsuperscript{c,*}, Torsten Schaub\textsuperscript{c}

\textsuperscript{a}University of Leipzig
\textsuperscript{b}Simon Fraser University
\textsuperscript{c}University of Potsdam

Abstract
We introduce a general, flexible, and extensible framework for quantitative and qualitative preferences among the stable models of logic programs. Since it is straightforward to capture propositional theories and constraint satisfaction problems with logic programs, our approach is also relevant to optimization in satisfiability testing and constraint processing. We show how complex preference relations can be specified through user-defined preference types and their arguments. We describe how preference specifications are handled internally by so-called preference programs, which are used for dominance testing. We also provide algorithms for computing one, or all, preferred stable models of a logic program, and study the complexity of these problems. We implemented our approach in the \textit{asprin} system by means of multi-shot answer set solving technology. We demonstrate the generality and flexibility of our methodology by showing how easily existing preference languages can be implemented in \textit{asprin}. Finally, we empirically evaluate our contributions and contrast them with dedicated implementations.

Keywords: Preferences, Optimization, Answer Set Programming

*Corresponding author

Email addresses: brewka@informatik.uni-leipzig.de (Gerhard Brewka),
jim@cs.sfu.ca (James Delgrande), javier@cs.uni-potsdam.de (Javier Romero),
torstens@cs.uni-potsdam.de (Torsten Schaub)
1. Introduction

Preferences are pervasive and often are a key factor in solving real-world applications. This was realized quite early in Answer Set Programming (ASP; [1]), where solvers offer optimization statements representing ranked, sum-based objective functions (viz. minimize statements or weak constraints [2, 3]). Such quantitative ways of optimization are often insufficient for applications, in particular, when they require more elaborate preference aggregation or even the combination of various types of qualitative and quantitative preferences. This is all the more remarkable, since there is a vast literature on qualitative and hybrid means of optimization [4, 5, 6, 7, 8]. And what makes things worse is that existing systems lack any means for integrating customized preferences.

We bridge this gulf with our approach and the resulting asprin\(^1\) system by providing a general, flexible, and extensible framework for implementing complex combinations of quantitative and qualitative preferences among the stable models of a logic program. Our framework is general and captures the major existing approaches to preference. It is flexible and deals with preferences in an elaboration tolerant way. Also, it is extensible and allows for an easy implementation of new or extended approaches to preference handling. Hence, it provides a uniform setting for comparing and combining preferences. The resulting system asprin builds on control capabilities for multi-shot ASP solving [9], providing successive yet operational grounding and solving of changing logic programs. This technology allows us to direct the search for specific preferred solutions without modifying the ASP solver. As well, it significantly reduces redundancies found in an iterated setting. Finally, the use of ASP technology paves the way for the high customizability of our framework by offering an implementation of preferences via ordinary encodings using meta-programming.

From an abstract point of view, our goal is to determine the preferred stable models of logic programs associated with preference specifications that express preference relations. Formally, a preference relation \(\succeq\) is a preorder, that is, a reflexive and transitive relation. Given two stable models \(X\) and \(Y\) of \(P\), \(X \succeq Y\) means that \(X\) is at least as preferred as \(Y\), or, in other words, that \(X\) is better or equal to \(Y\). The strict version of \(\succeq\) is defined as usual, i.e., \(X \succ Y\) if \(X \succeq Y\) but \(Y \nprec X\). In words, \(X \succ Y\) means that \(X\) is preferred to \(Y\), and we may also say that \(X\) is better than \(Y\) or that \(X\) dominates \(Y\). A stable model \(X\) of \(P\) is preferred with respect to \(\succeq\), if there is no other stable model \(Y\) such that \(Y \succ X\).

\(^{1}\)asprin stands for “ASP for preference handling”.

2
— that is, whenever $X$ is not dominated by any other stable model. Then, the preferred (or optimal) models of a logic program with a preference specification are the stable models of the logic program that are preferred with respect to the preorder expressed by the preference specification. Note that our approach applies as well to optimization in Satisfiability Testing and Constraint Processing, given that programs capturing these paradigms are easily designed (cf. [10, 11]).

asprin allows for declaring and evaluating preference relations among the stable models of a logic program. Preferences are declared by preference statements, composed of an identifier, a type, and an argument set, referred to as preference elements. The identifier names the preference relation, whereas its type and elements define the relation. Here is a simple (propositional) example:

\[
\#\text{preference}(\text{costs}, \text{less(\text{weight})})\{40 : \text{sauna}, 70 : \text{dive}\}
\]

This statement\(^2\) declares a preference relation named costs with type less(weight) and argument set \{40 : sauna, 70 : dive\}. Informally, the resulting preference relation prefers models whose atoms induce the minimum sum of weights. Hence, models with neither sauna nor dive are preferred over those with only sauna. Stable models with only dive are still less preferred, while those with both sauna and dive are least preferred. While the arguments of preference statements are sets, we see below that the elements contained in these sets can be more complex than in the example. In the most general case, we even admit conditional elements, which are used to capture conditional preferences. For instance, a preference element ‘dive > {sauna, swim}‖hot’ may express that, whenever it is hot, diving is preferred to saunaing and swimming. Moreover, some preference statements may refer to other statements in their arguments, for example:

\[
\#\text{preference}(\text{all}, \text{pareto})\{\text{name(costs), name(fun), name(temps)}\}
\]

This defines a preference relation all which is the Pareto ordering of three preference relations costs, fun and temps. That is, stable model $X$ is better or equal to $Y$, if it is better or equal to $Y$ with respect to the three argument orderings.

Since we can have more than one preference statement, one of them must be distinguished for optimization. This is done via an optimization directive of form \#optimize(s) with the name of the respective preference statement as argument. Then, a preference specification is a set of preference statements along with an

---

\(^2\)In ASP, meta statements are preceded by ‘#’. 
optimization directive. It expresses the preference relation given by the corresponding optimized statement. Unlike existing approaches to optimization, this separates the declaration of preference relations from optimization instructions.

Once the preference and optimize statements are given, the computation of preferred stable models is done via preference programs. Such programs, which need to be defined for each preference type, take two (reified) stable models and decide whether one is preferred to the other. During optimization, they take as input a stable model $X$ of the original program $P$ and produce a stable model of $P$ better than $X$, according to the optimization directive, if such a stable model exists. An optimal stable model is computed stepwise, by repeated calls to the ASP solver: first, an arbitrary stable model of $P$ is generated; then this stable model is “fed” to the preference program to produce a better one, etc. Once the solver returns unsatisfiable, the last stable model obtained is an optimal one. To compute many optimal solutions, the solver is extended by a modification of the preference program that eliminates those models worse than the last optimal one. Then, the optimization process is started again. Preferred stable models can also be computed via a translation to disjunctive logic programs. In this case, the preference program is used to check the optimality of the stable models of the original logic program $P$.

asprin provides a library containing a number of predefined, common, preference types along with the necessary preference programs. Users happy with what is available in the library can thus use the available types without having to bother with preference programs at all. However, if the predefined preference types are insufficient, users may define their own types, and so become preference engineers. In this case, they also have to provide the preference programs asprin needs to cope with the new preference types.

Our paper is structured as follows. We start in Section 2 by laying out the formal preliminaries of our approach. With them, we introduce in Section 3 our preference language and its semantic underpinnings. We then show in Section 4 how preference specifications are translated to ASP and how they can be used in conjunction with preference programs to decide preference dominance and related computational problems. These ideas form the basis of the algorithms presented in Section 5. More precisely, we give algorithms for computing one and enumerating all preferred models, respectively, and show that they are sound and complete. Additionally, we study the complexity of the different problems tackled by our algorithms. In Section 6, we introduce the first-order modeling language of asprin, in order to use it in Section 7 to incorporate existing approaches to preference from the literature. Section 8 gives further details of the asprin system. Then,
we report in Section 9 results obtained from series of experiments analyzing various aspects of asprin’s computational performance. Afterwards, in Section 10 we discuss related work, and we conclude in Section 11.

This article is based on the works presented in [12, 13]. We extend those papers in three ways: we provide a complexity analysis of the reasoning tasks addressed by our algorithms, we extend our previous discussion of related work, and we include for the first time the proofs of the old and the new theoretical results. The only technical modification with respect to those papers is that here we define preference relations as preorders and not as strict partial orders, like we did before. This change is not crucial, since one can easily translate a preorder into a strict partial order, and vice versa. But we decided to apply it for two reasons. First, because preorders fit better with our method for combining preferences, since composite preferences, like pareto, usually require the auxiliary preferences to provide a preorder. And second, because in this way our definition of preference relations agrees with most of the definitions from the literature, which makes the comparison with other approaches easier.

2. Background

A term is either a constant, a variable, a tuple or a functional term. A constant is either a string starting with some lowercase letter, or an integer. A variable is a string starting with some uppercase letter. A tuple of arity $n$ has the form $(t_1, \ldots, t_n)$ for some terms $t_1, \ldots, t_n$ and $n \geq 0$. A functional term has the form $f(t_1, \ldots, t_n)$ for some functor $f$ of arity $n \geq 1$ and some terms $t_1, \ldots, t_n$. An atom has the form $p(t_1, \ldots, t_n)$ for some predicate $p$ of arity $n \geq 0$ and some terms $t_1, \ldots, t_n$. An atom $p()$ is likewise represented by $p$ without parentheses. We say that a predicate is unary if its arity is 1, and we say that a term or an atom is ground if it contains no variables.

A logic program $P$ over a set $\mathcal{A}$ of ground atoms is a set of disjunctive rules of the form

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o$$ \hspace{1cm} (3)

and choice rules of the form

$$\{a_1\} \leftarrow a_2, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o$$ \hspace{1cm} (4)

where each $a_i$ is a ground atom in $\mathcal{A}$ for $1 \leq i \leq o$, $\neg$ stands for (default) negation, and we have that $m, n, o \geq 0$ in (3) and $n, o \geq 1$ in (4). A rule as in (3) is called a fact if $m = o = 1$, normal if $m = 1$, and an integrity constraint if $m = 0$. 
The dependency graph of a set of disjunctive rules over \( \mathcal{A} \) has nodes \( \mathcal{A} \), and for every rule of the form (3) it has an edge \( a_i \leftarrow a_j \) for \( 1 \leq i \leq m \) and \( m + 1 \leq j \leq n \), and an edge \( a_i \leftarrow a_j \) for \( 1 \leq i \leq m \) and \( n + 1 \leq j \leq o \). We say that a logic program is disjunctive in general, it is normal if it consists of normal rules, integrity constraints or choice rules, and it is stratified [14] if it consists of normal rules or integrity constraints, and its dependency graph has no cycle involving a negative edge (\( \leftarrow \)). Note that, differently than usual, we allow choice rules in disjunctive and normal programs.

Let \( P \) be a logic program over a set of ground atoms \( \mathcal{A} \). An interpretation of \( P \) is a subset of \( \mathcal{A} \). To specify the stable model semantics [15, 16], we identify every disjunctive rule (3) of \( P \) with the formula 
\[
a_{m+1} \land \cdots \land a_n \land \neg a_{n+1} \land \cdots \land \neg a_o \rightarrow a_1 \lor \cdots \lor a_m
\]
and every choice rule (4) of \( P \) with the formula 
\[
a_2 \land \cdots \land a_n \land \neg a_{n+1} \land \cdots \land \neg a_o \rightarrow a_1 \lor \neg a_1,
\]
where the empty disjunction and the empty conjunction are identified with \( \bot \) and \( \top \), respectively. Then, an interpretation of \( P \) is a stable model of \( P \) if it is a subset-minimal model of the set of formulas that results from replacing in \( P \) every literal not satisfied by \( X \) with \( \bot \).

We say that a logic program is satisfiable if it has some stable models, and it is unsatisfiable otherwise. Every stratified logic program without integrity constraints is satisfiable and has a unique stable model, while every stratified logic program in general is satisfiable if and only if the unique stable model of its normal rules is a model of its integrity constraints [14, 2].

As usual, rules with variables are viewed as shorthands for the set of their ground instances. More formally, the Herbrand universe of a logic program with variables \( P \) consists of all the ground terms constructible from constants and functors appearing in \( P \), and the set of ground instances of a rule \( r \in P \) is the set of all ground rules obtained by replacing all variables in \( r \) by ground terms from the Herbrand universe of \( P \). Given this, we identify a logic program with variables \( P \) with the set of ground instances of its rules.

We use typewriter font to express rules as source code in the input language of the ASP system clingo\(^3\) and stick with the mathematical notation used in (3) at the conceptual level. Also, we sometimes present examples as source code to make them more comprehensible. To ease the use of ASP in practice, several language extensions have been developed, including conditional literals and cardinality constraints [2]. The former are of the form \( a:b_1, \ldots, b_m \), the latter can be written as \( s\{c_1; \ldots; c_n\}t \), where \( a \) and \( b_i \) are possibly default-

---

\(^3\)This language slightly extends the ASP language standard ASP-Core-2 [17].
negated literals and each $c_j$ is a conditional literal; $s$ and $t$ provide lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $b_1, \ldots, b_m$ as a condition. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like $a(X):b(X)$ in a rule’s antecedent expands to the conjunction of all instances of $a(X)$ for which the corresponding instance of $b(X)$ holds. Similarly, $2 \{ a(X) : b(X) \} 4$ is true whenever at least two and at most four instances of $a(X)$ (subject to $b(X)$) are true. Specifically, we rely in the sequel on the input language of the ASP system clingo [18]; further language constructs are explained on the fly.

3. Expressing Preferences

We go beyond plain ASP and deal with logic programs with preferences, that are pairs $(P, S)$ of logic programs $P$ over a set of ground atoms $A$, and preference specifications $S$ over $A$. We define below preference specifications, and for now we just say that they define a preorder $\succeq \subseteq A \times A$ among the interpretations of $P$. We refer to $\succeq$ as a preference relation. Given two interpretations $X, Y \subseteq A$, the relation $X \succeq Y$ means that $X$ is at least as preferred as $Y$, and the corresponding strict version $X \succ Y$ means that $X$ is preferred to $Y$. We say that a stable model $X$ of $P$ is preferred wrt $\succeq$ (or $\succeq$-preferred) if there is no other stable model $Y$ such that $Y \succ X$; that is, if there is no stable model preferred to $X$. Then, $X$ is a preferred stable model of a logic program $P$ with a preference specification $S$ if $X$ is preferred wrt the preference relation $\succeq$ defined by $S$. We also refer to the preferred stable models as optimal stable models. In what follows, we often leave the preference specification implicit, and refer directly to a program with the corresponding preference relation.

In the remainder of this section, we provide a generic preference language for expressing a wide range of preference relations. The primitives of the language are inspired by the literature to provide the best possible coverage of preference structures. To keep our framework open for extensions, we do not fix a set of predefined preferences. Rather we give examples of how well-known preferences can be expressed and implemented (see also Section 7). Many of these are included in asprin’s preference library, which provides basic building blocks for defining new preferences.

We introduce our framework over a given set of ground atoms $A$. Expressions with variables are viewed as shorthands for their ground instances; they are detailed in the context of asprin’s input language in Section 6.
3.1. Syntax

Our language consists of the following parts: weighted formulas, preference elements, preference statements, optimization directives, and preference specifications.

A weighted formula is of the form \( t : \varepsilon \) where \( t \) is a tuple of terms and \( \varepsilon \) is either a Boolean expression \( \phi \) over \( A \) with logical connectives \( \top, \neg, \land, \) and \( \lor \); or a naming atom (see below). We drop the colon and simply write \( \varepsilon \) whenever \( t \) is empty. For expressing composite preferences, we use a dedicated unary naming predicate \( \text{name} \) that allows us to refer to auxiliary preferences. That is, a naming atom, \( \text{name}(s) \), refers to relations associated with a preference statement \( s \) (see below). Examples of weighted formulas include: \( 1 : \text{cuboid}(x) \land a \), and \( 2 : \text{name}(q) \).

A preference element is of the form

\[
\Phi_1 > \cdots > \Phi_m \parallel \phi
\]

where \( \phi \) is a non-weighted formula giving the context, and each \( \Phi_r \) is a set of weighted formulas for \( r = 1, \ldots, m \) and \( m \geq 1 \). Intuitively, \( r \) gives the rank of the respective set of weighted formulas. Preference elements provide a (possible) structure to a set of weighted formulas by giving a means of conditionalization and a symbolic way of defining preorders (in addition to using weights). For convenience, we may drop the surrounding braces of such sets and omit “\( \parallel \phi \)” if \( \phi \) is tautological. Also, we drop “\( > \)” if \( m = 1 \). Hence \( \{a, \neg b\} > c \) stands for \( \{a, \neg b\} > \{c\} \parallel \top \). Similarly, \( 40 : \text{sauna} \) in (1) abbreviates \( \{40 : \text{sauna}\} \parallel \top \).

Preferences are declared by preference statements of the form

\#preference(s, t)\{e_1, \ldots, e_n\}

where \( s \) and \( t \) are ground terms giving the preference name and its type, respectively, and each \( e_j \) is a preference element for \( j = 1, \ldots, n \). The identifier names the preference relation, whereas its type and arguments define the relation, as detailed in Section 3.2 below. In what follows, we sometimes abuse this notation and simply identify a preference statement with its identifier \( s \) and refer to its type by \( t_s \). The preference type determines its admissible sets of preference elements, since their full generality is not always needed. Formally, the domain of a preference type \( t \) is given by \( \text{dom}(t) \). For instance, \( \text{less(weight)} \) in (1) is restricted to

---

4This syntax follows aggregate elements [17, Section 2].

5This notation is inspired by [19].
weighted literals; see Section 3.2 for more examples. A preference type may or may not allow naming atoms, depending on whether it is composite or primitive. For instance, the preference type _pareto_ in (2) is composite, while _less(weight)_ in (1) is primitive.

To take effect, a set of preference statements is accompanied by a single _optimization directive_ of form

\[ \#\text{optimize}(s) \]

that tells a solver to restrict its reasoning mode to the preference relation declared by _s_. The collection of preference statements in a program has to satisfy certain requirements to be useful. We say a set of preference statements _S_ is

- **closed**, if _s_ ∈ _S_ whenever _name(s)_ occurs in _S_, and
- **acyclic**, if the dependency relation induced among preference statements in _S_ by naming atoms is acyclic.

A _preference specification_ is a set of preference statements _S_ along with a single directive _#optimize(s)_ such that _s_ ∈ _S_ and _S_ is acyclic and closed. We call _s_ the _primary_ preference statement in _S_ and refer to statements in _S_ \{ _s_ \} as _auxiliary_.

### 3.2. Semantics

A preference statement like _#preference(s, t)E_ declares a preference relation of preference type _t_ and preference elements _E_. More formally, a preference type _t_ is a function mapping an admissible set of preference elements _E_ ∈ _dom(t)_ to a preorder _t(E) ⊆ A × A_ over _A_. For simplicity, we often denote the relation _t(E)_ by _≥_ and its strict version by _>_.

As a first example of a definition of a preference type, consider a class of simple cardinality-based relations, referred to as _less(card)_:

\[(X, Y) ∈ \text{less(card)}(E) \iff |\{ℓ ∈ E \mid X \models ℓ\}| ≤ |\{ℓ ∈ E \mid Y \models ℓ\}|\]

where _dom(less(card)) = P(\{a, ¬a \mid a ∈ A\})_ and _P(S)_ is the power set of _S_. We observe that preference types are defined for specific kinds of preference elements and come with recipes for interpreting their preference elements.

---

6 \(X \models φ\) is the standard satisfaction relation between an interpretation _X_ and a formula _φ_.

9
The next preference type $\text{more}(\text{weight})$ is similar to the one used by $\text{maximize}$ statements in ASP solvers (cf. Section 7.1):

$$(X, Y) \in \text{more}(\text{weight})(E) \; \text{iff} \; \sum_{(w: \ell) \in E, X \models \ell} w \geq \sum_{(w: \ell) \in E, Y \models \ell} w$$

where $\text{dom}(\text{more}(\text{weight})) = \mathcal{P}(\{w : a, w : \neg a \mid w \in Z, a \in \mathcal{A}\})$.

Here is a further example from asprin’s library, going beyond existing preferences in ASP solvers:

$$(X, Y) \in \text{subset}(E) \; \text{iff} \; \{\ell \in E \mid X \models \ell\} \subseteq \{\ell \in E \mid Y \models \ell\}$$

where $\text{dom}(\text{subset}) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$.

All previous preference types are primitive because they do not refer to auxiliary preferences via naming atoms. More such preference types are defined throughout this paper. See also Section 8 for a list of preferences predefined in asprin’s library.

Composite preferences are meant to capture preference aggregation. The corresponding preference relations are formed inductively from auxiliary preferences referred to via the unary naming predicate $\text{name}$. Given a naming atom $\text{name}(s)$, we let $\succeq_s, \succ_s, =_s, \prec_s, \preceq_s$ refer to relations associated with preference statement $s$. That is, all of them are regrouped via the preference type and share the same preference elements. Although the various relations can be defined in terms of $\succeq_s$, their specific definition is left to the designer of the preference type. We use $\mathcal{N}$ to denote the set of naming atoms.

For example, we can define preference types corresponding to the well-known Pareto principle [20] as well as lexicographic orderings as follows:

$$(X, Y) \in \text{pareto}(E) \; \text{iff} \; \bigwedge_{s \in \text{E}} (X \succeq_s Y) \; (7)$$

where $\text{dom}(\text{pareto}) = \mathcal{P}(\{n \mid n \in \mathcal{N}\})$, and

$$(X, Y) \in \text{lexico}(E) \; \text{iff} \; \bigvee_{w : \text{name}(s) \in E} ((X \succ_s Y) \land \bigwedge_{w \succ w} ((X =_{s'} Y) \lor \bigwedge_{w : \text{name}(s) \in E} (X =_s Y) \; (8)$$

where $\text{dom}(\text{lexico}) = \{Z \in Z \mid \text{if } w : m \in Z \text{ and } w : n \in Z \text{ then } m = n\}$ for $Z = \mathcal{P}(\{w : n \mid w \in \mathbb{N}, n \in \mathcal{N}\})$. 

10
Combining several preference relations of type more\textit{(weight)} via \textit{lexico} amounts to the orderings induced by \textit{maximize} statements in ASP. Note, however, that those statements couple together the declaration of a preference relation and its usage for optimization, while our framework separates them. This is made precise in Section 7.1. An aggregation of \textit{subset} relations with \textit{lexico} is similar to prioritized circumscription [21]. Pareto-based preference relations are beyond existing preferences in ASP solvers. Other composite preference types are easily defined. More evidence of this is given in Section 7 where we show how various approaches from the literature can be captured in our framework. See also Section 8 for a list of composite preferences predefined in \textit{asprin}'s library.

A specific preference relation is obtained by applying a preference type to an admissible set of preference elements. Here are some examples of preference statements with specific preference elements:

\begin{itemize}
\item \textit{preference}(1, \textit{less}(\textit{card}))\{a, \neg b, c\} declares \(X \succeq_1 Y\) as
\[
|\{\ell \in \{a, \neg b, c\} \mid X \models \ell\} | \leq |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\} |
\]

\item \textit{preference}(2, \textit{more}(\textit{weight}))\{1 : a, 2 : \neg b, 3 : c\} declares \(X \succeq_2 Y\) as
\[
\sum_{w: \ell \in \{1:a,2: \neg b,3:c\} \mid X \models \ell} w \geq \sum_{w: \ell \in \{1:a,2: \neg b,3:c\} \mid Y \models \ell} w
\]

\item \textit{preference}(3, \textit{subset})\{a, \neg b, c\} declares \(X \succeq_3 Y\) as
\[
\{\ell \in \{a, \neg b, c\} \mid X \models \ell\} \subseteq \{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}
\]

\item \textit{preference}(4, \textit{pareto})\{\textit{name}(1), \textit{name}(2), \textit{name}(3)\} declares \(X \succeq_4 Y\) as
\[
(X \succeq_1 Y) \land (X \succeq_2 Y) \land (X \succeq_3 Y)
\]

\item \textit{preference}(5, \textit{lexico})\{1 : \textit{name}(1), 2 : \textit{name}(2), 3 : \textit{name}(3)\} declares \(X \succeq_5 Y\) as
\[
(X \succ_3 Y) \lor ((X \equiv_3 Y) \land (X \succ_2 Y)) \lor \\
((X \equiv_3 Y) \land (X \equiv_2 Y) \land (X \succ_1 Y)) \lor \\
((X \equiv_3 Y) \land (X \equiv_2 Y) \land (X \equiv_1 Y))
\]
\end{itemize}
We note that primitive preference relations are uniquely determined by their type and the result of evaluating the underlying preference elements, whereas composite ones additionally depend on auxiliary preference relations that must be provided within the encompassing preference specification.

Finally, we say that a preference specification \( S \) whose primary statement is \( s \) defines the preference relation \( \succeq_s \). Observe that this is well defined also for specifications that contain composite preference statements, since preference specifications are closed and acyclic by definition.

4. Handling Preferences

Our approach centers on the implementation of the decision problem: \((X,Y)\in\succ_s\)? That is, to decide whether one stable model is strictly preferred to another with respect to a preference relation defined by some preference statement. This dominance test is accomplished by so-called preference programs. Such programs need to be defined for each preference type. Their combination with facts representing preference elements implement preference relations. However, apart from deciding dominance among two fixed stable models, preference programs also allow for computing a model dominating a given one, or to show that none exists (not to mention the enumeration of the entire preference relation). This is of great practical relevance since it allows us to iteratively improve stable models until a non-dominated and hence optimal model is found (cf. Section 5).

4.1. Instance format

Preference statements are represented as a collection of facts. A weighted formula of form \( w_1, \ldots, w_l : \phi \) occurring in some set \( \Phi_r \) of a preference element \( e_j \) in a preference statement \( s \) as in (6) is represented as a fact of form

\[
\text{preference}(s,j,r,\text{for}(t_\phi),(w_1,\ldots,w_l)).
\]

where each \( w_i \) represents \( u_i \) for \( i = 1,\ldots,l \) and \( t_\phi \) is a term representing \( \phi \) by using functors neg, and, and or of arity 1, 2 and 2, respectively. For simplicity, we use indices \( r \) and \( j \) to identify the respective structural components.\(^8\) For representing the condition of \( e_j \) (cf. (5)), we set \( r \) to 0. A naming atom \( \text{name}(s) \) is represented analogously, except that \( \text{for}(t_\phi) \) is replaced by \( \text{name}(s) \).

\(^7\)For tangibility, we present the format in terms of ASP source as produced by \textit{asprin}.

\(^8\)In \textit{asprin}, a pair \((j,t)\) is used instead of \( j \), where \( t \) is a tuple of terms reflecting the respective instantiation of variables occurring in the original preference statement.
We let \( F_{s,j} \) denote the set of all facts obtained for all weighted formulas and naming atoms contained in preference element \( e_j \) belonging to preference statement \( s \). With this, we define the translation of a preference statement \( \#preference(s,t)\{e_1,\ldots,e_n\} \) as
\[
F_s = \{ \text{preference}(s,t) \} \cup \bigcup_{j=1}^{n} F_{s,j}
\]
where \( s \) and \( t \) represent \( s \) and \( t \), respectively.

For example, the previous preference statements are translated by \textit{asprin} as follows.

\( \#preference(1, \text{less(card)})\{a, \neg b, c\} \) yields
\[
\text{preference}(1, \text{less(cardinality)}).
\]
\[
\text{preference}(1,1,1,\text{for}(a),()).
\]
\[
\text{preference}(1,2,1,\text{for}(\neg(b)),()).
\]
\[
\text{preference}(1,3,1,\text{for}(c),()).
\]

\( \#preference(2, \text{more(weight)})\{1:a, 2:\neg b, 3:c\} \) yields
\[
\text{preference}(2, \text{more(weight)}).
\]
\[
\text{preference}(2,1,1,\text{for}(a),(1)).
\]
\[
\text{preference}(2,2,1,\text{for}(\neg(b)),(2)).
\]
\[
\text{preference}(2,3,1,\text{for}(c),(3)).
\]

\( \#preference(3, \text{subset})\{a, \neg b, c\} \) yields
\[
\text{preference}(3, \text{subset}).
\]
\[
\text{preference}(3,1,1,\text{for}(a),()).
\]
\[
\text{preference}(3,2,1,\text{for}(\neg(b)),()).
\]
\[
\text{preference}(3,3,1,\text{for}(c),()).
\]

\( \#preference(4, \text{pareto})\{\text{name}(1), \text{name}(2), \text{name}(3)\} \) yields
\[
\text{preference}(4, \text{pareto}).
\]
\[
\text{preference}(4,1,1,\text{name}(1),()).
\]
\[
\text{preference}(4,2,1,\text{name}(2),()).
\]
\[
\text{preference}(4,3,1,\text{name}(3),()).
\]

\( \#preference(5, \text{lexico})\{1: \text{name}(1), 2: \text{name}(2), 3: \text{name}(3)\} \) yields
preference(5,lexico).

preference(5,1,1,name(1),(1)).
preference(5,2,1,name(2),(2)).
preference(5,3,1,name(3),(3)).

4.2. Encoding

Although the ultimate purpose of preference programs is to decide whether one stable model is strictly preferred to another, their definition is independent of the notion of stable model. In fact, preference programs compare any pair of subsets \( X \) and \( Y \) of \( \mathcal{A} \). But for this to be possible, they have to be able to refer to the atoms of \( X \) and \( Y \). This is accomplished by reifying those atoms as the unique arguments of the unary predicates \( \text{holds} \) and \( \text{holds}' \). With this, preference programs can refer to the atoms of \( X \) and \( Y \) using atoms over those predicates. More formally, we define, for a set \( X \) of atoms, the following sets of facts:

\[
H(X) = \{ \text{holds}(a) \leftarrow \mid a \in X \} \quad \text{and} \quad H'(X) = \{ \text{holds}'(a) \leftarrow \mid a \in X \}.
\]

Then, the preference program implementing the preference relation induced by a preference statement is defined as follows.

**Definition 1.** Let \( \succeq_s \subseteq \mathcal{A} \times \mathcal{A} \) be a preference relation and let \( P_s \) be a logic program. Then, \( P_s \) is a preference program for \( \succeq_s \), if for all sets \( X, Y \subseteq \mathcal{A} \), we have

\[
X \succ_s Y \iff P_s \cup H(X) \cup H'(Y) \text{ is satisfiable.}
\]

Observe that we say that \( P_s \) is a preference program for \( \succeq_s \), but the condition only refers to its strict version \( \succ_s \). Note also that the definition abstracts from an underlying preference statement. In what follows, we often leave the preference relation \( \succeq_s \) implicit, and rather refer to the preference program \( P_s \) for the preference statement \( s \) inducing \( \succeq_s \). We also keep using \( P_s \) for denoting preference programs. Moreover, we point out that the definition and the following formal results apply as well if we consider that \( s \) is a preference specification. However, to simplify the presentation, from here on we focus on a single preference statement \( s \). Observe also that preference programs refer only to atom sets and are thus independent of any underlying programs. This changes (below) once sets \( H(X) \) and \( H'(Y) \) represent actual stable models of a logic program.

Our methodology foresees that a preference program for a primary preference statement with identifier \( s \) is composed of three parts:
1. a set $F_s$ of facts representing $s$,
2. a logic program $E_t$ accounting for the preference type $t$ of $s$,
3. a logic program $G$ comprising fixed auxiliary rules.

Whenever $s$ is the primary preference, the fact ‘\textit{optimize}(s) \leftarrow$’ is added to $F_s$ to represent the optimization directive \texttt{#optimize}($s$). We implement each preference type by an ASP encoding. Each such $E_t$ defines under which conditions a stable model is strictly better than another one with respect to preference type $t$ and facts $F_s$. In $E_t$ this is expressed using a unary predicate better that takes preference $s$ as argument. The implementation is required to yield better($s$) whenever the stable model captured by $H(X)$ is strictly better than that comprised in $H'(Y)$. The auxiliary rules in $G$ are common to all preference types. First, they extend the atomic truth assignment captured by $H(X)$ to all Boolean formulas occurring in the preference specification at hand. For any such formula $\phi$, the rules in $G$ warrant that holds($t_\phi$) is obtained when $\phi$ is entailed by the stable model $X$ captured in $H(X)$ where $t_\phi$ is the term representation of $\phi$. This is analogous for holds’ and $H'(X)$. To coordinate with $F_s$ and $E_t$, we stipulate that those rules only define atoms over the predicates holds and holds’. Second, $G$ contains the integrity constraint:

$$\leftarrow \neg \text{better}(P), \text{optimize}(P)$$

(9)

This constraint ensures that the preference program is unsatisfiable whenever no stable model is strictly better wrt the primary statement. Finally, the union $F_s \cup E_t \cup G$ constitutes a typical preference program in our framework.

In asprin a preference program is defined generically for the preference type, and consecutively instantiated to the specific preference in view of its preference elements. Let us illustrate this by the preference program for the primary preference statement \texttt{#preference}(3, subset)\{a, \neg b, c\} given in Listing 1.

The set $F_3$ representing preference statement 3 is given in lines 3-6, augmented by the fact in Line 1 accounting for its primariness. The encoding $E_{subset}$ of preference type \texttt{subset} consists of the single rule in lines 8 to 10. While the condition in Line 10 stipulates that each element captured by holds also belongs to the set represented by holds’, the literals in Line 9 make sure that there is at least one element that belongs to the extension of holds’ but not holds. Lines 12-15 comprise the auxiliary rules in $G$. The constraint in Line 12 is the same as (9). Lines 14 and 15 contain the satisfaction rules for negation.\textsuperscript{9} Altogether the three

\textsuperscript{9}The ones for conjunction and disjunction are omitted since they are irrelevant in the example.
optimize(3).

preference(3,subset).
preference(3,1,1,for(a),()).
preference(3,2,1,for(neg(b)),()).
preference(3,3,1,for(c),()).

better(P) :- preference(P,subset),
               not holds(X), holds'(X), preference(P,_,_,for(X),_),
               holds'(Y) : preference(P,_,_,for(Y),_), holds(Y).

:- not better(P), optimize(P).

holds(neg(A)) :- not holds(A), preference(_,_,_,for(neg(A)),_).
holds'(neg(A)) :- not holds'(A), preference(_,_,_,for(neg(A)),_).

Listing 1: Preference program for preference(3, subset){a, ¬b, c}.

parts constitute the preference program $P_3 = F_3 \cup E_{\text{subset}} \cup G$. With it, we may check whether $\{a, b\} \succeq_3 \{a\}$ holds. To this end, we reify both sets of atoms, and test whether $P_3 \cup H(\{a, b\}) \cup H'(\{a\})$ is satisfiable. This results in a stable model containing better(3) confirming that $\{a\} \subset \{a, ¬b\}$ holds.

The next proposition makes precise how preference programs capture the strict versions of the preference relations defined by preference statements.

**Proposition 1.** Let $s$ be a preference statement such that $\succeq_s \subseteq A \times A$ and $P_s$ be a preference program for $s$. Then, we have that $\succ_s$ is given by

$$\{(X, Y) \mid X, Y \subseteq A, P_s \cup H(X) \cup H(Y)' \text{ is satisfiable}\}.$$  

We can also capture those strict relations by single logic programs. For this, we consider all reified subsets of a set $X$ of atoms via the following choice rules:

$$C(X) = \{\{h\} \leftarrow \mid h \in H(X)\} \quad \text{and} \quad C'(X) = \{\{h'\} \leftarrow \mid h' \in H'(X)\}.$$  

In what follows, we stipulate that no atoms of the unary predicates holds and holds' occur in the heads of the rules of preference programs. With this, we can express our result as follows.

**Proposition 2.** Let $s$ be a preference statement such that $\succeq_s \subseteq A \times A$ and $P_s$ be a preference program for $s$. Then, we have that $\succ_s$ is given by

$$\{(X, Y) \mid Z \text{ is a stable model of } P_s \cup C(A) \cup C'(A), \quad X = \{a \mid \text{holds}(a) \in Z\} \text{ and } Y = \{a \mid \text{holds}'(a) \in Z\}\}.$$  

(10)
As an example, let us reconsider the above preference program \( P_3 \) implementing \#preference(3, subset)\( \{a, \neg b, c\} \). The logic program \( P_3 \cup C(\{a, b, c\}) \cup C'(\{a, b, c\}) \) results in 19 stable models capturing relation \( \succeq_3 \) over \( \{a, b, c\} \).

Next, we show how preference programs can be used for deciding whether a stable model of a logic program is preferred, and how a dominating model is obtained. We refer to the underlying programs as base programs and assume that they are formed over some set of ground atoms \( A \) that does not contain any atom over the unary predicates holds and holds'. We stipulate that base and preference programs are formed over disjoint sets of atoms. Interactions among those programs are controlled by mapping atoms in \( A \) to \( H(A) \) and \( H(A)' \), respectively.

To formulate our result, we need the following set of rules subject to a set \( X \) of atoms:

\[
R(X) = \{ \text{holds}(a) \leftarrow a \mid a \in X \}
\]

\( R(X) \) can be regarded as the dynamic counterpart of \( H(X) \) since reified atoms are now conditioned. With it, we obtain the following result.

**Proposition 3.** Let \( P \) be a logic program over \( A \), and \( P_s \) be a preference program for preference statement \( s \). Then, we have

1. If \( X \) is a stable model of \( P \), then \( X \) is \( \succeq_s \)-preferred iff \( (P \cup P_s \cup R(A) \cup H(X)') \) is unsatisfiable
2. If \( Y \) is a stable model of \( (P \cup P_s \cup R(A) \cup H(X)') \) for some \( X \subseteq A \), then \( Y \cap A \) is a stable model of \( P \) such that \( (Y \cap A) \succ_s X \).

We use \( (P \cup P_s \cup R(A) \cup H(X)') \) to check whether there is a model dominating \( X \). Note how the usage of program \( P \cup R_A \) restricts candidates to stable models of \( P \), unlike arbitrary subsets of \( A \) as in Proposition 2.

For illustration, consider a base program \( P \) only consisting of the cardinality constraint ‘\( \{a; b; c\} = 2.\)’ along with the above preference program \( P_3 \). Together with \( R(\{a, b, c\}) \) and \( H'(X) \) we may now check whether a set \( X \) is dominated by some stable model of \( P \). For instance, checking whether \( \{a\} \) is dominated is done with \( H'(\{a\}) \). This yields a stable model containing \( \text{holds}(a) \) and \( \text{holds}(b) \) and tells us that \( \{a\} \) is dominated by \( \{a, b\} \). Now, replacing \( H'(\{a\}) \) by \( H'(\{a, b\}) \) yields an unsatisfiable program, indicating that \( \{a, b\} \) is a \( \succeq_3 \)-preferred stable model of \( P \).

In what follows, we show how selected preference types are implemented in asprin. Each type is captured by a (non-ground) logic program.
less(card) is implemented by
\[
\text{better}(P) \leftarrow \text{preference}(P, \text{less(cardinality)}), \\
\sum\{ -1, X: \text{holds}(X), \text{preference}(P, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_); \\
\quad 1, X: \text{holds}'(X), \text{preference}(P, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_ ) \} > 0.
\]

more(weight) is implemented by\(^{10}\)
\[
\text{better}(P) \leftarrow \text{preference}(P, \text{more(weight)}), \\
\sum\{ W, X: \text{holds}(X), \text{preference}(P, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_
pareto is implemented by

\[
\text{better}(P) :- \text{preference}(P, \text{pareto}), \\
\text{better}(P''), \text{preference}(P, \_\_\_, \text{name}(P''), \_), \\
\text{betereq}(P') : \text{preference}(P, \_\_\_, \text{name}(P'), \_).
\]

lexico is implemented by

\[
\text{better}(P) :- \text{preference}(P, \text{lexico}); \\
\text{better}(P'), \text{preference}(P, \_\_\_, \text{name}(P'), (W)), \\
\text{equal}(P'') : \text{preference}(P, \_\_\_, \text{name}(P''), (V)), V > W.
\]

Observe that both types rely also on non-strict relations that need to be defined (cf. (7) and (8)). In analogy to preference programs for preference relations, our methodology foresees that they are implemented by rules defining unary predicates such as \text{betereq} or \text{equal}. For instance, a non-strict variant for \text{less(cardinality)} can be implemented by replacing \text{better} and ‘\text{>’} above by \text{betereq} and ‘\text{>=’}, respectively. Similarly, an equality-oriented variant is obtained by using \text{equal} and ‘\text{=’}.

The final configuration of a preference program for a composite preference depends upon the specific orderings referenced in the elements of the preference statement. For each ordering referred to in a naming atom, a corresponding program must be added. For instance, the preference program for preference statement 4 on page 11 must not only include the above program defining \text{better} for \text{pareto} but moreover programs defining \text{better} as well as \text{betereq} for \text{less(card)}, \text{more(weight)}, and \text{subset}.

5. Computing preferred models

In what follows, we consider two different approaches for computing one or all preferred stable models of a (ground) logic program. The first approach uses model-driven algorithms inspired by branch-and-bound optimization. The second one addresses both problems by translating a program with preferences into a disjunctive logic program, whose models correspond to the preferred models of the original program. The latter also extends seamlessly to query-answering, which is more involved with the former. After presenting these algorithms, we analyze the computational complexity of the problem landscape and relate it to the respective methods.

Our model-driven approach relies upon successive calls to a (multi-shot) ASP
solver. For a logic program $P$, we define

$$solve(P) = \begin{cases} X & \text{for some stable model } X \text{ of } P, \text{ if } P \text{ is satisfiable} \\ \bot & \text{if } P \text{ is unsatisfiable} \end{cases}$$

We also define the intersection $X \cap \bot$ for some set of atoms $X$ to be $\bot$.

The model-driven algorithms assume that their input logic program $P$ is finite. This implies that $P$ has a finite number of stable models, and guarantees that the algorithms always terminate (see also Section 8.1).

5.1. Computing one preferred model

Given a finite program $P$ and a preference program $P_s$ for a preference statement $s$, Algorithm 1\(^{11}\) computes a $\succeq_s$-preferred stable model of $P$. We put no restrictions on the program $P$ or on the preference program other than the finiteness of $P$; both may even be disjunctive programs.

\begin{algorithm}[h]
\caption{solveOpt($P$, $P_s$)}
\begin{algorithmic}[1]
\State \textbf{Input} : A finite program $P$ over $\mathcal{A}$ and a preference program $P_s$ for preference statement $s$.
\State \textbf{Output} : A $\succeq_s$-preferred stable model of $P$, if $P$ is satisfiable, and $\bot$ otherwise.
\State $Y \leftarrow solve(P)$
\State \If{$Y = \bot$} \textbf{return} $\bot$ \EndIf
\Repeat
\State $X \leftarrow Y$
\State $Y \leftarrow solve(P \cup P_s \cup R(\mathcal{A}) \cup H(X)') \cap \mathcal{A}$
\Until{$Y = \bot$}
\State \textbf{return} $X$
\end{algorithmic}
\end{algorithm}

The non-dominance test for candidate models is implemented as prescribed by Proposition 3. This is done in Line 5 where we check whether there is a $Y$ such that $Y \succ_s X$. That is, given a candidate $X$, we let a solver check whether $X$ is preferred wrt $\succeq_s$. If this succeeds, we obtain $\bot$ in Line 5, and return $X$ in Line 7. Otherwise, we obtain with $Y$ a counterexample dominating $X$, and we continue.

\(^{11}\)This algorithm is inspired by ideas in [4, 22], see the discussion for details.
the loop with \( Y \) as new candidate model. Note that Algorithm 1 can easily be
turned into an anytime algorithm returning the best stable model computed so far.

For capturing this in more detail, we define a trace of Algorithm 1 as the
sequence \((Y_0, Y_1, \ldots)\) of sets of atoms successively computed in Algorithm 1 as follows:

- \( Y_0 \) is the value of \( Y \) in Line 1 of Algorithm 1 and
- \( Y_i \) is the value of \( Y \) in Line 5 in the \( i \)th iteration of the loop for \( i > 0 \).

**Proposition 6.** Let \((Y_0, Y_1, \ldots)\) be a trace of Algorithm 1 for a finite program \( P \)
and a preference program \( P_s \) for preference statement \( s \). Then, the trace is finite,
has the form \((Y_0, \ldots, Y_n)\) for some integer \( n \geq 0 \), and

1. \( n > 0 \) if and only if \( P \) is satisfiable
2. \( Y_i \) is a stable model of \( P \) for \( 0 \leq i < n \)
3. \( Y_i \succ_s Y_{i-1} \) for \( 0 < i < n \)
4. \( Y_n = \bot \)
5. \( Y_{n-1} \) is a \( \succeq_s \)-preferred stable model of \( P \), if \( n > 0 \)
6. Algorithm 1 returns \( Y_0 \) if \( n = 0 \), and \( Y_{n-1} \) if \( n > 0 \)

**Theorem 1.** Given a finite program \( P \) and a preference program \( P_s \) for prefer-
ence statement \( s \), Algorithm 1 computes a \( \succeq_s \)-preferred stable model of \( P \) if \( P \) is
satisfiable, and \( \bot \) otherwise.

### 5.2. Computing all preferred models

Next, we address the problem of enumerating preferred models. While base
programs remain unrestricted, we first limit ourselves to preferences for which
we can decide whether \( X \succ Y \) holds for sets \( X, Y \) in polynomial time.\(^{12}\) In
view of this, we assume without loss of generality that preference programs are
stratified, since each problem decidable in polynomial time can be represented as
a stratified logic program (cf. [3]). Recall from Section 2 that a stratified logic
program is satisfiable if and only if the unique stable model of its normal rules
satisfies its integrity constraints.

Given a finite program \( P \) and a stratified preference program \( P_s \) for preference
statement \( s \), Algorithm 2 computes all \( \succeq_s \)-preferred stable models of \( P \). The idea

\(^{12}\)This restriction is lifted in Section 5.2.1.
Algorithm 2: solveOptAll\((P, P_s)\)

**Input**: A finite program \(P\) over \(A\) and a stratified preference program \(P_s\) for preference statement \(s\).

**Output**: The set of \(\succeq_s\)-preferred stable models of \(P\).

1. \(\mathcal{X} \leftarrow \emptyset\)
2. **loop**
3. \(Y \leftarrow \text{solve}\left(P \cup \bigcup_{X_i \in \mathcal{X}} (N_{X_i} \cup (\overline{P_s} \cup H(X_i))^i \cup R'(A)^i)\right) \cap A\)
4. **if** \(Y = \bot\) **then** return \(\mathcal{X}\)
5. **repeat**
6. \(X \leftarrow Y\)
7. \(Y \leftarrow \text{solve}\left(P \cup P_s \cup R(A) \cup H(X)^i \right) \cap A\)
8. **until** \(Y = \bot\)
9. \(\mathcal{X} \leftarrow \mathcal{X} \cup \{X_{|\mathcal{X}|+1}\}\)

is to collect preferred models computed in analogy to Algorithm 1. To see this, observe that Lines 3-8 correspond to Lines 1-6 in Algorithm 1. That is, starting from an initial model \(Y\) in Line 3, a preferred model \(X\) is obtained after the repeat loop via successive non-dominance tests. Preferred models are accumulated in the indexed set \(\mathcal{X}\) of form \(\{X_i \mid i \in I\}\) and use the indices in \(I\) to refer to different preferred models. The index set \(I\) grows with each addition to \(\mathcal{X}\) in Line 9, where we add \(X\) indexed with \(|\mathcal{X}|+1\) to \(\mathcal{X}\), viz. \(X_{|\mathcal{X}|+1}\).

The most intricate part of Algorithm 2 is Line 3. The goal is to compute a stable model of \(P\) that is neither dominated by nor equal to any preferred model in \(\mathcal{X}\). Line 3 checks whether there is a stable model \(Y\) of \(P\) such that \(X_i \neq Y\) and \(X_i \not\succ_s Y\) for all \(i \in I\). We already have \(Y \not\succ_s X_i\) since each \(X_i \in \mathcal{X}\) is \(\succeq_s\)-preferred.

For each \(i \in I\), Condition \(X_i \neq Y\) is guaranteed by the integrity constraint \(N_{X_i}\) of form

\[
N_{X_i} = \{\leftarrow X \cup \{-a \mid a \in A \setminus X\}\}.
\]  

(11)

Although such solution recording is exponential in space, it has the advantage of being non-intrusive to the solver.

For addressing condition \(X_i \not\succ_s Y\), preference programs are not directly applicable since they result in an unsatisfiability problem according to Definition 1. Instead, we need to encode the condition as a satisfiability problem in order to
obtain a stable model as a starting point for the subsequent search. Due to our restriction to stratified preference programs, this is accomplished as follows. Given a program \( P \), define \( \overline{P} \) as the program

\[
\left( P \setminus \{ r \in P \mid \text{head}(r) = \emptyset \} \right) \cup \\
\{ u \leftarrow \text{body}(r) \mid r \in P, \text{head}(r) = \emptyset \} \cup \{ \leftarrow \neg u \},
\]

where \( u \) is a new atom. Then, we observe the following property.

**Proposition 7.** If program \( P \) is stratified, \( \overline{P} \) is satisfiable iff \( P \) is unsatisfiable.

The next proposition shows how non-dominance is encoded as a satisfiability problem.

**Proposition 8.** Let \( P_s \) be a stratified preference program for preference statement \( s \). Then, for all sets \( X, Y \subseteq A \), we have

\[ X \not\succ_s Y \iff P_s \cup H(X) \cup H(Y)' \text{ is satisfiable.} \]

The next result captures the essence of the non-dominance test in Line 3 of Algorithm 2.

**Proposition 9.** Let \( P \) be a program over \( A \) and \( P_s \) be a stratified preference program for preference statement \( s \). If \( Y \) is a stable model of

\[ (P \cup P_s \cup H(X) \cup R'(A)) \]

for some \( X \subseteq A \), then \( Y \cap A \) is a stable model of \( P \) such that \( X \not\succ_s (Y \cap A) \).

Note that we also have \((Y \cap A) \not\succ_X X\) whenever \( X \) is preferred wrt \( \succeq_s \).

Given that \( X \) contains several preferred models, we need to test each model in \( X \) for the condition in Proposition 9. To do so, we let \( P^i \) denote the program obtained from \( P \) by replacing each atom \( a \) occurring in \( P \) by \( a^i \). Moreover, we generalize the definition of \( R(X) \) to \( R^i(X) = \{ \text{holds}^i(a)^i \leftarrow a \mid a \in X \} \). With this, the next proposition captures the functioning of Line 3 of Algorithm 2.

**Proposition 10.** Let \( \{ X_i \mid i \in I \} \) be the value of \( X \) in Line 2 of Algorithm 2 and let \( Y \) be the value returned in Line 3 of Algorithm 2. Then,

1. either \( Y \) is a stable model of \( P \) such that \( Y \neq X_i \), and \( X_i \not\succ_s Y \) for all \( i \in I \),
2. or no such stable model exists, and \( Y \) is \( \bot \).
In analogy to Section 5.1, we detail the execution of Algorithm 2 by means of traces. A trace of Algorithm 2 is a (possibly empty) sequence \((X_i)_{i \in I}\) of sets \(X_i\) added to \(X\) in Line 9 at different iterations of the outermost loop.

**Proposition 11.** Let \((X_i)_{i \in I}\) be a trace of Algorithm 2 for a finite program \(P\) and a stratified preference program \(P_s\) for preference statement \(s\). Then, the trace is finite, and

1. for each \(i \in I\), \(X_i\) is a \(\succeq_s\)-preferred stable model of \(P\)
2. for each \(\succeq_s\)-preferred stable model \(Y\) of \(P\), there is a unique \(i \in I\) such that \(Y = X_i\)
3. \(X_i \neq \bot\) for all \(i \in I\)
4. there is no \(\succeq_s\)-preferred stable model of \(P\), if \(X = \emptyset\) (or \(I = \emptyset\))

For each iteration \(i\) of the outermost loop of Algorithm 2, starting at \(i=1\), we define an \(i\)-trace as the sequence \((Y_{i0}, Y_{i1}, \ldots)\) of sets of atoms successively computed as follows:

- \(Y_{i0}\) is the value of \(Y\) in Line 3 and
- \(Y_{ij}\) is the value of \(Y\) in Line 7 at the \(j\)th iteration of the repeat loop for \(j > 0\).

Note that there is one \(i\)-trace for every \(i \in I\), and there is one additional \(i\)-trace for the last iteration of the algorithm. To state the properties of \(i\)-traces, we let \(X_i\) denote the set \(\{X_j \mid j \in I, j \leq i\}\) for each iteration \(i\).

**Proposition 12.** Every \(i\)-trace of the algorithm is finite, has the form \((Y_{i0}, \ldots, Y_{in})\) for some integer \(n \geq 0\), and

1. \(n > 0\) if \(X_{i-1}\) does not contain all \(\succeq_s\)-preferred stable models of \(P\)
2. \(Y_{ij}\) is a stable model of \(P\) for \(0 \leq j < n\)
3. \(Y_{ij} \succ_s Y_{ij-1}\) for \(0 < j < n\)
4. \(Y_{in} = \bot\)
5. \(Y_{in-1}\) is a \(\succeq_s\)-preferred stable model of \(P\), if \(n > 0\)
6. \(X_i\) is undefined if \(n = 0\), and is \(Y_{in-1}\) if \(n > 0\)
7. \(Y_{ij}\) is such that \(X_k \neq Y_{ij}\) and \(X_k \not\succ_s Y_{ij}\) for \(0 \leq j < n\) and all \(X_k \in X_{i-1}\)

Note the similarity with Proposition 6. Basically, every \(i\)-trace amounts to a trace of Algorithm 1 where the stable models satisfy the property of item 7, and the preferred stable models computed are not directly returned, but they are added to the set \(X\) as \(X_i\)’s (see item 6).

Based on this, we can prove soundness and completeness.
Theorem 2. Given a finite program \( P \) and a stratified preference program \( P_s \) for preference statement \( s \), Algorithm 2 computes exactly the set of \( \succeq_s \)-preferred stable models of \( P \).

5.2.1. Computing all preferred models for complex preferences

We now drop the restriction of polynomially decidable preference relations, and consider preferences decidable in \( NP \). Without loss of generality, we thus allow for preference programs being normal because each problem decidable in \( NP \) can be represented as a normal logic program.

As above, the crucial point is to express the non-dominance test in Line 3 of Algorithm 2 as a satisfiability problem. For addressing this in the case of normal programs, Eiter and Gottlob invented in [23] the saturation technique. The idea is to re-express the problem as a positive disjunctive logic program, containing a special-purpose atom, say \( bot \). Whenever \( bot \) is obtained, saturation derives all atoms (belonging to a “guessed” model). Intuitively, this is a way to materialize unsatisfiability. For automatizing this process, we build upon the meta-interpretation-based approach described in [24, 25]. The idea is to map a program \( P \) onto a set \( \mathcal{R}(P) \) of facts via reification.\(^{13}\) The facts in \( \mathcal{R}(P) \) are then combined with a meta-encoding \( \mathcal{M} \) implementing saturation.\(^{14}\) Then, \( P \) has a stable model iff \( \mathcal{R}(P) \cup \mathcal{M} \) has a corresponding one excluding \( bot \), and \( P \) is unsatisfiable iff \( \mathcal{R}(P) \cup \mathcal{M} \) has a unique saturated stable model containing \( bot \). Note that \( \mathcal{R}(P) \cup \mathcal{M} \) is always satisfiable since it is a positive program.

In our case, we consider for a preference statement \( s \) the positive disjunctive logic program

\[
\mathcal{R}\left(P_s \cup C(A) \cup C'(A)\right) \cup \mathcal{M}.
\]  

(13)

In analogy to Proposition 2, this reified program has a stable model (excluding \( bot \)) for each pair \( X, Y \subseteq A \) satisfying \( X \succeq_s Y \), and it has a saturated stable model (including \( bot \)) if there is no such pair. Note that \( X \) and \( Y \) are merely subsets of \( A \), not necessarily stable models.

Given a reified program \( Q_s \) as in (13), it is sufficient to replace the program passed to \texttt{solve} in Line 3 of Algorithm 2 by the following disjunctive program

\[
\left( P \cup \bigcup_{X \subseteq A} N_X \right) \cup Q_s \cup \mathcal{N}(A') \cup \mathcal{R}'(A) \cup \{ \leftarrow \neg \text{bot} \}
\]  

(14)

\(^{13}\)For instance, programs can be reified with \texttt{clingo} via option \texttt{--output=reify}.

\(^{14}\)Specific meta-encodings for saturation come with the respective distributions of \texttt{clingo}, and are bundled at https://github.com/potassco/clingo/tree/master/examples/reify.
As in Algorithm 2, \((P \cup \bigcup_{X \in \mathcal{X}} N_X)\) generates stable model candidates different from those in \(\mathcal{X}\). Programs \(N(\mathcal{X})\) and \(\mathcal{R}'(\mathcal{A})\) restrict the choices of \(C(\mathcal{A})\) and \(C'(\mathcal{A})\) in (13), respectively.

\[
N(\mathcal{X}) = \{ \text{bot} \leftarrow \bigcup_{X_i \in \mathcal{X}} \{ u_i \} \} \cup \bigcup_{X_i \in \mathcal{X}} \{ u_i \leftarrow R^-(\text{holds}(a)) \mid a \in X_i \} \\
\cup \bigcup_{X_i \in \mathcal{X}} \{ u_i \leftarrow R^+(\text{holds}(a)) \mid a \in \mathcal{A} \setminus X_i \}
\]

\[
\mathcal{R}'(\mathcal{A}) = \{ R^+(\text{holds}'(a)) \leftarrow a \mid a \in \mathcal{A} \} \cup \{ R^-(\text{holds}'(a)) \leftarrow \neg a \mid a \in \mathcal{A} \}
\]

Just like \(\text{bot}\), \(R^+(a)\) and \(R^-(a)\) belong to the signature of the used saturation encoding \(\mathcal{M}\); they encode reified positive and negative atoms, respectively. Unlike this, \(u_i\) is a new atom for each \(X_i \in \mathcal{X}\) (similar to atom \(u\) in (12)). While \(N(\mathcal{X})\) eliminates all non-preferred models (outside of \(\mathcal{X}\)) from the candidate sets generated by \(C(\mathcal{A})\) via saturation, program \(\mathcal{R}'(\mathcal{A})\) maps all candidate models generated by \((P \cup \bigcup_{X \in \mathcal{X}} N_X)\) to the signature of \(\mathcal{M}\), more specifically to \(R(H(\mathcal{A}'))\).

Interestingly, all above properties of Algorithm 2 carry over to the more complex case. To see this, observe that reified preference programs decide preference relations just as regular ones. However, saturation leads to a subproblem of elevated complexity. If the original preference program is normal, and its decision problem is thus \(NP\) complete, the resulting saturated program leads to an \(NP^{NP}\) problem. Hence, Algorithm 2 cannot be used to enumerate preferred models for preferences whose decision problem is beyond \(NP\), since then saturation is inapplicable. A general view of this limitation is presented in Section 5.4.

5.3. Computing preferred models via a translation to disjunctive logic programs

The axiomatic way of computing preferred models translates a normal program with preferences into a disjunctive logic program without preferences, so that the stable models of the translated program correspond to the preferred stable models of the original program. This reduces the problem of computing optimal stable models of normal programs with preferences to the problem of computing stable models of disjunctive logic programs. Then, in practice, we can use a disjunctive ASP solver to compute stable models of the translated program, and in virtue of the translation, those stable models are also preferred models of the original program. In ASP, this kind of translation is commonly formulated using

\[\text{\textsuperscript{15}}\text{The definition of } N_X \text{ is given in (11).}\]

\[\text{\textsuperscript{16}}\text{For instance, in the meta encoding in [24] literals } a \text{ and } \neg a \text{ are represented as } \text{true}(\text{atom}(a)) \text{ and } \text{fail}(\text{atom}(a)), \text{ respectively.}\]
Saturation techniques (cf. [26, 24]) because one has to combine the generation of model candidates with the failure to generate dominating counter-models.

This approach is easily accomplished by means of the above building blocks. In analogy to the basic approach in Section 4.2, we consider for a base program $P$ and a preference program $P_s$ for a preference statement $s$ the positive disjunctive program

$$R(P_s \cup (P \cup R(A)) \cup C'(A)) \cup M.$$  

(15)

However, instead of generating arbitrary sets of atoms via $C'(A)$ as in (13), we only generate stable models of $P$. Hence, the program in (15) has a stable model (excluding $\text{bot}$) for each pair $X,Y \subseteq A$ such that $X$ is a stable model of $P$ and $X \succ_s Y$, and it has a saturated stable model (including $\text{bot}$) if there is no such pair.

This leads us to the following disjunctive logic program, $E_{P,s}$, extending $P$.

$$P \cup (R(P_s \cup (P \cup R(A)) \cup C'(A)) \cup M) \cup R'(A) \cup \{\leftarrow \neg \text{bot}\}$$

(16)

The stable models $Y$ of $P$ are mapped to the signature of $M$ by $R'(A)$, while the rest of the rules check that there is no stable model $X$ of $P$ such that $X \succ_s Y$.

For computing one or all $\succeq_s$-preferred models of $P$, respectively, it is sufficient to pass $E_{P,s}$ to a disjunctive ASP solver along with the appropriate option. Similarly, checking whether a query $a$ is true in some $\succeq_s$-preferred model of $P$ can be done by passing the program $E_{P,s} \cup \{\leftarrow \neg a\}$ to the solver. To do the same with Algorithm 2, one had to enumerate preferred models until one comprising $a$ is found. See [27] for other alternatives to this.

The major difference between the algorithmic and axiomatic approach is that the former relies on a sequence of similar problems successively passed to a solver, while the latter encapsulates all into solving a single problem specification. Hence, in ASP, the axiomatic approach is restricted to problems at the second level of the polynomial hierarchy, while the model-driven approach can go beyond this because only each solver invocation is restricted to such problems. Also, as we will see in the complexity analysis, the complete problem captured by $E_{P,s}$ is often harder than the successive problems considered in Algorithm 1 and 2. This is reflected in the type of programs used by each approach. For instance, when considering a normal base program along with a stratified preference program, Algorithm 1 considers a suite of normal programs, while $E_{P,s}$ is a disjunctive program. The other side of this is that the axiomatic approach requires a single call
to a disjunctive solver, while Algorithm 1 may require an exponential number of calls to the underlying solver.\footnote{In our experiments of Section 9 we have not observed that exponential number of solving calls, but certainly in some cases the number of calls is large. As we see in that section, this issue can be alleviated using domain specific heuristics.}

On the more pragmatic side, note that today’s disjunctive ASP solvers rely on the interaction of two internal solvers, so that one external call is usually dealt with by several internal interactions. Also, the successive programs in Algorithm 1 and 2 are rather similar, so that multi-shot solving can be used for reducing redundancies among successive solver invocations. A major drawback of Algorithm 2 is that it faces an exponential space consumption in the worst case. Again, this is also a common problem when enumerating models in today’s solvers, although here polynomial space enumeration can be used as well.\footnote{For instance, clasp offers both types of enumeration because “solution recording” is sometimes more effective [28].}

However, all these aspects can only be evaluated empirically, as done in Section 9.

5.4. Computational complexity

As mentioned in the introductory section, we face the manifold combinations of optimization problems, base programs, and preference programs, which results in a whole spectrum of distinct computational complexities.

We start by introducing shortly the complexity classes that show up in our analysis. See [29] for an introduction to complexity theory. By $\mathcal{P}^C$ ($\mathcal{NP}^C$) we denote the class of decision problems that are solvable in polynomial time by a deterministic (nondeterministic, respectively) Turing machine that has access to an oracle for any problem in the class $C$. Then, the classes $\Sigma^p_k$, $\Pi^p_k$ and $\Delta^p_k$ are defined as follows:

$$\Delta^p_0 = \Sigma^p_0 = \Pi^p_0 = \mathcal{P}$$ and for all $k \geq 1$, $\Delta^p_k = \mathcal{P}^{\Sigma^p_{k-1}}$, $\Sigma^p_k = \mathcal{NP}^{\Sigma^p_{k-1}}$, $\Pi^p_k = \mathcal{co-}\Sigma^p_{k}$.

In particular, $\mathcal{NP} = \Sigma^p_1$ and $\mathcal{co-}\mathcal{NP} = \Pi^p_1$. We also consider the classes $F\Sigma^p_k$, $F\Pi^p_k$ and $F\Delta^p_k$ of function problems that correspond to the previous decision problems, where the task is not to decide whether a solution exists, but to actually compute such a solution.

We continue by enumerating the problems that we consider. Their input always contains a base program $P$ over a set of ground atoms $\mathcal{A}$, and a preference program $P_s$ for some preference statement $s$. We assume all those elements to
be finite. Also, note that neither the preference statement $s$ nor the preference relation $\succeq_s$ are part of the input.

- **Model Finding**: Find some $\succeq_s$-preferred stable model of $P$.

- **Query**: Given some atom $a \in A$, decide whether there exists some $\succeq_s$-preferred stable model $X$ of $P$ such that $a \in X$.

- **Optimality**: Given a stable model $X$ of $P$, decide whether $X$ is a $\succeq_s$-preferred stable model or $P$.

- **Non Dominance**: Given a set $\mathcal{X}$ of $\succeq_s$-preferred stable models of $P$, decide whether there exists some stable model $Y$ of $P$ such that $X \neq Y$ and $X \not\succeq Y$ for all $X \in \mathcal{X}$.

Model Finding is a function problem, while the others are decision problems. The former is solved by Algorithm 1, and by the axiomatic approach presented in Section 5.2.1. We are also interested in the Query problem, that can either be solved by the axiomatic approach, or by some extension of Algorithm 2 (see Section 5.2.1). We have also presented methods for computing all preferred models, but we do not study the complexity of that task. The problems Optimality and Non Dominance are solved at different stages of Algorithms 1 and 2. Namely, Line 5 of Algorithm 1 and Line 7 of Algorithm 2 solve the complement of the Optimality problem; while Line 3 of Algorithm 2 solves the Non Dominance problem, using program (14) if the preference program is normal.

We analyze the complexity of these problems for different types of base and preference programs. More specifically, we consider the combinations where the base program is either normal or disjunctive; and the preference program is either stratified, normal or disjunctive. We do not delve into the case where the base program is stratified because in this case it is clear that all problems become polynomial. The preference programs shown in this paper are all stratified. In [30], we have used a normal preference program to represent CP-nets [31]. And so far, we have never used a disjunctive preference program. Recall that the problem of deciding if a program $P$ has some stable model is $\Sigma_0^p$-complete if $P$ is stratified, is $\Sigma_1^p$-complete if $P$ is normal, and is $\Sigma_2^p$-complete if $P$ is disjunctive (cf. [3]).\footnote{The complexity results from [3], unlike ours, consider normal and disjunctive logic programs \textit{without} choice rules. However, it is well known how to translate choice rules to normal rules, using additional atoms, in polynomial time. Therefore, we can apply these results to our setting.}

29
Accordingly, we say that stratified, normal and disjunctive logic programs are of class 0, 1 and 2, respectively.

**Theorem 3.** The following complexity results are parametrized by the class $i \in \{1, 2\}$ of the base program and the class $j \in \{0, 1, 2\}$ of the preference program:

- The problem Model Finding is $F\Delta_{i+1}^p$-hard and belongs to $F\Sigma_{\max\{i,j\}+1}^p$.
- The problem Query is $\Sigma_{\max\{i,j\}+1}^p$-complete.
- The problem Optimality is $\Pi_{\max\{i,j\}}^p$-complete.
- The problem Non Dominance is $\Sigma_{\max\{i,j+1\}}^p$-complete.

The full proofs of these results can be found in Appendix A.4. Here, we only describe their basic structure.

The membership proofs for Model Finding, Query, and Non Dominance are similar. The first one introduces a nondeterministic Turing machine that guesses possible stable models of the base program, checks their stability with a $\Sigma_{i-1}^p$ oracle, and their optimality with a $\Sigma_{\max\{i,j\}}^p$ oracle. For the Query problem, guesses are restricted to those that contain the query atom, while for Non Dominance, the last optimality check is replaced by a check of the condition on the sets in $X$ using a $\Sigma_j^p$ oracle. The membership proof for Optimality is based on Proposition 3, that shows how to solve the complement of the problem using programs of class $\max\{i, j\}$.

The hardness proof for the Model Finding problem is done by a reduction from the problem of computing an optimal stable model of a logic program with optimization statements. The hardness proof for the Query problem is done by reductions from problems of abduction in logic programming [32]. The hardness proof for the Optimality problem is done by reductions from the problem of deciding the satisfiability of a logic program. Finally, the hardness proof for the Non Dominance problem is done by reductions from problems of abduction and of satisfiability of logic programs.

Model Finding is the only problem for which we do not provide a completeness result. It is an open question to us what complexity class could be complete for this problem. To simplify matters, let us focus on the case where $i = 1$ and $j = 0$. In this situation, a reasonable option would be the class $F\Delta_2^p$. In fact, for all preference types that we have studied that are decidable in polynomial time, the Model Finding problem is in $F\Delta_2^p$ (when $i = 1$). For example, for $\text{more}(\text{weight})$, etc.
that falls into this case, it is possible to perform a binary search over the possible
total weights of the solutions, bounding the number of calls to an \( \text{NP} \) oracle by
a polynomial (cf. [2]). But in asprin in general there is no apparent structure that
can be exploited to perform that kind of search, given that the preference relation
is simply represented by a logic program. Actually, Algorithm 1 may need an ex-
ponential number of solving calls, and it is hard for us to imagine how to improve
that bound in this general setting. The other obvious option is the class \( F\Sigma_2^p \). But
if the problem was hard for that class, then the complexity would be like the one
for the Query problem, which is certainly possible, but unexpected to us. In that
case, one would be able to compute stable models of disjunctive logic programs
by computing preferred models of normal logic programs, and it is also hard for
us to imagine how that could be possible.

We can now discuss our implementations in view of these complexity results.

For Model Finding, the hardness result suggests that the problem cannot be
solved by a single solving call to a solver for the base program. This justifies
the usage of either an iterative method as in Algorithm 1, or a disjunctive logic
program like (16) when both the base and the preference programs are not dis-
junctive.

For the Query problem, the complexity result tells us that, if neither the base
nor the preference program are disjunctive, then the problem is in \( \Sigma_2^p \) and can be
solved by a disjunctive logic program, just like we do in Section 5.2.1, while this
is in principle not possible if any of those programs is disjunctive. As mentioned
in Section 5.2.1, in this situation we could still solve the problem iterating over
all preferred solutions with Algorithm 2, but this could require an exponential
amount of space.

For the Optimality problem, as expected, the complexity is aligned with the
type of logic programs used in Lines 5 of Algorithm 1 and Line 7 of Algorithm 2.
For stratified and normal preference programs, membership in \( \Pi_1^p \) suggests that
the subproblems tackled by those lines are easier than the overall optimization
problem, tackled by the axiomatic approach at once, which is \( F\Delta_{i+1}^p \)-hard. We
conjecture that this also happens when preference programs are disjunctive, but
our current results do not allow us to support that claim.

For the Non Dominance problem, the complexity result justifies Line 3 of
Algorithm 2 and the alternative version using program (14). If the preference pro-
gram is stratified, then the problem is in \( \Sigma_2^p \) and can be represented by a program
of the same type as the base program, as in Line 3 of Algorithm 2. If the prefer-
ence program is normal, then the problem is \( \Sigma_2^p \)-hard and demands the additional
expressive power of disjunctive logic programs, as in (14). In the remaining case,
if the preference program is disjunctive, then the problem is $\Sigma^p_3$-hard and therefore out of the scope of our algorithms, under the usual assumptions.

6. *asprin*'s input language

We present the first-order modeling language of *asprin*, that we use below to express existing approaches to preferences.

A *logic program with preferences* in *asprin* consists of a logic program in the language of the ASP system *clingo* [18] together with a preference specification. The *preferred stable models* of such a logic program with preferences are the stable models of the logic program that are preferred wrt the preference relation defined by the ground instantiation of the preference specification.

We present preference specifications and their ground instantiations following the steps of Section 3.1. The expressions that we introduce here can be simplified like we did in that section. Further details can be found in [33].

A weighted formula is of the form ‘$t : : \varepsilon$’ where $t$ is a term tuple of the form $t_1, \ldots, t_n$ for some terms $t_1, \ldots, t_n$ and $n \geq 0$, and $\varepsilon$ is either a Boolean formula or a naming atom. Both $t$ and $\varepsilon$ may contain variables. Boolean formulas are formed from atoms using connectives $\text{not}$, $\&$, and $|$. If they have the form $\phi_1 \& \ldots \& \phi_n$, then they can also be written using the body notation $\phi_1, \ldots, \phi_n$. Naming atoms of form ‘$**s$’ refer to the preference associated with term $s$. We use ‘$**s$’ rather than name(s) to free the usage of the latter predicate. If $t_1 :: \varepsilon_1, \ldots, t_n :: \varepsilon_n$ are weighted formulas, then ‘$\{ t_1 :: \varepsilon_1; \ldots; t_n :: \varepsilon_n \}$’ is a set of weighted formulas.

A preference element is of the form ‘$\Phi_1 >> \ldots >> \Phi_n \mid | \phi : B$’ where each $\Phi_i$ is a set of weighted formulas, $\phi$ is a weighted formula, and $B$ is a rule body. Preference elements are required to be safe, that is, all their variables must occur either in some positive body literal or in the body of the encompassing preference statement.

A preference statement is of the form

#preference(s, t) {e_1; \ldots; e_n} : B.

where $s$ and $t$ are terms denoting the preference name and its type, and each $e_j$ is a preference element. The body $B$ of a preference statement is used to instantiate the variables of $s, t$ and each $e_i$. For safety, all variables appearing in $s, t$ and $B$ must appear in a positive literal in $B$.

An optimization directive is of the form ‘$\#\text{optimize}(s) : B,$’ where $s$ is a term and $B$ a body. Like before, all variables appearing in $s$ and $B$ must
appear in a positive literal in \( B \).

A preference specification consists of at least one preference statement, and at least one optimization directive. Its ground instantiation, that we specify below, must comply with the conditions stated in Section 3.1: the set of its ground preference statements must be closed and acyclic, it must contain a unique ground optimization directive, and the argument of that directive must be the name of some ground preference statement.

The ground instantiation of a preference specification is determined by the logic program encompassing the specification, together with the rule bodies \( B \) that may occur after the colon ‘:’ in preference elements, preference statements and optimization directives. Those bodies must consist only of atoms over domain and built-in predicates, cf. [33]. This is important, because the truth value of those atoms is unique for all stable models of the encompassing logic program. Accordingly, we can let \( D \) be the set of those atoms that is true. Then, to define the ground instantiation of a preference specification, first we take all the ground instances of the preference elements, preference statements and optimization directives whose ground bodies are satisfied by \( D \), and then we just remove the ground bodies from them. For example, given this logic program with preferences:

1. \( \text{dom}(1..2) \).
2. \{ a(X,Y) : \text{dom}(X), \text{dom}(Y) \}.
3. \#preference(p(X),subset){ a(X,Y) : \text{dom}(Y) } : \text{dom}(X).
4. \#optimize(p(X)) : \text{dom}(X), \text{not} \ \text{dom}(X+1).

the set \( D \) is \{ \text{dom}(1), \text{dom}(2) \}, and the ground instantiation of the preference specification is as follows:

\#preference(p(1),subset)\{ a(1,1) ; a(1,2) \}.
\#preference(p(2),subset)\{ a(2,1) ; a(2,2) \}.
\#optimize(p(2)).

In the implementation, the restriction about the type of atoms occurring in rule bodies \( B \), together with the safety conditions mentioned above, guarantee that a preference specification is translated into a set of rules that is grounded into facts, that are the same for all stable models.

We conclude this section with a variant of our introductory preference specification about leisure activities, without base program, where all predicates except \( \text{do} \) and \( \text{hot} \) are domain predicates:
The preference relations costs, fun, and temps are combined in Line 12 via pareto and the resulting relation is declared to be subject to optimization in Line 14. The syntax mixes that of regular ASP with preference-oriented constructs. The latter are characterized by doublings such as ::, >>, | |, and **. More examples are given in the next section. Also, preference types less(weight) and aso are described in Sections 7.1 and 7.2, respectively.

7. Integrating approaches to preferences

To further illustrate the generality of our approach as well as the usage of actual preference programs in asprin, we show below how well-known approaches to preferences can be implemented.

7.1. Optimization statements

First of all, let us see how common optimization statements are expressed in asprin. A #minimize directive is of the form

\[ \#\text{minimize}\{w_1@k_1, t_1, \ldots, w_n@k_n, t_n : \tilde{\ell}_1, \ldots, \tilde{\ell}_n\} \]

where each \(w_i\) and \(k_i\) is an integer, and \(t_i = t_{i1}, \ldots, t_{im}\) and \(\tilde{\ell}_i = \ell_{i1}, \ldots, \ell_{ik}\) are tuples of terms and literals, respectively. For a set \(X\) of atoms and an integer \(k\), let \(\Sigma^X_k\) denote the sum of weights \(w_i\) over all occurrences of elements \((w_i@k_i, t_i : \tilde{\ell}_i)\) in \(M\) such that \(X \models \tilde{\ell}_i\). Then, for sets \(X, Y\) of atoms and minimize statement \(M\)

\[34\]
as above, we have that $X \succ_M Y$ if there is some integer $k$ such that $\Sigma^X_k < \Sigma^Y_k$ and $\Sigma^X_{k'} = \Sigma^Y_{k'}$ for all $k' > k$; and $X \succeq_M Y$ if either $X \succ_M Y$ or $\Sigma^X_k = \Sigma^Y_k$ for all integers $k$.

In asprin, a minimize statement $M$ as above can be represented by the following preference specification.

```plaintext
#preference(s_M, lexico) { k :: **s_k | (w@k, t: l) ∈ M }.
#preference(s_k, less(weight)) { w, (t) :: l | (w@k, t: l) ∈ M }.
#optimize(s_M).
```

The preference type `less(weight)` is implemented as follows.

```plaintext
better(P) :- preference(P, less(weight)),
    #sum { W,T,F : holds(F), preference(P,_,_,for(F),(W,T)) ;
           W,T,F : holds'(F), preference(P,_,_,for(F),(W,T)) } > 0.
```

Note that by wrapping tuples $t$ into $(t)$, we only deal with pairs $w, (t)$ rather than tuples of varying length. For the aggregation by `lexico`, the implementation includes also a rule defining an equally-oriented variant, replacing `better` and `$>$` by `equal` and `$=$`, respectively.

asprin’s separation of preference declarations from optimization directives not only illustrates how standard optimization statements conflate both concepts but it also explicates the interaction of preference types `lexico` and `less(weight)`.

To see the correctness of the approach, consider a minimize statement $M$, and let $E_{lexico}$ be the preference program for `lexico` from Section 4.2, $E_{less(weight)}$ be the previous rules for `less(weight)`, $F_M$ be the set of facts for the corresponding preference specification, and $G$ be the auxiliary rules from Section 4.2. Then, it can be shown that $E_{lexico} \cup E_{less(weight)} \cup F_M \cup G$ is a preference program for $\succeq_M$.

### 7.2. Answer set optimization

For capturing answer set optimization (ASO; [4]), we consider ASO rules of form

$$\phi_1 > \cdots > \phi_m \leftarrow B$$

(17)

where each $\phi_i$ is a propositional formula for $1 \leq i \leq m$ and $B$ is a rule body.

The semantics of ASO is based on satisfaction degrees for rules as in (17). The satisfaction degree of such a rule $r$ in a set of atoms $X$, written $v_X(r)$, is 1 if $X \models b$ for some $b \in B$, or if $X \models b$ for some $\neg b \in B$, or if $X \not\models \phi_i$ for every $1 \leq i \leq m$, and it is $\min\{k \mid X \models \phi_k, 1 \leq k \leq m\}$ otherwise. Then, for sets $X, Y$ of atoms and a set $O$ of rules of form (17), $X \succeq_O Y$ if for all rules $r \in O$, $X \not\models_r Y$.
\(v_X(r) \leq v_Y(r)\), and it follows that \(X \succ_O Y\) if \(X \succeq_O Y\) and there is some rule \(r \in O\) such that \(v_X(r) < v_Y(r)\).

In \textit{asprin}, we can represent an ASO rule \(r\) as in (17) as a preference statement of the form

\[
\text{#preference}(s_r, \text{aso})\{ \phi_1 \gg ... \gg \phi_m \mid B \}.
\]

A set \(\{r_1, \ldots, r_n\}\) of ASO rules is represented by corresponding preference statements \(s_{r_1} \) to \(s_{r_n}\) along with an aggregating \textit{pareto} preference subject to optimization.

\[
\text{#preference}(\text{paraso}, \text{pareto})\{ **s_{r_1}, \ldots, **s_{r_n} \}.
\]

\textbf{#optimize}(\text{paraso}).

Note that other composite preferences instead of \textit{pareto} could be used just as well.

The core implementation of preference type \textit{aso} is given in Lines 1-23 below. Predicate \textit{one} is true whenever an ASO rule has satisfaction degree 1 wrt the stable model captured by \(H(X)\). The same applies to \textit{one’} but wrt \(H'(X)\).

1. \textit{one}(P) :- preference(P, aso), not holds(F) : preference(P, _, R, for(F), _), R > 1.
2. \textit{one}(P) :- preference(P, aso), holds(F), preference(P, _, 1, for(F), _).
3. \textit{one}(P) :- preference(P, aso), not holds(F), preference(P, _, 0, for(F), _).
4. \textit{one’}(P) :- preference(P, aso), not holds’(F) : preference(P, _, R, for(F), _), R > 1.
5. \textit{one’}(P) :- preference(P, aso), holds’(F), preference(P, _, 1, for(F), _).
6. \textit{one’}(P) :- preference(P, aso), not holds’(F), preference(P, _, 0, for(F), _).

With these rules, we derive \textit{better}(s_r) in Line 15 whenever some ASO rule \(r\) has satisfaction degree 1 in \(X\) and one greater than 1 in \(Y\). Otherwise, \textit{better}(s_r) is derivable in Line 16 whenever \(r\) has satisfaction degree \(R\) in \(X\) but none of the formulas \(\phi_1\) to \(\phi_R\) are true in \(Y\). This is analogous for \textit{bettereq} in lines 20-23.

15. \textit{better}(P) :- preference(P, aso), \textit{one}(P), not \textit{one’}(P).
16. \textit{better}(P) :- preference(P, aso), preference(P, _, R, for(F), _), holds(F), R > 1, not \textit{one’}(P), not holds’(G) : preference(P, _, R’, for(G), _), 1 < R’, R’ <= R.
17. \textit{bettereq}(P) :- preference(P, aso), \textit{one}(P).
bettereq(P) :- preference(P,aso),
    preference(P,_,R,for(F),_), holds(F), R > 1, not one'(P),
    not holds'(G) :- preference(P,_,R',for(G),_), 1 < R',R' < R.

The remaining rules consist of the program implementing the composite preference type \textit{pareto}, as given in Section 4.2 on Page 19.

Altogether, these rules capture the semantics of ASO. To see this, consider a set \( O \) of ASO rules, and let \( E_{\text{pareto}} \) be the preference program for \textit{pareto}, \( E_{\text{aso}} \) be the previous rules for \textit{aso}, and \( F_O \) be the set of facts for the corresponding preference specification. Then, it can be shown that \( E_{\text{pareto}} \cup E_{\text{aso}} \cup F_O \cup G \) is a preference program for \( \succeq_O \).

7.3. Posets

In [8], qualitative preferences are modeled by a strict partially ordered set \((\Phi,\prec)\) of literals, also called a \textit{poset}. The literals in \( \Phi \) represent propositions that are preferably satisfied and the strict partial order \( \prec \) on \( \Phi \) gives their relative importance. We adapt this to sets of Boolean formulas. Then, for sets \( X,Y \) of atoms and a strict partially ordered set \((\Phi,\prec)\), \( X \succ_{(\Phi,\prec)} Y \) if there exists a formula \( \phi \in \Phi \) such that \( X \models \phi \) and \( Y \not\models \phi \), and for every formula \( \phi \in \Phi \) such that \( Y \models \phi \) and \( X \not\models \phi \), there is a formula \( \phi' \in \Phi \) such that \( \phi' < \phi \) and \( X \models \phi' \) but \( Y \not\models \phi' \). For our purposes, we additionally define the preorder \( \succeq_{(\Phi,\prec)} \) as follows: \( X \succeq_{(\Phi,\prec)} Y \) if either \( X \succ_{(\Phi,\prec)} Y \), or for all \( \phi \in \Phi \) it holds that \( X \models \phi \) if and only if \( Y \models \phi \).

We represent a partially ordered set \((\Phi,\prec)\) by a preference statement \( s_{(\Phi,\prec)} \) of form:
\[
#\text{preference}(s_{(\Phi,\prec)},\text{poset}) \Phi \cup \{ \phi' \gg \phi \mid \phi' < \phi \}.
\]

The preference type \textit{poset} captures the preference relations \( \succeq_{(\Phi,\prec)} \) for all strict partially ordered sets \((\Phi,\prec)\).

The core implementation of preference type \textit{poset} is given in Lines 1-13 below. In fact, Line 1 to 4 are only given for convenience to project the components of \((\Phi,\prec)\).

\begin{verbatim}
1 poset(P,F) :- preference(P,poset),
2     preference(P,_,_,for(F),_).
3 poset(P,F,G) :- preference(P,poset),
4     preference(P,I,1,for(F),_), preference(P,I,2,for(G),_).
5 better(P,F) :- preference(P,poset),
6     poset(P,F), holds(F), not holds'(F).
7 notbetter(P) :- preference(P,poset),
8     poset(P,F), not holds'(F).
\end{verbatim}
poset(P,F), not holds(F), holds'(F),
not better(P,G) : poset(P,G,F).

better(P) :- preference(P,poset),
better(P,_), not notbetter(P).

Given the reification of two sets \(X,Y\) in terms of \(\text{holds}\) and \(\text{holds'}\), we derive an instance of \(\text{better}(P,F)\) whenever \(X \models \phi_F\) but \(Y \not\models \phi_F\) (and \(F\) is the representation of \(\phi_F\)). Additionally, we derive \(\text{notbetter}(P)\) whenever there is a formula \(\phi_F\) such that \(Y \models \phi_F\), \(X \not\models \phi_F\), and \(\text{better}(P,G)\) fails to hold for all \(\phi_G\) preferred to \(\phi_F\) by the strict partial order \(<\). Then, these two auxiliary predicates are combined in Line 12 and 13 to define the preference type \(\text{poset}\).

Finally, we sketch how these rules capture the intended semantics. For this, given a strict partially ordered set \((\Phi,<)\), we let \(E_{\text{poset}}\) consist of the previous rules in Lines 1-13, and \(F_{(\Phi,<)}\) stand for the facts of the corresponding preference specification. Then, it can be shown that \(E_{\text{poset}} \cup F_{(\Phi,<)} \cup G\) is a preference program for \(\succeq_{(\Phi,<)}\).

### 7.4. Basic desires

Son and Pontelli [7] propose a language for specifying preferences in planning that distinguishes three types of preferences: basic, atomic, and general preferences. A basic preference is originally expressed by a propositional formula using Boolean as well as temporal connectives. Given that our focus does not lie on planning, we restrict basic preferences to Boolean formulas. Then, for sets \(X,Y\) of atoms and a formula \(\phi\), [7] defines \(X \succeq_{\phi} Y\) by \(Y \models \phi\) implies \(X \models \phi\), from which it follows that \(X \succ_{\phi} Y\) if \(X \models \phi\) and \(Y \not\models \phi\).

In *asprin*, such a basic preference is declared by a preference statement \(s_{\phi}\) of form

```
#preference(s_{\phi},basic)\{ \phi \}.
```

The preference type \(\text{basic}\) is implemented by the following rule.

```
better(P) :- preference(P,basic), preference(P,_,_,for(F),_),
holds(F), not holds'(F).
```

Interestingly, atomic and general preferences can be captured by composite preferences pre-defined in *asprin’s* library. That is, the language constructs \(!, \&\), \(|\), and \(<\) directly correspond to *asprin’s* preference types \(\text{neg}, \text{and}, \text{pareto}, \text{and lexico}\). For brevity, we refrain from further details and refer the reader to [7] for formal definitions.
8. The *asprin* system

8.1. Overview

*asprin* is implemented in Python using *clingo*’s API [18]. This interface provides *clingo* objects maintaining a logic program and supporting methods for adding, deleting and grounding rules, as well as for solving the current logic program. This allows for continuously changing the logic program at hand without any need for re-grounding rules. Also, it benefits from information learned in earlier solving steps.

The model-driven approach creates a *clingo* object, grounds the input logic program and runs Algorithm 1 or 2, that interact continuously with the running *clingo* object. Due to the usage of functional terms, it can happen that grounding does not terminate and *asprin* runs indefinitely, just as it can happen with *clingo* (cf. [34]). Otherwise, the solving algorithms receive a finite and ground program, in accordance with the specification of Section 5.

On the other hand, the implementation of the axiomatic approach combines Python with the meta-programming techniques of *clingo* (cf. footnote 13) to create the logic program (16) and solve it once with a *clingo* object.

The system *asprin* is publicly available at https://potassco.org/asprin.

8.2. Library

The current *asprin* library includes the preferences that we have seen in the previous sections: *more* and *less* as regards *cardinality* and *weight*, respectively, *subset* and *superset*, *aso* [4], *poset* [22], *lexico*, *pareto*, *and* and *neg* [7]. The library also includes the preference types *cp*, *maxmin*, and *minmax*. The first one represents CP-nets [30] and was introduced in [31]. The other two were included in [27] as part of an approach to compute diverse optimal solutions. While *maxmin* aims at maximizing the minimum value among a set of sums, *minmax* aims at minimizing the maximum value of such a set. See Section 10 for more details about these extensions.

8.3. Input

The input of *asprin* consists of a set of ASP files structured by means of *clingo*’s *#program* directives into base and preference programs. Base programs consist typically of a regular logic program along with a preference specification (just as with *#minimize* statements).

---

21If no preference specification is given, *asprin* computes stable models of the base program.
ported from asprin’s library, and/or from the input files provided by the user. Rules common to all types of preference programs are grouped under program blocks headed by ‘#program preference.’, while type-specific ones use ‘#program preference(t).’ where t is the preference type. Among all these type-specific programs, asprin only loads those for the preference types appearing in the preference specification of the base program. On the other hand, for every preference type t of the preference specification, asprin requires a corresponding preference program ‘preference(t)’. asprin’s implementation relies on the correctness of preference specifications. In other words, if the preference programs implement correctly the corresponding preference types, then asprin also functions correctly.

8.4. Usage

asprin can be configured by several command line options. As with standard ASP solvers, a natural number n tells asprin how many optimal models should be computed (where 0 initiates the computation of all optimal models). By default, asprin computes optimal models using Algorithms 1 and 2. Option --project allows for projecting the optimal models on the atoms occurring in the preference specification. Options for modifying the underlying clingo solver can be directly issued from the command line. More options and details are obtained with asprin’s --help option.

8.5. Heuristic support

Optimization problems are clearly more difficult than decision problems, since they involve the identification of optimal solutions among all feasible ones. To this end, it seems advantageous to direct the solving process towards putative optimal solutions by supplying heuristic information. Although this runs the risk of search degradation [35], it has already provided very promising prospects by improving regular optimization in ASP [36] as well as with poset preferences [8]. While the latter had to be realized by modifying solver implementations, in asprin we draw upon the integration with clingo’s declarative heuristic framework [36]. Heuristic support is added to logic programs via directives of form

\[
\text{#heuristic } a \ : \ l_1, \ldots, l_n \ : \ [k, m]
\]

\[22\]This syntax is analogous to the one of weak constraints [17].
where \( a \) is an atom and \( l_1, \ldots, l_n \) are literals. The integer value for \( k \) along with \init, \factor, \level, \sign, \true, \) or \false\ for \( m \) determine a heuristic modification to \( a \) provided that \( l_1, \ldots, l_n \) hold [36].

Different types of heuristic information can be controlled with clingo's domain heuristic along with the basic modifiers \sign, \level, \init, and \factor. In brief, \sign allows for controlling the truth value assigned to variables subject to a decision within the solver, while \level establishes a ranking among atoms such that unassigned atoms of highest rank are chosen first. With \init, a value is added to the initial heuristic score of an atom. The whole search is biased with \factor by multiplying heuristic scores by a given value. Furthermore, modifiers \true and \false are defined as the combination of a positive \sign and a \level, and a negative \sign and a \level, respectively. See [36] for details.

This framework seamlessly integrates into \asprin by means of the command line option \--domain-heuristic=<m>[,\langle v \rangle], that applies heuristic modifier \( m \) with value \( v \) (1 by default) to the formulas occurring in preference statements. In the implementation, this results in the addition of the heuristic directive

\[
\text{#heuristic holds}(X) : \text{preference}(_,_,_,\text{for}(X),_). \ [v,m]
\]

For example, the option \--domain-heuristic=false tells the underlying solver to decide first on formulas appearing in preference statements and to assign \false to them. As another example, we can replicate the modification of the sign heuristic proposed in [8] for \poset just by issuing the option \--domain-heuristic=sign, that leads to assigning \true when deciding on formulas of a preference statement. In general, the goal of these heuristic specifications is to direct the search towards optimal solutions in such a way that fewer intermediate solutions have to be computed. As we see in the next section, this often helps boosting the performance of \asprin.

9. Experiments

This section is devoted to the empirical analysis of \asprin’s performance. To this end, we conducted several experimental series addressing the following questions:

1. How does \asprin’s general approach compare to dedicated implementations? We consider:
(a) cardinality and weight optimization to contrast asprin using preference types cardinality and weight, with clingo using model-driven optimization as in its default setting;

(b) subset optimization to contrast asprin using preference type subset, with a saturation-based approach using disjunctive logic programs with clingo;

(c) poset preferences to contrast asprin using preference type poset, with satpref [37], that relies on inner solver modifications;

(d) ASO optimization to contrast asprin using preference type aso, with the dedicated system presented in [38], that wraps clingo.

2. What is the effect of heuristics on asprin’s performance?
We consider cardinality and weight as well as subset optimization to contrast different strengths of heuristic support. Additionally, for poset we evaluate the modification of the sign heuristic in both asprin and satpref.

3. What is the effect of the number of aggregated preferences on asprin’s performance?
We consider multi-objective optimization using pareto and lexico to contrast different levels of granularity when aggregating cardinality- and weight-based preferences.

We interleave the study of questions 1a to 1d with the study of question 2, and we address question 3 at the end.

While we rely on benchmarks accompanying the dedicated systems in (1c) and (1d), respectively, we built a benchmark set consisting of 193 instances from eight different classes of optimization problems expressed in ASP. In detail, we have the following classes: 15-Puzzle, Crossing, and Valves stemming from the ASP competitions\textsuperscript{23} of 2009 and 2013; Ricochet Robots from [39], Circuit Diagnosis from [40] and adapted to ASP in [36], Metabolic Network Expansion from [41], Transcription Network Repair from [42], and Timetabling from [43]. All classes involve a single optimization statement;\textsuperscript{24} Valves and Timetabling deal with weight summation, all others with cardinality. We selected from each class (if possible) the 30 most runtime-consuming instances solvable in 300 seconds by clingo-4.\textsuperscript{25} The resulting set of benchmarks is summarized in Table 1 by giving the respective preference type, the number of instances, and the average number of (ground) preference elements.

\textsuperscript{23}Other competition classes were either too easy or too difficult.

\textsuperscript{24}This is originally expressed as a common minimize statement.

\textsuperscript{25}A cutoff at 900 seconds brought only a handful of additional instances.
<table>
<thead>
<tr>
<th>Benchmark class</th>
<th>Type</th>
<th>Instances</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricochet</td>
<td>c</td>
<td>30</td>
<td>20.00</td>
</tr>
<tr>
<td>Valves</td>
<td>w</td>
<td>30</td>
<td>56.63</td>
</tr>
<tr>
<td>Crossing</td>
<td>c</td>
<td>24</td>
<td>211.92</td>
</tr>
<tr>
<td>Puzzle</td>
<td>c</td>
<td>7</td>
<td>580.57</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>c</td>
<td>30</td>
<td>1669.00</td>
</tr>
<tr>
<td>Repair</td>
<td>c</td>
<td>30</td>
<td>6750.73</td>
</tr>
<tr>
<td>Expansion</td>
<td>c</td>
<td>30</td>
<td>7501.87</td>
</tr>
<tr>
<td>Timetabling</td>
<td>w</td>
<td>12</td>
<td>23687.75</td>
</tr>
</tbody>
</table>

Table 1: Benchmark classes of ASP optimization problems.

We ran all benchmark instances with asprin-1 and clingo-4\textsuperscript{26} on Linux machines with Intel dual-core Xeon 3.4 GHz processors, imposing a limit of 900 seconds and 4 gigabyte of memory per run. We always used asprin’s default configuration, that executes Algorithms 1 and 2.

The tables where we present the results have all the following form. Rows represent different benchmark classes, and columns represent different system configurations. The entries of each cell (above the last line) provide the average runtime per class and configuration. The number of timeouts is given in parentheses. To calculate the average runtimes, timeouts are counted as 900 seconds throughout all experiments.\textsuperscript{27} The last line gives, for each configuration, the average of the runtimes per class and the sum of the timeouts. The best values per line are highlighted in bold. Additional data about the experiments can be found at https://github.com/potassco/asprin.

9.1. Questions (1a) and (2)

We address question (1a) by comparing the performance of asprin and clingo on cardinality and weight optimization. We use clingo’s default setting using model-driven optimization, that is similar to the configuration used by asprin. See [44] for alternative optimization modes of clingo. Afterwards, for question (2), we analyze the impact of heuristic information on the performance of

\textsuperscript{26}More in detail, we used asprin-1.1 and clingo-4.5 in all experiments except on the evaluation of aggregate preferences of question 2 and in the saturation-based approach of question 1b, where we used asprin-1.0 and clingo-4.4. The changes between the different versions are minor.

\textsuperscript{27}The additional data about the experiments contains a document with versions of all tables of this section where each timeout is counted as 9000 seconds (PAR10).
both systems in the same benchmarks. In total, for each of clingo and asprin there are four configurations: without heuristics, modifying the heuristic sign, the level or both, viz. false. We refer to the three last settings using the subscripts s, l and f, respectively. For clingo, the heuristic modifications are applied using option --dom-mod, while for asprin we use option --domain-heuristic. Table 2 summarizes the results.

<table>
<thead>
<tr>
<th></th>
<th>clingo</th>
<th>clingo_s</th>
<th>clingo_l</th>
<th>clingo_f</th>
<th>asprin</th>
<th>asprin_s</th>
<th>asprin_l</th>
<th>asprin_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricochet</td>
<td>407 (5)</td>
<td>425 (5)</td>
<td>74 (0)</td>
<td>74 (0)</td>
<td>432 (4)</td>
<td>407 (4)</td>
<td>68 (0)</td>
<td>71 (0)</td>
</tr>
<tr>
<td>Valves</td>
<td>64 (0)</td>
<td>55 (0)</td>
<td>392 (11)</td>
<td>728 (24)</td>
<td>69 (0)</td>
<td>65 (0)</td>
<td>460 (11)</td>
<td>715 (22)</td>
</tr>
<tr>
<td>Crossing</td>
<td>60 (0)</td>
<td>75 (0)</td>
<td>731 (17)</td>
<td>381 (6)</td>
<td>104 (1)</td>
<td>98 (0)</td>
<td>805 (20)</td>
<td>387 (6)</td>
</tr>
<tr>
<td>Puzzle</td>
<td>81 (0)</td>
<td>105 (0)</td>
<td>128 (0)</td>
<td>372 (1)</td>
<td>82 (0)</td>
<td>112 (0)</td>
<td>136 (0)</td>
<td>416 (1)</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>88 (0)</td>
<td>90 (0)</td>
<td>27 (0)</td>
<td>144 (4)</td>
<td>196 (3)</td>
<td>76 (0)</td>
<td>43 (0)</td>
<td>118 (2)</td>
</tr>
<tr>
<td>Repair</td>
<td>93 (0)</td>
<td>10 (0)</td>
<td>7 (0)</td>
<td>8 (0)</td>
<td>76 (0)</td>
<td>15 (0)</td>
<td>71 (2)</td>
<td>8 (0)</td>
</tr>
<tr>
<td>Expansion</td>
<td>143 (0)</td>
<td>7 (0)</td>
<td>16 (0)</td>
<td>11 (0)</td>
<td>216 (0)</td>
<td>10 (0)</td>
<td>38 (0)</td>
<td>12 (0)</td>
</tr>
<tr>
<td>Timetabling</td>
<td>106 (1)</td>
<td>13 (0)</td>
<td>825 (11)</td>
<td>3 (0)</td>
<td>345 (3)</td>
<td>255 (2)</td>
<td>900 (12)</td>
<td>6 (0)</td>
</tr>
<tr>
<td>Total</td>
<td>130 (6)</td>
<td>97 (5)</td>
<td>275 (39)</td>
<td>215 (35)</td>
<td>190 (11)</td>
<td>130 (6)</td>
<td>315 (45)</td>
<td>217 (31)</td>
</tr>
</tbody>
</table>

Table 2: Cardinality and weight optimization with clingo and asprin.

For question (1a), comparing clingo and asprin without heuristics, we observe that their runtimes are similar for all classes except Diagnosis, Expansion and Timetabling. Note that for Crossing the time difference is due to a single instance where asprin timeouts. Without this instance, in that class clingo averages 58 seconds and asprin goes down to 70. The three classes where the performance of asprin is worse, comprise optimization statements involving large sets of atoms, but this cannot be a direct cause of the slowdown in view of the performance improvement on Repair. In fact, a closer look reveals that those three classes exhibit the longest convergence to the optimum. For this analysis, we focus only on the instances that were solved by both clingo and asprin.28 It turns out that the average number of models computed by asprin in Diagnosis, Expansion and Timetabling is 341, 299 and 287, respectively, while for the other classes that number is always below 50. These values are similar for clingo, that never differs more than 15% (up or down, depending on the class) with respect to asprin. For these three classes, this high number of intermediate models leads asprin to spend

---

28We have found that the additional data, such as the number of enumerated models, is not always reliable for the runs that timed out. Hence, when we use this additional data, we focus on instances where none of the compared systems timed out.

44
on the `solve` calls only 72%, 73% and 11% of its runtime, respectively. The rest of the time is spent mainly on grounding the preference programs at each of the many iterations of the solving algorithm. This is obviously a bottleneck of our approach. However, as we see shortly, it can be remedied by improving the convergence to the optimum using heuristics. To complete the picture, we add that `asprin` in the other classes always spends more than 90% of its time on the `solve` calls, while `clingo` spends in all classes almost all of its time on solving.

We move now to question (2), where we evaluate the heuristic modifications on these benchmarks. In `asprin`, as can be seen in Table 2, heuristics do not improve much on `Valves`, `Crossing` and `Puzzle`, but in the other five classes there is always at least one modification that boosts the performance of the system. Overall, it seems that modifying the `sign` is the best compromise, but the best heuristic option must be decided case-by-case. The more intrusive modifications of `level` and `false` lead to very bad results in some classes, but they lead to huge improvements in others. This agrees with previous results in the literature, that show that in theory the restriction of the choices of the solver may generate exponentially larger search spaces [35], while in practice it can also result in significant speedups, cf. [36] and [8].

Focusing now on the instances that are solved by both `asprin` and `asprin_s`, we observe that the average number of models enumerated by `asprin_s` is almost the same as with `asprin`, except in the classes that were problematic before, `Diagnosis`, `Expansion` and `Timetabling`, where it has decreased to 66, 15 and 141, respectively. As a consequence, now most of the runtime is spent on the `solve` calls, except on `Timetabling` where solving only amounts to 17% of the total runtime. The values of `asprin` in these instances are basically the same as before. These results help to explain the improvement on those three classes, and the similar runtimes on `Ricochet`, `Valves` and `Crossing`. The worse runtime in `Puzzle` and the better runtime in `Repair` seem to be due to the adequacy of the heuristic modification for searching each single model. In the first case, the `sign` heuristic hinders the search, while in the second it boosts it.

Comparing now `asprin` and `asprin_f`, we can see that the average number of models enumerated by `asprin_f` decreases abruptly. It is less than 3 in all classes except on `Crossing`, where it is 6. Consequently, now in all classes the solving time is very close to the total runtime. Interestingly, when we look at the solving time per enumerated model, we see that those values are always higher for `asprin_f` than for `asprin`, and the difference is more acute in the classes `Valves`, `Ricochet` and `Puzzle`, where `asprin_f` performs very badly. This means that the `false` heuristic hinders the search of each single model. But at the same time we have seen that it
improves the convergence to an optimum. It seems that it is the balance between
these two factors what determines the diverse results that we observe. Moreover,
it turns out that we reach a similar conclusion from having a closer look at the
results of asprin, but we do not detail them here for brevity.

Turning our attention to clingo, Table 2 shows that the effect of heuristics in
this system is similar to their effect in asprin. And comparing clingo and asprin
both enhanced with heuristics, we observe that asprin is still slower than clingo
but very close to it. The best configuration of asprin is only 10 or less seconds
slower than the best configuration of clingo on all classes except on Crossing and
Diagnosis, where asprin needs 38 and 16 seconds more in average, respectively.
Overall, heuristics clearly improve the convergence to the optimal, and in this way
the additional operations of asprin that hindered its performance in the basic case
are no longer that important, helping asprin to narrow the gap with clingo.

9.2. Questions (1b) and (2)

We consider now subset optimization to answer question (1a), and the effect
of heuristics in this setting to answer question (2). In ASP, this problem is tradi-
tionally solved via saturation-based encodings using disjunctive logic programs.
The metasp system [24] implements this technique by compiling a normal logic
program along with a subset-oriented optimization statement into such a disjunc-
tive logic program, which can then be solved with clingo (see also [25]). This is
very similar to the axiomatic approach of asprin, and from our experience their
performance is comparable. Here, we only evaluate the former, and we contrast it
with asprin without and with heuristics. The benchmarks of this experiment are
based on the ones for cardinality and weight optimization, but we have replaced
the original optimization statements by subset preferences over the atoms in the
original statements. The results of the experiments are shown in Table 3.

For question (1a), the first two columns show that asprin (using the mode-
driven approach) clearly outperforms metasp. Moreover, they show that finding
subset-minimal models is easier than finding cardinality- or weight-minimal ones.

The answer to question (2) for subset optimization is the same as for
cardinality- and weight- optimization. In the classes where asprin enumerates
few models, Valves, Crossing, and Puzzle, heuristics lead to no or small improve-
ments. On the others, heuristics can improve the convergence to the optimum,
and boost the performance of asprin. As before, which heuristic to use should be
decided on a case-by-case basis.

46
9.3. Questions (1c) and (2)

We compare asprin with the satpref system for poset preferences [37], as put forward in question (1c). Interestingly, satpref not only extends the SAT solver minisat with branch-and-bound-based optimization but also uses sign-based heuristics for boosting optimization. Then, we address question (2) by comparing both systems using those heuristics.²⁹

We conducted our comparison using random and structured benchmarks from [37]. Table 4 presents the results of the comparison on random benchmarks. Every class consists of 100 random instances, each with 500 variables and 1750 clauses, in which an order \(a > b\) or \(b > a\) between variables \(a\) and \(b\) is generated with the probabilities listed in the left column. The results for additional random classes from [37], with at most 125, 250, or 275 variables subject to optimization, are similar to those of Table 4, and we do not show them here. Table 5 presents the results of the comparison on structured benchmarks, showing only the benchmark classes where the average solving time of asprin using the sign heuristic was above 1 second. Starting with Maxsat and following the order in which they are listed in Table 5, they consist of 35, 16, 28, 188, 148 and 15 instances, respectively. In our experiments, we compared both systems in their basic setting and with sign-based heuristics (s).

Overall, the results of our comparison show that the general-purpose approach of asprin is comparable to the dedicated approach of satpref, and that heuristics

<table>
<thead>
<tr>
<th></th>
<th>metasp</th>
<th>asprin</th>
<th>asprin_s</th>
<th>asprin_l</th>
<th>asprin_f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricochet</td>
<td>811 (24)</td>
<td>365 (3)</td>
<td>461 (10)</td>
<td><strong>69</strong> (0)</td>
<td>71 (0)</td>
</tr>
<tr>
<td>Valves</td>
<td>900 (30)</td>
<td><strong>38</strong> (0)</td>
<td>39 (0)</td>
<td>339 (6)</td>
<td>673 (21)</td>
</tr>
<tr>
<td>Crossing</td>
<td>62 (0)</td>
<td><strong>0</strong> (0)</td>
<td>1 (0)</td>
<td>7 (0)</td>
<td>3 (0)</td>
</tr>
<tr>
<td>Puzzle</td>
<td>35 (0)</td>
<td>31 (0)</td>
<td>32 (0)</td>
<td><strong>21</strong> (0)</td>
<td>51 (0)</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>182 (6)</td>
<td>19 (0)</td>
<td>2 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Repair</td>
<td>900 (30)</td>
<td>8 (0)</td>
<td>3 (0)</td>
<td><strong>1</strong> (0)</td>
<td>1 (0)</td>
</tr>
<tr>
<td>Expansion</td>
<td>900 (30)</td>
<td>64 (0)</td>
<td>14 (0)</td>
<td>4 (0)</td>
<td><strong>3</strong> (0)</td>
</tr>
<tr>
<td>Timetabling</td>
<td>799 (10)</td>
<td>217 (2)</td>
<td>21 (0)</td>
<td>900 (12)</td>
<td><strong>5</strong> (0)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>574 (130)</td>
<td>93 (5)</td>
<td><strong>72</strong> (10)</td>
<td>168 (18)</td>
<td>101 (21)</td>
</tr>
</tbody>
</table>

Table 3: Subset optimization with metasp and asprin.

²⁹In [13], we also evaluated more elaborated heuristics, that guarantee that the first computed model is optimal [8]. The interested reader can look at the results in that paper. We do not detail them here because the performance is worse than with sign-based heuristics, and also because that approach in asprin has been superseded by the recent work in [45] (see Section 10).
improve the performance of both systems. In the random benchmarks, asprin is faster than satpref on the class with more preferences, while satpref is faster on the others. In the structured benchmarks, asprin is faster on Pseudo-routing, on pair on Pseudo-primes, but slower on the other classes. As before, heuristics decrease the number of enumerated models and, as a result, decrease the total runtime for both systems. However, a closer look at the results shows that asprin still spends only a small portion of its runtime on the solve calls. More in detail, there is no class where it spends more than 12% of its runtime on those calls. This suggests that there is still room for improvement in the implementation of the system.

9.4. Question (1d)

We compare asprin with the system for ASO preferences from [38], that implements a branch-and-bound approach in C++ and calls clingo each time from scratch via a system call. We refer to this system as aso.

We used the benchmark generator from [38] to generate random 3CNF formulas with \( n \) variables and \( 4n \) clauses. For each formula of \( n \) variables, it randomly generates \( 3n \) preference rules with \( a > \neg a \) or \( \neg a > a \) for some \( a \) in the head, and 0.
to 2 literals in the body. In addition, the approach handles ranked ASO preferences \((aso_1)\), which amount to an aggregation of \(aso\) preferences with \(lexico\) in \(asprin_1\). The generator assigns a higher rank to half of the ASO rules to account for this.

We created a benchmark set comprising 20 instances for every value of \(n\) ranging from 350 to 490 in increments of 10. Table 6 shows the results of the comparison of \(asprin\) and \(aso\) on this benchmark set. Overall, we observe that the general-purpose approach of \(asprin\) is comparable with the dedicated approach of \(aso\). On the other hand, we observed with \(asprin\) a very fast convergence, so that no real difference can be expected on this set of instances.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(aso)</th>
<th>(aso_1)</th>
<th>(asprin)</th>
<th>(asprin_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>350</td>
<td>9 (0)</td>
<td>17 (0)</td>
<td>4 (0)</td>
<td>5 (0)</td>
</tr>
<tr>
<td>360</td>
<td>14 (0)</td>
<td>22 (0)</td>
<td>48 (0)</td>
<td>50 (0)</td>
</tr>
<tr>
<td>370</td>
<td>15 (0)</td>
<td>25 (0)</td>
<td>38 (0)</td>
<td>39 (0)</td>
</tr>
<tr>
<td>380</td>
<td>10 (0)</td>
<td>23 (0)</td>
<td>8 (0)</td>
<td>9 (0)</td>
</tr>
<tr>
<td>390</td>
<td>59 (0)</td>
<td>72 (0)</td>
<td>50 (1)</td>
<td>52 (1)</td>
</tr>
<tr>
<td>400</td>
<td>22 (0)</td>
<td>33 (0)</td>
<td>28 (0)</td>
<td>30 (0)</td>
</tr>
<tr>
<td>410</td>
<td>87 (1)</td>
<td>96 (1)</td>
<td>124 (2)</td>
<td>125 (2)</td>
</tr>
<tr>
<td>420</td>
<td>97 (1)</td>
<td>108 (1)</td>
<td>60 (0)</td>
<td>62 (0)</td>
</tr>
<tr>
<td>430</td>
<td>68 (0)</td>
<td>79 (0)</td>
<td>144 (0)</td>
<td>147 (0)</td>
</tr>
<tr>
<td>440</td>
<td>165 (3)</td>
<td>175 (3)</td>
<td>165 (2)</td>
<td>167 (2)</td>
</tr>
<tr>
<td>450</td>
<td>45 (0)</td>
<td>61 (0)</td>
<td>52 (0)</td>
<td>54 (0)</td>
</tr>
<tr>
<td>460</td>
<td>112 (1)</td>
<td>125 (1)</td>
<td>117 (2)</td>
<td>120 (2)</td>
</tr>
<tr>
<td>470</td>
<td>201 (4)</td>
<td>210 (4)</td>
<td>161 (2)</td>
<td>162 (2)</td>
</tr>
<tr>
<td>480</td>
<td>152 (2)</td>
<td>165 (2)</td>
<td>70 (1)</td>
<td>72 (1)</td>
</tr>
<tr>
<td>490</td>
<td>206 (2)</td>
<td>218 (2)</td>
<td>265 (4)</td>
<td>267 (4)</td>
</tr>
<tr>
<td>Total</td>
<td>84 (14)</td>
<td>95 (14)</td>
<td>89 (14)</td>
<td>91 (14)</td>
</tr>
</tbody>
</table>

Table 6: ASO optimization with \(aso\) and \(asprin\).

9.5. Question (3)

We investigate now the effect of the number of aggregated preferences on the performance of \(asprin\). To do this, we convert the cardinality and weight mono-objective problems from before into multi-objective problems, using separately \(pareto\) and \(lexico\) for their composition. We begin by dividing the atoms in a preference statement into 16 basic statements of the same type. Then, we use 1, 3, 7, or 15 \(pareto\) (or \(lexico\)) statements to combine the 16 basic statements in a tree-like structure, in such a way that the four aggregations represent the same preference relation. This experimental design allows us to attribute different runtimes observed in the experiments to different forms of aggregation. Figure 1 represents
compositions \( \text{pareto}-1, -3, -7 \) and \(-15\), where the \( b_i \)'s are basic statements, and the \( p_i \)'s are pareto statements. The representation using \( \text{lexico} \) is similar, but the pareto statements \( p_i \) are replaced by lexico statements \( l_i \), whose priority decreases as their subindex \( i \) increases.

![Diagram of Pareto aggregation using different numbers of composite preferences.](image)

Tables 7 and 8 summarize the results for \( \text{pareto} \) and \( \text{lexico} \), respectively, using \( \text{asprin}'s \) default configuration.

Overall, we find that the number of composite preferences does not significantly impact the performance or \( \text{asprin} \), except in the classes \( \text{Repair} \) and \( \text{Expansion} \) using \( \text{pareto} \), where the performance deteriorates slightly. At the moment, we do not have a good explanation for these exceptions. On the other hand, we observe that \( \text{pareto} \) optimization is faster than \( \text{lexico} \) optimization, which is expected since, by design, every \( \text{lexico} \) optimal model is also \( \text{pareto} \) optimal.
9.6. Summary

In general, the experiments show that asprin compares well to dedicated implementations. It is faster than the saturation-based approach of metasp for subset optimization, and it is as fast as the dedicated implementations for poset and ASO. The only exception occurs in the comparison with clingo in the basic setting, where both systems perform similarly in some classes (5 out of 8) but asprin is clearly slower in the others. A closer look reveals that in those classes asprin enumerates many intermediate models and spends most of its time grounding the many intermediate preference programs. This is a bottleneck of this approach, and we plan to improve this part of the implementation in the future.

The experiments show clearly that domain specific heuristics can boost the performance of asprin. They often improve the convergence to the optimum, leading in some cases to huge speedups. In particular, they help to close the gap with clingo, although in the end clingo is still a bit faster. Which heuristic
modification is better depends on the class of each instance, but is stable among
the instances of the same class.

Finally, our study of pareto and lexico preferences shows that, in general, the
number of those composite preferences does not hinder the performance of asprin,
although this was not the case in 2 classes (out of 8) on pareto optimization.

10. Discussion

This paper introduces a general, flexible and extensible framework for pref-
erence handling in ASP. Our intention was not primarily to come up with new
preference relations on stable models that have not been previously studied (al-
though one can certainly introduce such new relations in asprin). Rather our goal
was to provide ASP technology matching the substantial research on preference
handling in ASP and beyond. Essentially, we wanted to put this research into
practice. We believe that asprin may play a similar role for answer set optimiza-
tion as the development of efficient ASP solvers had in boosting the basic answer
set solving paradigm.

There are two types of users of asprin: those who are happy using the prefer-
erence relations in the asprin library, and those who want to exploit the extensibility
of the system and define their own preference orderings. For the former, much of
the technical capabilities of the system are not needed. In fact, they can use as-
prin as a preference handling system where all one needs to know are the available
preference types and their arguments. For the latter type of users, let us call them
preference engineers, the system provides all the additional functionality to define
interesting new preference orderings. We see examples of both in this section.

From a knowledge representation perspective, the approach closest to ours
is [5], that introduces a specific preference language with a set of basic and com-
posite preference types. Those preference types are predefined, but the language
as such is presented as open to extensions. Our work in asprin can be seen as a
generalization and implementation of this approach, providing that extensibility
and offering additional flexibility.

From a solving perspective, our work is inspired by the methods presented
in [4, 46, 8, 24]. In particular, [4] presents a version of Algorithm 1 for ASO, [46]
and [8] implement Algorithms 1 and 2 for poset, and [24] implements a version of
the axiomatic approach for the preferences pareto, lexico, subset and less(weight).
Recent advances in solving logic programs with weak constraints and subset pref-
erences are discussed in [47] and [48], respectively. While the methods presented
in all those papers are defined for specific types of preferences, our algorithms can
handle any preference type defined in ASP. The key for the generality of our approach is the usage of meta-programming to specify the preference programs for the different preference types. This provides a simple way to add new preferences to asprin, while at the same time it allows our algorithms to be applicable to any preference type defined that way.

In the remainder of this section, we discuss other related work. Some of it precedes the original publication of asprin in [12], some of it was published afterwards. We start describing two of our extensions to asprin. Then, we focus on abstract approaches to preferences and related theoretical results. Next, we comment on different preference languages of the literature. The work published in this topic is quite extensive, and we concentrate on the approaches that are of special interest to us. For the interested reader, an overview of the work on preferences in Artificial Intelligence can be found in the Special Issue on Preferences of Artificial Intelligence [49], and in the tutorial [50]. A comprehensive, though somewhat dated, survey of approaches to preferences in logic programming can be found in [51], and a general overview of the topic can be found in [52]. We close this section with a description of different applications of asprin.

10.1. Two extensions of asprin

Combinatorial problems often have numerous solutions, and preferences can be used to select the optimal ones. However, even after applying preferences, sometimes we are left with a large number of optimal solutions. In these cases, it can be useful to identify small subsets of diverse optimal solutions. To this aim, [53] studied the computation of diverse stable models in plain ASP, and [27] generalized this to logic programs with preferences in asprin. This extension of asprin allows us to compute a set of optimal solutions that are as diverse as possible. For doing so, it introduces various methods, including generalizations of previous work to logic programs with preferences. We also mention that, for ASO preferences, [38] investigated the problem of computing an optimal answer set that is diverse with respect to another solution, and presented various algorithms for this task.

On another issue, the experiments in Section 9 show that there is still a performance gap between clingo and asprin when dealing with cardinality and weight optimization. This raises the question of how to leverage the native solving functionalities of clingo to improve the performance of asprin. This has been answered in [45], where all preference types implemented in asprin are mapped to either weak constraints or heuristic directives, in such a way that one optimal model can
be computed using clingo’s native machinery. The results in that paper indicate that this approach effectively closes the gap observed in our experiments.

10.2. Abstract and theoretical approaches to preferences

The work of [45] takes a more abstract approach than this paper. Instead of logic programs, it considers knowledge bases, that are simply defined as sets of models. Such an abstract approach is interesting because it can be instantiated by different languages, that are used to specify those models. Then, the results obtained for the abstract approach apply to those specific languages. In this line, [54] introduces and studies weighted abstract modular systems. Such systems consist of two parts: a set of modules, possibly written in different languages over different vocabularies, that define a set of models; and a set of weighted conditions, similar to weak constraints, that select the optimal models. The approach is instantiated by MaxSAT and logic programs with weak constraints, and allows for the study of the relations among them. In a related fashion, the work of [55] presents a model-theoretic framework for reasoning about preferences. In this case, the models (called structures) are defined by a formula in some logic, and the preference relation is defined by lifting a preorder over a set of ground atoms. The authors present three different methods of doing such a lifting, and the approach applies to any lifting method where the problem of deciding dominance can be solved in polynomial time. Related to this, [56] presents an in-depth study of methods for lifting a preorder. The work of [55] also introduces the problem of Model Finding (called Prioritized Model Expansion) and study the complexity of the problems of Optimality and Query (called Optimal Expansion Problem, and Goal-Oriented Optimal Expansion Problem, respectively). The authors provide a membership result for the Optimality problem and a completeness result for the Query problem that coincide with ours where they overlap, namely, where \( j = 0 \).

Faber et. al [57] introduce also an abstract approach to preferences. Their goal is to study the property of strong equivalence. In other words, they investigate under which conditions a logic program with preferences can be replaced by another. For this, they define abstract preference frameworks, and study strong equivalence in that general setting. The approach can be instantiated by different preference languages, to which the general results obtained for abstract preference frameworks apply. They focus on so-called separated preference frameworks, where one can identify two syntactic components: generators that determine sets of fea-
sible outcomes, and selectors that determine which of them are optimal. This fits with the approach of asprin, where logic programs are generators, and preference specifications are selectors. In fact, it seems that the language of asprin falls under the scope of these frameworks. This would allow us to apply to asprin the theorems about separated preference frameworks from [57], and obtain directly strong equivalence results for our approach. We leave this for future work.

Another contribution of [45] is the establishment of different relations between preference types. In particular, the work presents two types of translations from an input statement of some preference type to an output statement of another preference type. The first type of translation, called approximation, guarantees that the preorder defined by the output statement is a superset of the preorder defined by the input statement. This ensures that every optimal model with respect to the output statement is also optimal with respect to the input one. For example, a subset statement can be approximated by a less(cardinality) statement with the same preference elements. The second translation guarantees that the preorders defined by the input and the output statements are the same. As an example, poset can be translated in this way to subset, using additional atoms.

The works of [58] and [59] address related issues. In the first case, the authors compare the expressiveness and succinctness of different preference languages. Expressiveness, according to [58], is about what type of preorders can be expressed in one preference language. For example, less(weight) can express all complete preorders and nothing else. Succinctness is about the existence of polynomial translations between different preference languages, such that the preorder that results from the translation coincides with the input preorder. The work of [58] studies the case where the size of the output statements is polynomial in the size of the input statements. The work of [59] develops a similar study on succinctness, but in this case the requirement is a bit stronger, since the translations should be computable in polynomial time. Both studies consider a variety of preference languages, related to the ones we have studied and implemented here.

10.3. Preference languages

In Section 7, we have seen how many approaches to preferences can be modelled in our system. These include established ASP optimization techniques like #minimize directives [2] and weak constraints [3], but also ASO [4], poset [22]
and the language for specifying preferences in planning domains from [7]. In the following paragraphs, we comment on other preference formalisms.

We start with logic programs with ordered disjunction (LPODs, [6]), that extend logic programs with the capability of expressing alternatives with decreasing degrees of preference. For example, the following rule expresses that if it is hot then we prefer going diving, and only if that is not possible then we want to go to the beach:

\[
dive \times beach \leftarrow hot
\]

Together with the fact \( hot \leftarrow \), that rule leads to the unique optimal solution \( \{ hot, dive \} \), while adding the constraint \( \leftarrow dive \) the unique optimal solution becomes \( \{ hot, beach \} \). Observe how the same rule is used for the generation of stable models and for the selection of the optimal ones. Recent years have seen a growing interest in LPODs. For example, [60] introduced a novel model-theoretic semantics for LPODs, and [61] investigated a family of choice logics closely related to LPODs. Namely, Qualitative Choice Logic (QCL) is one of those choice logics, and LPODs can be seen as the combination of logic programs and QCL. An implementation of LPODs using asprin was presented in [62] and is available at http://reasoning.eas.asu.edu/lpod2asprin. The system translates an LPOD to the input language of asprin and runs our system. In this case, the authors acted as preference engineers and defined preference programs for the different preference relations underlying the semantics of LPODs.

Another related approach in ASP was presented in [63], where logic programs are extended with so-called consistency restoring rules, that only are activated if they are needed to restore the consistency of the rest of the logic program. For example, the following consistency restoring rule could be used in a diagnosis application to generate some abnormal faults only when they are necessary to obtain a model:

\[
r(C) : ab(C) \leftarrow component(C)
\]

The term \( r(C) \) is the identifier of the rule, that states that if \( C \) is a component, then \( ab(C) \) may be added to a stable model to recover consistency. This kind of rules can be represented in asprin using additional atoms as follows:

\[
\{ appl(r(C)) \} \leftarrow component(C)
ab(C) \leftarrow component(C), appl(r(C))
#preference(1, subset)\{ appl(r(C)) : component(C) \}
\]
The first rule chooses the application of consistency restoring rules, the second rule generates abnormalities whenever consistency restoring rules are applied, and the preference statement minimizes the application of consistency restoring rules, to ensure that they are only applied whenever they are needed to recover consistency. In addition to this, when different consistency restoring rules can be applied, it is possible to specify a preference over them. For example, the fact prefer\((r(c_1), r(c_2))\) ← can represent our preference towards faults of component \(c_1\) over faults of component \(c_2\). The semantics of the language define a relation between models using these prefer\(\, /\, 2\) atoms, but this relation is not a preorder, and therefore does not fit directly into our approach. In fact, it can happen that a satisfiable logic program becomes unsatisfiable after adding some preferences between consistency restoring rules, something that cannot happen in asprin when we add preferences to a satisfiable logic program.

We continue with weighted LARS [64], a framework for quantitative stream reasoning in ASP. This approach captures some quantitative extensions of ASP, and lifts them to the streaming setting. It is based in LARS, a kind of temporal language for stream reasoning in ASP, that includes usual temporal operators like eventually and always, and others specific to stream reasoning like a window operator. Formally, weighted LARS extends LARS programs by adding a weighted LARS formula that is interpreted over some semiring. This weighted formula assigns to every solution of the LARS program one element of the semiring, its weight. The framework is extended for preferential reasoning by providing a strict partial order over those weights. Then, optimal solutions are those whose weight is not dominated by the weight of another solution. In some respects, weighted LARS is more general than asprin. It applies to answer streams, and it uses weighted formulas to determine the weight of a solution. Compare this, for example, with a less\(\, (\text{weight})\) preference, that assigns a weight to a normal answer set, and uses a logic program to determine it. On the other hand, in wLARS, the specification of preferences is restricted to the combination of a weighted formula and a given order, while the highlight of asprin is precisely that it provides a very rich and extensible language for doing that thing.

Outside the field of ASP, preference trees [65, 66] and lexicographic preference trees [67] have received significant attention in the recent years, specially in connection with the problem of learning preferences. A preference tree is a binary tree, whose nodes are labeled by propositional formulas, that defines a total order over the interpretations of the formulas. As an example, the preference tree of Figure 2 expresses that we prefer hot weather, and whenever it is hot we prefer diving, while otherwise we prefer going to the sauna. More precisely, the pref-

57
Figure 2: A preference tree.

Preference tree defines an order where solutions that satisfy hot and dive are most preferred, followed by those that satisfy hot but not dive, and then by those not satisfying hot but satisfying sauna, that are preferred to those that neither satisfy hot nor sauna. The same order can be captured by the partially ordered set preference over formulas $\{\text{hot, hot} \land \text{dive, } \neg \text{hot} \land \text{sauna}\}$, $>$ where the partial order specifies that hot $>$ hot $\land$ dive and hot $> \neg$hot $\land$ sauna. We conjecture that preference trees can be captured in general like this. Lexicographic preference trees are a special type of preference trees where all labeling formulas are literals and from every path from the root to one leaf every atom occurs exactly once. It would be certainly interesting to add these preference types to asprin’s library.

CP-nets [30] are one of the main approaches to preferences in the literature. They allow us to represent preferences between alternatives under a ceteris paribus assumption, i.e., as long as everything else stays the same. For example, the following CP-net expresses that whenever it is hot we prefer going diving, and otherwise we prefer going to the sauna, everything else being equal:

$$
dive > \neg\text{dive} \parallel \text{hot} \quad \text{sauna} > \neg\text{sauna} \parallel \neg\text{hot}
$$

The net represents the order where $\{\text{dive, hot}\}$ is preferred to $\{\text{hot}\}$, $\{\text{dive, hot, sauna}\}$ is preferred to $\{\text{hot, sauna}\}$, $\{\text{sauna}\}$ is preferred to $\emptyset$, and $\{\text{dive, sauna}\}$ is preferred to $\{\text{dive}\}$. As can be seen in the example, the syntax of CP-nets is similar to the syntax of ASO, but their semantics are different. We refer the reader to the original paper on ASO [4] for a comparison between both languages. CP-nets have been integrated into asprin in [31]. Deciding whether one stable model is strictly better than another wrt a CP-net is in general PSPACE-complete [68]. For this case, asprin includes a normal preference program whose size is exponential in the size of the input in the worst case. For tree-shaped CP-nets, dominance can be decided in linear time [69]. Accordingly, asprin is equipped with a preference program specific to this class of CP-nets. This shows the versatility of asprin’s approach, that allows us to improve the performance of
the system by adding more efficient preference programs for specific cases. Similarly, for acyclic CP-nets, dominance testing is known to be NP-hard [30], but in asprin these nets are handled specifically by an approximation using poset.

The works of [56, 19] introduce preference languages where CP-nets can be easily embedded, while the complexity of dominance remains PSPACE-complete. The first one introduces a language where the user can specify different forms of atomic improvements between solutions, and a solution is better than another if there is a sequence of such improvements transforming the former in the latter. As an example, using the translation presented in [56], the previous CP-net can be represented by the following statements:

\[
\text{gain(dive)} : \text{in(hot)} \quad \text{gain(sauna)} : \neg\text{in(hot)}
\]

The second work introduces a preference logic where preference formulas are Boolean combinations of preference statements of the form \(\alpha \succ \beta \parallel F\)\(^31\) where \(\alpha\) and \(\beta\) are propositional formulas and \(F\) is a set of propositional formulas. The models of such formulas are preference relations \(\succeq\) that interpret Boolean connectives in the usual way, and that satisfy a statement \(\alpha \succ \beta \parallel F\) if for every two sets \(X\) and \(Y\) such that (i) both interpret the same way \(F\), (ii) \(X\) satisfies \(\alpha\) and (iii) \(Y\) satisfies \(\beta\), it holds that \(X \succ Y\). Following [19], the previous CP-net can be represented by the next preference formula:

\[
(dive \land \text{hot} > \neg\text{dive} \land \neg\text{hot} \parallel \{\text{sauna}\}) \land (\text{sauna} \land \neg\text{hot} > \neg\text{sauna} \land \neg\text{hot} \parallel \{\text{dive}\})
\]

It would be interesting to integrate these languages, or fragments of them, into asprin. The challenge in this case, as with CP-nets, is to find some subsets of the languages that are useful and at the same time can be implemented efficiently.

In the field of constraint processing (CP), the work of [70, 71] can be seen as paralleling the approach of asprin in that area, with the purpose of representing and reasoning about solution dominance. In that work, the authors define constraint dominance problems, that extend the usual constraint satisfaction problems by dominance relations, just like in asprin logic programs are extended by preference relations. Similarly, in the implementation, dominance nogoods fulfill the role of preference programs in asprin.

10.4. Applications of asprin

We begin with the implementations of some description logics with preferences by L. Giordano and D. Dupré. The work in [72] presents a nonmonotonic

\(^{31}\)There are also non-strict preference statements where \(\succ\) is replaced by \(\sqsupset\).
extension of the description logic $SROEL(\sqcap, \times)$ for defeasible reasoning, and applies asprin to reason in this logic. The implementation uses the preference types pareto, lexico and less(weight). The same authors, in [73, 74], extend description logics of the $\mathcal{EL}$ family with preferences (ranks and weights over defeasible inclusions) and reduce reasoning in those logics to reasoning in asprin. In this case, the authors have defined a new preference type that matches the preference semantics defined for that logic. In would be interesting to see if those kind of preferences could be captured by the preference types already present in asprin’s library. But independently of that, it is interesting to see that, given the preference semantics defined in their paper, the definition and the implementation of the new preference type in asprin is quite direct. This line of work has been continued in [75], where a similar approach is applied to reasoning about neural networks. A finitely multi-valued extension of the description logic $A\mathcal{LC}$ is used to formalize neural networks, and reasoning in the Boolean fragment of that extension is reduced to reasoning in asprin. As before, the authors have defined and implemented their own preference types, and acted as preference engineers.

The proposal of [76] is grounded also in an extension of description logics for nonmonotonic reasoning. In this case, the task is to represent and reason about context dependent language in the framework of Contextualized Knowledge Repositories (CKR). The authors show how to model an extension of CKR in weighted LARS [77], and then show how to represent a fragment of that language in asprin, using the preference types poset, pareto and lexico.

In another application to description logics, in [78] asprin has been used to compute justifications in a version of the description logic $SROIQ$ under fixed domain semantics, where the cardinality of the domain is known a priori. Specifically, that computational task has been reduced to reasoning in asprin with the preference type subset.

On a completely different application, [79] presents an approach to formalize natural language sentences in logic in the presence of inconsistency, with an application to job puzzles. For this, the authors introduce a nonmonotonic semantics for the paraconsistent logic of Annotated Predicate Calculus. Then, they show how to reason in that logic using asprin with the preference types subset and lexico.

In the domain of sequential pattern mining, [80] describes an application of asprin to the task of identifying frequent subsequences in sequence databases. This work tries to identify frequent patterns that are of special interest to the user, and to this aim it extends with preferences the usual setting in sequential pattern mining. The implementation uses the preference types more(cardinality) and pareto,
as well as a new preference type that represents a preference for the highest value of an arithmetical division. This new preference type could be represented by \textit{more(weight)}, but the new preference type, specific for that mathematical operation, provides a more efficient encoding. This is another example of how the flexibility of \textit{asprin} can be used to improve its own efficiency.

Last but not least, the works of [81, 82] present ASP-based approaches to argumentative reasoning. The first addresses the task of reasoning about the logic programming fragment of assumption-based argumentation frameworks extended with preferences (ABA+), and reduces reasoning in that framework to reasoning in \textit{asprin} with \textit{superset} preferences. The second focuses on representing and reasoning about the rule-based argumentation framework of ASPIC+, using \textit{asprin} to represent the preferred semantics, again with the preference type \textit{superset}.

11. Conclusion

In this work, we presented \textit{asprin}, a framework for representing and reasoning with preferences in ASP. We started with the definition of a general language to specify preferences. We then showed how preferences can be implemented by preference programs, and introduced different reasoning methods that rely on these programs. We complemented this with an extensive complexity analysis of the reasoning tasks tackled by our system. We also introduced the first-order modeling language of \textit{asprin}, and used it to implement various approaches to preferences from the literature. We gave an overview of the features of the system, and evaluated its performance experimentally. In this evaluation, we concluded that \textit{asprin} in general compares well to dedicated implementations for preferences, although there was a gap with respect to \textit{clingo} on cardinality and weight optimization. Fortunately, this gap has been closed by the new solving methods presented in [45]. Moreover, we found that domain-specific heuristics improve the performance of \textit{asprin}, and that the usage of composite preferences does not significantly affect the performance of the system. Finally, we discussed our approach and put it in connection with related work.

For future work, we plan to integrate further approaches into \textit{asprin}’s library, such as preference trees [65] and the general preference languages in [56, 19] discussed above. A major step forward would be the extension of \textit{asprin} to multi-shot solving and theory solving, paralleling the extension of \textit{clingo} in the same directions [9, 83]. Additionally, we would like to study the application of \textit{asprin}’s framework to the problem of learning user preferences. From a more theoretical
perspective, we plan to apply the theoretical results from [57] to study strong equivalence in asprin.

**Declaration of competing interests**

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Acknowledgements**

The first author acknowledges support from ScaDS.AI, one of the AI competence centers in Germany. This work was partially funded by DFG grant SCHA 550/15.

**12. References**


[19] M. Bienvenu, J. Lang, N. Wilson, From preference logics to preference languages, and back, in: Lin and Sattler [85].


Appendix A. Proofs of results

Appendix A.1. Proofs of Section 4 (Handling Preferences)

PROOF. (Proposition 1)

The proposition follows from the following equivalences: \((X,Y)\) belongs to \(\succ_{s}\) iff \(X \succ_{s} Y\) (by a simple rewriting) iff \(X,Y \subseteq A\) and the program \(P_{s} \cup H(X) \cup H(Y)'\) is satisfiable (by Definition 1) iff \((X,Y)\) belongs to \(\{(X,Y) \mid X,Y \subseteq A, P_{s} \cup H(X) \cup H(Y)'\}\) is satisfiable} (by a simple rewriting).

PROOF. (Proposition 2)

We prove the proposition by showing that the following statements are equivalent:

1. \((X,Y) \in \succ_{s}\).
2. \( X, Y \subseteq \mathcal{A} \) and \( P_s \cup H(X) \cup H'(Y) \) is satisfiable.

3. \( X, Y \subseteq \mathcal{A} \) and there is a stable model \( Z \) of \( P_s \cup H(X) \cup H'(Y) \) such that 
   \[ X = \{ a \mid \text{holds}(a) \in Z \} \quad \text{and} \quad Y = \{ a \mid \text{holds}'(a) \in Z \}. \]

4. There is a stable model \( Z \) of \( P_s \cup \mathcal{C}(\mathcal{A}) \cup \mathcal{C}'(\mathcal{A}) \) such that 
   \[ X = \{ a \mid \text{holds}(a) \in Z \} \quad \text{and} \quad Y = \{ a \mid \text{holds}'(a) \in Z \}. \]

5. \( (X, Y) \in (10) \).

The equivalence between statements 1 and 2 follows from Definition 1, and the equivalence between statements 4 and 5 is trivial. Next, we prove the other two equivalences.

**Statement 2 is equivalent to statement 3:** Statement 2 follows trivially from statement 3. In the other direction, if statement 2 holds, then

\[ X, Y \subseteq \mathcal{A} \] and there is some stable model \( Z \) of \( P_s \cup H(X) \cup H'(Y) \). (A.1)

From (A.1) it follows that \( H(X) \subseteq Z \) and, given that atoms in \( H(X) \) have the form \( \text{holds}(a) \) for some \( a \in X \), we can conclude that

\[ H(X) \subseteq \{ \text{holds}(a) \mid \text{holds}(a) \in Z \}. \] (A.2)

On the other hand, given that in the heads of \( P_s \) there are no atoms of the predicate \( \text{holds} \), we have that the only rules whose heads contain atoms of the predicate \( \text{holds} \) are \( H(X) \), and therefore

\[ H(X) \supseteq \{ \text{holds}(a) \mid \text{holds}(a) \in Z \} \]

that, together with (A.2), implies that

\[ H(X) = \{ \text{holds}(a) \mid \text{holds}(a) \in Z \}. \]

By a simple rewriting, we obtain that

\[ X = \{ a \mid \text{holds}(a) \in Z \}. \] (A.3)

We can reason similarly about \( H(Y)' \) to obtain that

\[ Y = \{ a \mid \text{holds}'(a) \in Z \}. \] (A.4)

Then, statement 3 follows from (A.1), (A.3), and (A.4).

**Statement 3 is equivalent to statement 4:** Let us consider the program

\[ P_s \cup \mathcal{C}(\mathcal{A}) \cup \mathcal{C}'(\mathcal{A}) \] (A.5)

72
of statement 4. Given that in the heads of \( P_s \) there are no atoms of the predicates \( \text{holds} \) or \( \text{holds}' \), by the Splitting Set Theorem [91] it follows that

\[
Z \text{ is a stable model of (A.5) iff } Z \text{ is a stable model of } P_s \cup Z_1 \text{ for some stable model } Z_1 \text{ of } C(\mathcal{A}) \cup C'(\mathcal{A}). \tag{A.6}
\]

It is easy to see that the stable models of \( C(\mathcal{A}) \cup C'(\mathcal{A}) \) are the subsets of the set \( H(\mathcal{A}) \cup H(\mathcal{A}') \), or in other words, the sets \( H(V) \cup H(W)' \) such that \( V, W \subseteq \mathcal{A} \). Hence, from (A.6) it follows that

\[
Z \text{ is a stable model of (A.5) iff } Z \text{ is a stable model of } P_s \cup H(V) \cup H(W)'
\text{ for some sets } V, W \subseteq \mathcal{A}. \tag{A.7}
\]

Applying the same reasoning that we used to obtain (A.3), we can infer that \( V = \{ a \mid \text{holds}(a) \in Z \} \) and \( W = \{ a \mid \text{holds}'(a) \in Z \} \). Then by (A.7) it follows that

\[
Z \text{ is a stable model of (A.5) iff } Z \text{ is a stable model of } P_s \cup H(V) \cup H(W)'
\text{ for some sets } V, W \subseteq \mathcal{A} \text{ such that } V = \{ a \mid \text{holds}(a) \in Z \} \text{ and } W = \{ a \mid \\
\text{holds}'(a) \in Z \}.
\]

This implies that statement 4 is equivalent to the following statement:

There is a stable model \( Z \) of \( P_s \cup H(V) \cup H(W)' \) for some sets \( V, W \subseteq \mathcal{A} \) such that \( V = \{ a \mid \text{holds}(a) \in Z \} \), \( W = \{ a \mid \text{holds}'(a) \in Z \} \), \( X = \{ a \mid \\
\text{holds}(a) \in Z \} \) and \( Y = \{ a \mid \text{holds}'(a) \in Z \} \). \tag{A.8}

Finally, using the fact that \( V = X \) and \( W = Y \), this statement can be rewritten as statement 3.

**PROOF.** (Proposition 3)

Let \( X \subseteq \mathcal{A} \) be a set of atoms, and let \( Q \) denote the program \( P \cup P_s \cup R(\mathcal{A}) \cup H(X)' \). We start by proving the following statement:

\[
M \text{ is a stable model of } Q \text{ iff } M = M_1 \cup M_2 \text{ for some sets of atoms } M_1 \text{ and } M_2 \text{ such that } M_1 \text{ is a stable model of } P, \text{ and } M_2 \text{ is a stable model of } P_s \cup H(M_1) \cup H(X)'.
\tag{A.9}
\]

Recall that by the form of the programs \( P \) and \( P_s \) we have that the sets of atoms occurring in \( P \) and \( P_s \) are disjoint, no atoms over \text{holds} or \text{holds}' occur in \( P \), and no such atoms occur in the heads of \( P_s \). Then, by the Splitting Set Theorem, we can split \( Q \) in three parts: \( P \), \( R(\mathcal{A}) \), \( \text{and } P_s \cup H(X)' \); and it follows that
\( M \) is a stable model of \( Q \) iff there are some sets of atoms \( M_1 \) and \( M_3 \) such that \( M_1 \) is a stable model of \( P \), \( M_3 \) is a stable model of \( M_1 \cup R(A) \), and \( M \) is a stable model of \( M_3 \cup \mathcal{P}_s \cup H(X)^\prime \).

It is easy to see that the program \( M_1 \cup R(A) \) has a unique stable model \( M_1 \cup H(M_1) \). Then, the set \( M_3 \) is exactly that set of atoms, and we have that

\[ M \text{ is a stable model of } Q \text{ iff there is a set of atoms } M_1 \text{ such that } M_1 \text{ is a stable model of } P, \text{ and } M \text{ is a stable model of } M_1 \cup H(M_1) \cup \mathcal{P}_s \cup H(X)^\prime. \]  
(A.10)

Since \( M_1 \) is a stable model of \( P \), by the form of \( P \) the atoms in \( M_1 \) do not occur in the rules \( H(M_1) \cup \mathcal{P}_s \cup H(X)^\prime \), and therefore \( M \) has the form \( M_1 \cup M_2 \) for some stable model \( M_2 \) of \( H(M_1) \cup \mathcal{P}_s \cup H(X)^\prime \). This allows us to rewrite (A.10) as (A.9), and prove the latter.

Let \( X \) be a stable model of \( P \). Then it holds that \( X \subseteq A \). We prove part 1 by showing that

\( X \) is not \( \succeq_s \)-preferred iff the program \( Q \) is satisfiable.

We do this by showing that the following statements are equivalent:

1. \( X \) is not \( \succeq_s \)-preferred.
2. There is some stable model \( Y \) of \( P \) such that \( Y \succ_s X \).
3. There is some stable model \( Y \) of \( P \) such that \( \mathcal{P}_s \cup H(Y) \cup H(X)^\prime \) is satisfiable.
4. There are some sets of atoms \( Y \) and \( Z \) such that \( Y \) is a stable model of \( P \), and \( Z \) is a stable model of \( \mathcal{P}_s \cup H(Y) \cup H(X)^\prime \).
5. The program \( Q \) has some stable model \( M \).
6. The program \( Q \) is satisfiable.

The equivalence between 1 and 2 holds by definition of \( \succeq_s \)-preferred, the equivalence between 2 and 3 holds because \( \mathcal{P}_s \) is a preference program for \( s \), the equivalence between 3 and 4 holds trivially, the equivalence between 4 and 5 holds by (A.9) given that \( X \subseteq A \), replacing \( Y \) by \( M_1 \) and \( Z \) by \( M_2 \), and the equivalence between 5 and 6 holds trivially.

For part 2, if \( Y \) is a stable model of \( Q \) for some \( X \subseteq A \), by (A.9) it follows that

\[ Y = M_1 \cup M_2 \text{ for some sets of atoms } M_1 \text{ and } M_2 \text{ such that } M_1 \text{ is a stable model of } P, \text{ and } M_2 \text{ is a stable model of } \mathcal{P}_s \cup H(M_1) \cup H(X)^\prime. \]  
(A.11)
Given that the atoms of \( A \) do not occur in the program \( P_s \cup H(M_1) \cup H(X)' \), the set \( M_2 \cap A \) is empty, and since we have that \( Y = M_1 \cup M_2 \) by (A.11), we can conclude that

\[
Y \cap A = M_1. \tag{A.12}
\]

On the other hand, from (A.11) and the fact that \( P_s \) is a preference program for \( s \), it follows that

\[
M_1 \succ^s X. \tag{A.13}
\]

Putting all together, we can prove that \( Y \cap A \) is a stable model of \( P \) by (A.11) and (A.12), and that \( Y \cap A \succ^s X \) by (A.12) and (A.13).

**Proof.** (Proposition 4, sketch)

Let \( X, Y \subseteq A \), then by \( Q(X, Y) \) we denote the logic program

\[
E_{\text{subset}} \cup F_s \cup G \cup H(X) \cup H(Y)'.
\]

The set \( G \) consists of the set of rules defining \textit{holds} and \textit{holds}' atoms for the boolean formulas in \( E \), that we denote by \( G' \), together with the rule (9). To simplify the proof, we assume that \( G' \) is a set of facts. By definition 1, we have to prove that

for all sets \( X, Y \subseteq A \) the program \( Q(X, Y) \) is satisfiable iff \( X \succ^s Y \). \tag{A.14}

Take any sets \( X, Y \subseteq A \), and let \( Q'(X, Y) \) be the logic program

\[
E_{\text{subset}} \cup F_s \cup G' \cup H(X) \cup H(Y)'.
\]

Given that (9) is an integrity constraint,

the stable models of \( Q(X, Y) \) are the stable models of \( Q'(X, Y) \) that do not violate the integrity constraint (9). \tag{A.15}

Observe that the program \( Q'(X, Y) \) consists of a set of facts along with the rule \( E_{\text{subset}} \), that defines the predicate \textit{better}, that does not occur in any fact. Then, it is easy to see that

the program \( Q'(X, Y) \) has a unique stable model \( M \) that has the form \( N \cup O \) where \( N \) is the set of atoms \( F_s \cup G' \cup H(X) \cup H(Y)' \) and \( O \) is a set of atoms over \textit{better}. \tag{A.16}
More precisely, below we prove that

\[ O \text{ is the set of atoms } \{ \text{better}(s) \} \text{ if } X \succ_s Y, \text{ and it is } \emptyset \text{ otherwise. } \quad (A.17) \]

On the other hand, given that the fact \textit{optimize}(s) \leftarrow \text{ belongs to } F_s, \text{ it follows that } \textit{optimize}(s) \text{ belongs to } M. \text{ Since predicate optimize does not occur in any other rule head, we have that no other instances of predicate \textit{optimize} belong to } M. \text{ Therefore, } M \text{ violates the constraint (9) iff } \text{better}(s) \notin M. \text{ Moreover, since } M \text{ has the form } N \cup O \text{ and } \text{better}(s) \notin N, \text{ we can conclude that}

\[ M \text{ violates the constraint (9) iff } \text{better}(s) \notin O. \]

By this, (A.15), (A.16) and (A.17), it follows that the program \( Q(X,Y) \) has a unique stable model \( M \) that has the form \( N \cup \{ \text{better}(s) \} \) if \( X \succ_s Y \), and it has no stable models otherwise. This implies (A.14) and completes the main part of the proof.

\textbf{Proof of (A.17).} Given the form of \( E_{\text{subset}} \), \( O \) is the set of atoms of the form \textit{better}(p) such that there are some terms \( x, t_1, t_2 \) and \( t_3 \) satisfying these conditions:

1. \( \text{preference}(p, \text{subset}) \in N; \)
2. \( \text{holds}(x) \notin N, \text{ holds}'(x) \in N, \text{ and } \text{preference}(p, t_1, t_2, \text{for}(x), t_3) \in N; \)
3. for all terms \( y \), if \( \text{holds}(y) \in N \) and there are some terms \( t_4, t_5 \) and \( t_6 \) such that \( \text{preference}(p, t_4, t_5, \text{for}(y), t_6) \in N, \text{ then holds}'(y) \in N. \)

In what follows, we abuse notation and identify the term representation \( x \) of a boolean formula with the boolean formula itself. Observe that, given the form of \( N \), the following equivalences hold:

- \( \text{preference}(p, \text{subset}) \in N \text{ iff } s = p; \)
- there are some terms \( t_1, t_2 \) and \( t_3 \) such that \( \text{preference}(p, t_1, t_2, \text{for}(x), t_3) \in N \text{ iff } x \in E; \)
- for every \( x \in E \) we have that \( \text{holds}(x) \in N \text{ iff } X \models x, \text{ and } \text{holds}'(x) \in N \text{ iff } Y \models x. \)

They allow us to reformulate the previous conditions as follows:

1. \( s = p; \)
2. \( X \not\models x, Y \models x, \text{ and } x \in E; \)
3. for all terms \( y \), if \( X \models y \) and \( y \in E \), then \( Y \models y. \)
Then, it follows that \( O \) is the set of atoms \( \{\text{better}(s)\} \) if:

1. there is some \( x \in E \) such that \( X \not\models x \) and \( Y \models x \), and
2. for all \( y \in E \), if \( X \models y \) then \( Y \models y \);

and it is \( \emptyset \) otherwise. The first condition can be rewritten as

\[
\{ x \in E \mid X \models x \} \not\supseteq \{ x \in E \mid Y \models x \},
\]

and the second one can be rewritten as

\[
\{ x \in E \mid X \models x \} \subseteq \{ x \in E \mid Y \models x \}.
\]

Together, they can be rewritten as

\[
\{ x \in E \mid X \models x \} \subset \{ x \in E \mid Y \models x \}.
\]

By definition of the subset preference type, this is equal to

\[
(X, Y) \in \text{subset}(E),
\]

which, given the form of \( s \), is equal to

\[
X \succ_s Y.
\]

Hence, (A.17) follows: \( O \) is the set of atoms \( \{\text{better}(s)\} \) if \( X \succ_s Y \), and it is \( \emptyset \) otherwise.

**Proof.** (Proposition 5, sketch)

The proof is essentially the same as the proof of Proposition 4, except for the proof of the statement (A.17):

\[
O \text{ is the set of atoms } \{\text{better}(s)\} \text{ if } X \succ_s Y, \text{ and it is } \emptyset \text{ otherwise;}
\]

that we prove next using the notation introduced in Proposition 4.

Given the form of \( E_{\text{more(\text{weight})}} \), \( O \) is the set of atoms of the form \( \text{better}(p) \) satisfying these conditions:

1. \( \text{preference}(p, \text{more}(\text{weight})) \in N \);
2. the sum of the following integers is greater than 0:
   - the integer terms \( w \) for every pair of terms \((w, x)\) such that \( \text{holds}(x) \in N \) and there are some terms \( t_1 \) and \( t_2 \) such that \( \text{preference}(p, t_1, t_2, \text{for}(x), w) \in N \); and
• the integer terms $-w$ for every pair of terms $(w, x)$ such that $\text{holds}'(x) \in N$ and there are some terms $t_1$ and $t_2$ such that preference$(p, t_1, t_2, \text{for}(x), w) \in N$.

In what follows, we abuse notation and identify the term representation $x$ of a boolean formula with the boolean formula itself. Observe that, given the form of $N$, the following equivalences hold:

- preference$(p, \text{more}(\text{weight})) \in N$ iff $s = p$;
- there are terms $t_1$ and $t_2$ such that preference$(p, t_1, t_2, \text{for}(x), w) \in N$ iff $w : x \in E$;
- for every $w : x \in E$ we have that holds$(x) \in N$ iff $X \models x$, and holds$(x) \in N$ iff $Y \models x$.

They allow us to reformulate the previous conditions as follows:

1. $s = p$;
2. the sum of the following integers is greater than 0:
   • the integers $w$ for every $w : x \in E$ such that $X \models x$, and
   • the integers $-w$ for every $w : x \in E$ such that $Y \models x$.

Then, it follows that $O$ is the set of atoms $\{\text{better}(s)\}$ if the sum of the following integers is greater than 0:

- the integers $w$ for every $w : x \in E$ such that $X \models x$, and
- the integers $-w$ for every $w : x \in E$ such that $Y \models x$;

and it is $\emptyset$ otherwise. The sum can be formally written as:

$$\sum_{(w,x) \in E, X \models x} w + \sum_{(w,x) \in E, Y \models x} -w$$

and it is greater than 0 iff

$$\sum_{(w,x) \in E, X \models x} w > \sum_{(w,x) \in E, Y \models x} w.$$

By definition of the more$(\text{weight})$ preference type, this is equal to

$$(X, Y) \in \text{more}(\text{weight})(E),$$
which, given the form of \( s \), is equal to
\[
X \succ_s Y.
\]
Hence, (A.17) follows: \( O \) is the set of atoms \( \{\text{better}(s)\} \) if \( X \succ_s Y \), and it is \( \emptyset \) otherwise.

**Appendix A.2. Proofs of Section 5.1 (Computing a preferred model)**

We first prove Lemmas 1 and 2. They are used to prove Proposition 6, that in turn is used to prove Theorem 1.

**Lemma 1.** Let \((Y_0, Y_1, \ldots)\) be a trace of Algorithm 1 for a finite program \( P \) and a preference program \( P_s \) for preference statement \( s \). For every \( Y_i \) of the trace, it holds that

1. either \( Y_i \) is \( \bot \) and it is the last element of the trace;
2. or \( Y_i \) is a stable model of \( P \) such that it is not the last element of the trace and \( Y_i \succ_s Y_{i-1} \) if \( i > 0 \).

**Proof.** Take any \( Y_i \), and observe that it has been computed either at Line 1 or Line 5 of Algorithm 1. We consider four cases, depending on whether \( Y_i \) is \( \bot \) or not, and on whether \( Y_i \) was computed at Line 1 or Line 5. We prove the lemma by showing that for all cases either item 1 or item 2 holds.

If \( Y_i \) is \( \bot \) and it has been computed at Line 1, then the algorithm returns at Line 2, \( Y_i \) is the last element of the trace, and the first item of the lemma holds.

If \( Y_i \) is \( \bot \) and it has been computed at Line 5, then the algorithm exists the loop at Line 6. Hence, \( Y_i \) is the last element of the trace, and the first item of the lemma holds.

If \( Y_i \) is not \( \bot \) and it has been computed at Line 1, then it is a stable model of \( P \). Moreover, in this case the condition of Line 2 is not true, hence the algorithm enters the loop and in Line 5 defines \( Y_{i+1} \). Therefore, \( Y_i \) is not the last element of the trace. Given all this and the fact that in this case \( i = 0 \), the second item of the lemma holds.

If \( Y_i \) is not \( \bot \) and it has been computed at Line 5, then we have that \( i > 0 \). Moreover, in this case the condition of Line 6 is not true, hence the algorithm re-enters the loop and in Line 5 defines \( Y_{i+1} \). Therefore, \( Y_i \) is not the last element of the trace. Given this, the second item of the lemma follows from the following statement, that we prove by induction over \( j \):
for every $Y_j$ of the trace such that $Y_j$ is not $\bot$ and $j > 0$, it holds that $Y_j$ and $Y_{j-1}$ are stable models of $P$ such that $Y_j \succ_s Y_{j-1}$. (A.18)

In the basic case we consider that $j = 1$. Then, $Y_{j-1} = Y_0$ has been computed in Line 1 and is a stable model of $P$. After Line 1, the condition of Line 2 fails, and $X$ is assigned the value $Y_{j-1}$ in Line 4. After executing Line 5, we have that $Y_j$ is

$$\text{solve}(P \cup P_s \cup R(A) \cup H(Y_{j-1}')) \cap A.$$  

Given that $Y_j$ is not $\bot$, it must have the form $M \cap A$ for some stable model $M$ of

$$P \cup P_s \cup R(A) \cup H(Y_{j-1}')'.$$

Then, by the second item of Proposition 3, we obtain that $Y_j$ is a stable model of $P$ and $Y_j \succ_s Y_{j-1}$, and this completes the proof of the basic case.

In the induction case, we consider that $j > 1$ and assume that (A.18) holds for $j - 1$. Hence, $Y_{j-1}$ is a stable model of $P$ computed in Line 5. After that, the condition of Line 6 fails, and $X$ is assigned the value $Y_{j-1}$ in Line 4. Then, after executing Line 5, $Y_j$ has the same form as in the basic case, and the rest of the proof is the same as before.

The following Lemma proves items 2 to 5 of Proposition 6, using the previous Lemma 1.

**Lemma 2.** Let $(Y_0, Y_1, \ldots)$ be a trace of Algorithm 1 for a finite program $P$ and a preference program $P_s$ for preference statement $s$. Then, the trace is finite and has the form $(Y_0, \ldots, Y_n)$ for some integer $n \geq 0$, and

1. $Y_i$ is a stable model of $P$ for $0 \leq i < n$
2. $Y_i \succ_s Y_{i-1}$ for $0 < i < n$
3. $Y_n = \bot$
4. $Y_{n-1}$ is a $\succeq_s$-preferred stable model of $P$, if $n > 0$

**PROOF.** First, we prove that the trace is finite. For this, we show that

every pair of elements $Y_i$ and $Y_j$ of the trace such that $i < j$ are different. (A.19)
Assume, for the sake of contradiction, that for some $i < j$, the sets $Y_i$ and $Y_j$ are the same. By Lemma 1, $Y_i$ and $Y_j$ must be either $\bot$, or a stable model $M$ of $P$. The first case is not possible because, by Lemma 1, if $Y_i = \bot$ then it must be the last element of the trace. In the second case, consider the subtrace $(Y_i, Y_{i+1}, \ldots, Y_j)$. By Lemma 1, all the $Y_k$ for $i < k \leq j$ must be stable models of $P$ such that $Y_k \succ_s Y_{k-1}$. Hence, by transitivity of $\succ_s$, it follows that $Y_j \succ_s Y_i$ and, by our assumption, we have that $M \succ_s M$, which contradicts the fact that $\succ_s$ is a strict partial order. Both cases lead to a contradiction, therefore the assumption does not hold and (A.19) follows.

By Lemma 1, every element $Y_i$ of the trace must be either $\bot$, or a stable model $M$ of $P$. Given that the program $P$ is finite, the set of its stable models is also finite. Then, by (A.19) it follows that the trace is finite, and therefore the trace must have the form $(Y_0, \ldots, Y_n)$ for some integer $n \geq 0$. In addition, from this and Lemma 1 we can conclude items 1, 2 and 3 of the proposition.

For item 4, if $n > 0$, then $Y_{n-1}$ is a stable model that has been computed either at Line 1 or Line 5 of Algorithm 1, and in both cases the assignment in Line 4 is executed afterwards. Then, after executing Line 5, $Y_n$ is

$$\text{solve}(P \cup P_s \cup R(\mathcal{A}) \cup H(Y_{n-1}')) \cap \mathcal{A}.$$ 

By item 3 we have that $Y_n$ is $\bot$, therefore the program $P \cup P_s \cup R(\mathcal{A}) \cup H(Y_{n-1}')$ is unsatisfiable, and item 4 follows by the first item of Proposition 3.

**PROOF.** (Proposition 6)

From the previous Lemma 2, we have that the trace is finite and has the form $(Y_0, \ldots, Y_n)$ for some integer $n \geq 0$, and we also have items 2 to 5 of the proposition.

We prove item 1 of the proposition as follows. If $P$ is satisfiable, then $Y_0$ is a stable model of $P$, and by item 4 it must be the case that $n > 0$. Otherwise, if $P$ is not satisfiable, then $Y_0$ is $\bot$, and by Lemma 1 the element $Y_0$ is the last of the trace, that must have the form $(Y_n)$ with $n = 0$.

For item 6, if $n = 0$ then by item 4 we have that $Y_0 = \bot$, hence $Y$ has the value $\bot$ in Line 1, and the algorithm returns $\bot = Y_0$ in Line 2. If $n > 0$, then we are in the same situation as for item 5, that was proved as item 4 of Lemma 2: in Line 4 the variable $X$ is assigned the value $Y_{n-1}$, in Line 5 the variable $Y$ is assigned the value $\bot$, then the condition to stop the loop is true and $X = Y_{n-1}$ is returned in Line 7.
PROOF. (Theorem 1)

The theorem follows applying the items of Proposition 6. If \( P \) is satisfiable, then by item 1 it follows that \( n > 0 \) and by item 6 it follows that the algorithm returns \( Y_{n-1} \), that is a \( \geq \)s-preferred stable model by item 5. Otherwise, if \( P \) is not satisfiable, then by item 1 it follows that \( n = 0 \) and by item 6 the algorithm returns \( Y_0 \), that is equal to \( \bot \) by item 4.

Appendix A.3. Proofs of Section 5.2 (Computing all preferred models)

PROOF. (Proposition 7)

Let \( C \) be the integrity constraints of \( P \), and \( Q \) be the rest of the rules, such that \( P = Q \cup C \). It is easy to see that

\[
N \text{ is a stable model of } P \text{ iff } N \text{ is a stable model of } Q \text{ that satisfies all constraints in } C. \tag{A.20}
\]

Using this notation, we can represent \( \overline{P} \) as \( Q \cup \overline{C} \), where \( \overline{C} \) is the set of rules \( \{ u \leftarrow \text{body}(r) \mid r \in C \} \cup \{ \leftarrow \neg u \} \). Observe that \( Q \) is a stratified program without integrity constraints. Hence,

\[
Q \text{ has a unique stable model, that we denote by } M. \tag{A.21}
\]

Consider the next statement, that we will use afterwards:

\[
M \text{ satisfies } \text{body}(r) \text{ for some } r \in C. \tag{A.22}
\]

The proposition is implied by the following statements, that we prove below:

\[
P \text{ is satisfiable iff (A.22) does not hold,} \tag{A.23}
\]

and

\[
\overline{P} \text{ is satisfiable iff (A.22) holds.} \tag{A.24}
\]

**Proof of (A.23).** \( P \) is satisfiable iff there is some stable model \( N \) of \( P \) iff there is some stable model \( N \) of \( Q \) that satisfies all constraints in \( C \) (by (A.20)) iff \( M \) satisfies all constraints in \( C \) (by (A.21)) iff (A.22) does not hold.

**Proof of (A.24).** We show that the following statements are equivalent:

1. \( \overline{P} \) is satisfiable.
2. There is some stable model \( N \) of \( \overline{P} \).
3. There is some stable model \( N \) of \( O \cup \overline{C} \) for some stable model \( O \) of \( Q \).
4. There is some stable model \( N \) of \( M \cup \overline{C} \).
5. \( M \cup \{ u \} \) is a stable model of \( M \cup \overline{C} \).
6. \( M \cup \{ u \} \) satisfies \( \text{body}(r) \) for some \( r \in C \).
7. (A.22) holds.

The equivalence between 1 and 2 is trivial, the equivalence between 2 and 3 follows by the Splitting Set Theorem, and the equivalence between 3 and 4 follows by (A.21). Statement 4 implies 5 because, on the one side, \( N \) must contain all atoms asserted by the facts in \( M \), as well as the atom \( u \) to satisfy the constraint \( \leftarrow \neg u \), and on the other side, \( N \) cannot contain more atoms because no other atoms occur in the heads of \( M \cup \overline{C} \). In the other direction, statement 5 implies 4 trivially. Statement 5 implies 6 because the atom \( u \) in \( M \cup \{ u \} \) must be supported by some rule of \( M \cup \overline{C} \), and the only rules that could support \( u \) are those in \( \{ u \leftarrow \text{body}(r) \mid r \in C \} \). In the other direction, statement 6 implies 5 because \( M \cup \{ u \} \) satisfies all rules in \( M \cup \overline{C} \), the atoms in \( M \) are justified by the corresponding facts, and the atom \( u \) is justified by some rule \( r \) in \( \{ u \leftarrow \text{body}(r) \mid r \in C \} \) such that \( M \cup \{ u \} \) satisfies \( \text{body}(r) \). Finally, statement 6 is equivalent to statement 7 because the atom \( u \) does not occur in any \( \text{body}(r) \) for \( r \in C \).

**Proof.** (Proposition 8)

We prove the proposition by showing that, for all \( X, Y \subseteq A \), the following statements are equivalent:

1. \( X \not\models_s Y \).
2. \( P_s \cup H(X) \cup H(Y)' \) is not satisfiable.
3. \( P_s \cup H(X) \cup H(Y)' \) is satisfiable.
4. \( P_s \cup H(X) \cup H(Y)' \) is satisfiable.

Statement 1 is equivalent to 2 by Definition 1. Observe that \( P_s \cup H(X) \cup H(Y)' \) is stratified, given that \( P_s \) is stratified and the other rules are facts. Then, statement 2 is equivalent to statement 3 by Proposition 7. Finally, the equivalence between statements 3 and 4 holds because, by definition, the programs \( P_s \cup H(X) \cup H(Y)' \) and \( P_s \cup H(X) \cup H(Y)' \) are the same.

**Proof.** (Proposition 9)

The proof is similar to the proof of part 2 of Proposition 3.

Let \( X \subseteq A \) be a set of atoms, and let \( Q \) denote the program \( (P \cup \overline{P_s} \cup H(X) \cup R'(A)) \). Then, the following statement holds:

\[
M \text{ is a stable model of } Q \iff M = M_1 \cup M_2 \text{ for some sets of atoms } M_1 \text{ and } M_2 \text{ such that } M_1 \text{ is a stable model of } P, \text{ and } M_2 \text{ is a stable model of } P_s \cup H(X) \cup H(M_1)' \text{.} \tag{A.25}
\]
The proof of this statement is the same as the proof of statement (A.9) of Proposition 3, changing what has to be changed.

To prove the proposition, let $Y$ be a stable model of $Q$ for some $X \subseteq A$. Hence, by (A.25) it follows that

$$Y = M_1 \cup M_2$$

for some sets of atoms $M_1$ and $M_2$ such that $M_1$ is a stable model of $P$, and $M_2$ is a stable model of $\overline{P_s} \cup H(X) \cup H(M_1)'$. (A.26)

Given that the atoms of $A$ do not occur in the program $\overline{P_s} \cup H(X) \cup H(M_1)'$, the set $M_2 \cap A$ is empty, and since we have that $Y = M_1 \cup M_2$ by (A.26), we can conclude that

$$Y \cap A = M_1.$$ (A.27)

On the other hand, from (A.26), the fact that $P_s$ is a stratified preference program for $s$, and Proposition 8, it follows that

$$X \not\triangleright_s M_1.$$ (A.28)

Putting all together, we can prove that $Y \cap A$ is a stable model of $P$ by (A.26) and (A.27), and that $X \not\triangleright_s Y \cap A$ by (A.27) and (A.28).

Proof. (Proposition 10)

We divide the proof in three steps.

Step 1. We prove that:

For every $i \in I$, it holds that $X_i \subseteq A$. (A.29)

Observe that the set $X$ is initialized to the empty set in Line 1. The elements $X_i$ for $i \in I$ are added in Line 9, and they always obtain their value from variable $Y$ in Line 6. In turn, variable $Y$ is assigned either in Line 3 or in Line 7. In both cases, either (i) the corresponding solving call returns some stable model $M$, or (ii) it returns $\perp$. If (i), then $Y$ is assigned the value $M \cap A$. Hence, $Y \subseteq A$ holds, and $X$ will be assigned in Line 6 some subset of $A$. If (ii), then $Y$ is assigned the value $\perp$. But in this case $X$ is never assigned that value, because the algorithm either returns in Line 4, or stops the innermost loop in Line 8. Then it is clear that $X$ only obtains the value of $Y$ in case (i). Hence, $X$ will always be assigned some subset of $A$, and (A.29) holds.

Step 2. Let $\{X_1, \ldots, X_n\}$ be the value of $X$ in Line 2, $I = \{1, \ldots, n\}$, and $Q$ denote the program $P \cup \bigcup_{X_i \in X} (N_{X_i} \cup (\overline{P_s} \cup H(X_i))' \cup R'(A)')$ used in Line 3. We prove that:
$M$ is a stable model of $Q$ iff $M = M_0 \cup M_1 \cup \ldots \cup M_n$ for some sets of atoms $M_0$, $M_1$, $\ldots$, $M_n$ such that $M_0$ is a stable model of $P$ that satisfies all constraints in $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$, and $M_i$ is a stable model of $(\overline{P}_s \cup H(X_i) \cup H(M_0)^i)$ for every $X_i \in \mathcal{X}$.

We start by applying the Splitting Set Theorem. First, we can split $Q$ in two parts:

- $Q_1 = P \cup \bigcup_{X_i \in \mathcal{X}} N_{X_i}$, and
- $Q_2 = \bigcup_{X_i \in \mathcal{X}} (\overline{P}_s \cup H(X_i))^i \cup R'(A)^i$.

This splitting is possible because the atoms that occur in $Q_1$ do not occur in the heads of the rules of $Q_2$, that consist of new atoms with some superscript $i \in I$.

The program $Q_2$ can in turn be split in two parts:

- $Q_3 = \bigcup_{X_i \in \mathcal{X}} R'(A)^i$, and
- $Q_4 = \bigcup_{X_i \in \mathcal{X}} (\overline{P}_s \cup H(X_i))^i$.

Like before, this splitting is possible because the atoms that occur in $Q_3$ do not occur in the heads of the rules of $Q_4$. To see this, observe that the atoms of $Q_3$ either belong to $A$ or have the form $\text{holds'}(a)^i$ for some $a \in A$ and $i \in I$, while the heads of $Q_4$ consist of new atoms with some superscript $i \in I$ that do not have the form $\text{holds'}(a)^i$ for some $a \in A$ and $i \in I$, given that no atoms over $\text{holds'}$ occur in the heads of $P_s \cup H(X_i)$. Then, applying the Splitting Set Theorem, we have that:

$M$ is a stable model of $Q$ iff there are some sets of atoms $M_0$ and $M'$ such that $M_0$ is a stable model of $Q_1$, $M'$ is a stable model of $M_0 \cup Q_3$, and $M$ is a stable model of $M' \cup Q_4$.

It is easy to see that the program $M_0 \cup Q_3$ has a unique stable model $M_0 \cup \bigcup_{X_i \in \mathcal{X}} H(M_0)^i$. Then, the set $M'$ is exactly that set of atoms, and we have that:

$M$ is a stable model of $Q$ iff there is some set of atoms $M_0$ such that $M_0$ is a stable model of $Q_1$, and $M$ is a stable model of $M_0 \cup \bigcup_{X_i \in \mathcal{X}} H(M_0)^i \cup Q_4$.

Since $M_0$ is a stable model of $Q_1$, by the form of $Q_1$, the atoms in $M_0$ do not occur in the rules $\bigcup_{X_i \in \mathcal{X}} H(M_0)^i \cup Q_4$. Hence, $M$ has the form $M_0 \cup M''$ for some stable model $M''$ of $\bigcup_{X_i \in \mathcal{X}} H(M_0)^i \cup Q_4$, and we have that:
$M$ is a stable model of $Q$ iff $M = M_0 \cup M''$ for some sets of atoms $M_0$ and $M''$ such that $M_0$ is a stable model of $Q_1$, and $M''$ is a stable model of $\bigcup_{X_i \in \mathcal{X}} H(M_0)^{\prime i} \cup Q_4$. \hspace{1cm} (A.31)

Observe now that the program $\bigcup_{X_i \in \mathcal{X}} H(M_0)^{\prime i} \cup Q_4$ is the same as $\bigcup_{X_i \in \mathcal{X}} (\overline{P_s} \cup H(X_i) \cup H(M_0)^{\prime i})$, and that for every $X_i, X_j$ such that $i \neq j$, the atoms occurring in $(\overline{P_s} \cup H(X_i) \cup H(M_0)^{\prime i})$ are disjoint from those occurring in $(\overline{P_s} \cup H(X_i) \cup H(M_0)^{\prime i})$. Hence, it follows that:

$M''$ is a stable model of $\bigcup_{X_i \in \mathcal{X}} H(M_0)^{\prime i} \cup Q_4$ iff $M'' = M_1 \cup \ldots \cup M_n$ for some sets of atoms $M_1, \ldots, M_n$ such that $M_i$ is a stable model of $(\overline{P_s} \cup H(X_i) \cup H(M_0)^{\prime i})$ for every $X_i \in \mathcal{X}$.

Using this, we can rewrite (A.31) as follows:

$M$ is a stable model of $Q$ iff $M = M_0 \cup M_1 \cup \ldots \cup M_n$ for some sets of atoms $M_0, M_1, \ldots, M_n$ such that $M_0$ is a stable model of $Q_1$, and $M_i$ is a stable model of $(\overline{P_s} \cup H(X_i) \cup H(M_0)^{\prime i})$ for every $X_i \in \mathcal{X}$. \hspace{1cm} (A.32)

By the form of $Q_1$, it is easy to see that:

$M_0$ is a stable model of $Q_1$ iff $M_0$ is a stable model of $P$ that satisfies all constraints in $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$.

Finally, from this and (A.32) we can conclude (A.30).

**Step 3.** Using (A.29), (A.30), and Proposition 9, we prove the following statements:

1. If the program $Q$ used in Line 3 is satisfiable, then item 1 of the proposition holds.
2. Otherwise, item 2 of the proposition holds.

This will prove that either item 1 or item 2 of the proposition holds, concluding the proof.

**Proof of statement 1.** If $Q$ is satisfiable, then there is some stable model $M$ of $Q$, and

$$Y = M \cap \mathcal{A}. \hspace{1cm} (A.33)$$

Given that $M$ is a stable model of $Q$, by (A.30) we have that:
$M = M_0 \cup M_1 \cup \ldots \cup M_n$ for some sets $M_0, M_1, \ldots, M_n$ such that $M_0$ is a stable model of $P$ that satisfies all constraints in $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$, and $M_i$ is a stable model of $(\overline{P}_s \cup H(X_i) \cup H(M_0'))^i$ for every $X_i \in \mathcal{X}$. \hfill (A.34)

All the atoms occurring in $P$ belong to $\mathcal{A}$. Then, given that $M_0$ is a stable model of $P$, we have that:

$$M_0 \cap \mathcal{A} = M_0.$$ \hfill (A.35)

On the other hand, no atoms occurring in the programs $(\overline{P}_s \cup H(X_i) \cup H(M_0'))^i$ for $X_i \in \mathcal{X}'$ belong to $\mathcal{A}$. Then, given that the $M_i$’s for $X_i \in \mathcal{X}'$ are stable models of those corresponding programs, we have that:

$$M_i \cap \mathcal{A} = \emptyset \text{ for } X_i \in \mathcal{X}.'\hfill (A.36)$$

Now, by (A.33) and (A.34) it follows that $Y = (M_0 \cap A) \cup (M_1 \cap A) \cup \ldots \cup (M_n \cap A)$, and then by (A.35) and (A.36) it follows that:

$$Y = M_0.$$ \hfill (A.37)

We are ready to show that $Y$ satisfies the conditions of item 1 of the proposition:

- By (A.34) and (A.37), $Y$ is a stable model of $P$.
- Moreover, $Y$ satisfies all constraints in $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$, and this implies that $Y \neq X_i$ for all $i \in I$.
- In addition, the programs $(\overline{P}_s \cup H(X_i) \cup H(Y)^i)\text{ for every } X_i \in \mathcal{X}'$ are satisfiable. This implies that the programs $\overline{P}_s \cup H(X_i) \cup H(Y)'i$ for every $i \in I$ are satisfiable, and then by (A.29) and Proposition 9 it follows that $X_i \not\models_s Y$ for all $i \in I$.

**Proof of statement 2.** If $Q$ is not satisfiable, then clearly $Y$ is $\bot$. We only have to show that in this case there is no stable model $Y$ of $P$ such that $Y \neq X_i$, and $X_i \not\models_s Y$ for all $i \in I$. We prove this by contradiction, assuming that there is some stable model $Y$ of $P$ that satisfies those properties. Given that $Y \neq X_i$ for all $i \in I$, it is easy to see that:

$Y$ is a stable model of $P$ that satisfies all constraints in $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$. \hfill (A.38)
Given that $X_i \not\models s Y$ for all $i \in I$, by Proposition 9 and (A.29), it follows that the programs $P_s \cup H(X_i) \cup H(Y)'$ for every $i \in I$ are satisfiable. This implies that the programs $(P_s \cup H(X_i) \cup H(Y)')'$ for every $X_i \in \mathcal{X}$ are satisfiable. Then, let $M_i$ be the stable model of $(P_s \cup H(X_i) \cup H(Y)')'$ for every $X_i \in \mathcal{X}$. Given this, (A.30), and (A.38), it follows that $Y \cup \bigcup_{X_i \in \mathcal{X}} M_i$ is a stable model of $Q$, which contradicts the fact that $Q$ is not satisfiable.

Next, we have Lemmas 3 and 4, that are very similar to the previous Lemmas 1 and 2. We use them afterwards to prove Propositions 11 and 12, that in turn are used to prove Theorem 2.

**Lemma 3.** Let $(Y_{i_0}, Y_{i_1}, \ldots)$ be an $i$-trace of Algorithm 2 for a finite program $P$ and a preference statement $s$. For every $Y_{i_j}$ of the trace, it holds that

1. either $Y_{i_j}$ is $\bot$ and it is the last element of the trace;
2. or $Y_{i_j}$ is a stable model of $P$ such that it is not the last element of the trace and $Y_{i_j} \succeq_s Y_{i_{j-1}}$ if $j > 0$.

The proof of this Lemma is almost the same as the proof of Lemma 1. The only differences are that we have to consider an $i$-trace $(Y_{i_0}, Y_{i_1}, \ldots)$ instead of a trace $(Y_0, Y_1, \ldots)$, Lines 3 to 8 of Algorithm 2 instead of Lines 1 to 6 of Algorithm 1, and that we have to use Proposition 10 to justify that Line 3 of Algorithm 2 assigns to $Y$ either $\bot$ or some stable model of $P$.

**Lemma 4.** Every $i$-trace of the algorithm is finite, has the form $(Y_{i_0}, \ldots, Y_{i_n})$ for some integer $n \geq 0$, and

1. $Y_{i_j}$ is a stable model of $P$ for $0 \leq j < n$
2. $Y_{i_j} \succeq_s Y_{i_{j-1}}$ for $0 < j < n$
3. $Y_{i_n} = \bot$
4. $Y_{i_{n-1}}$ is an $s$-preferred stable model of $P$, if $n > 0$

The proof of this Lemma is almost the same as the proof of Lemma 2, except for using Lemma 3 instead of Lemma 1, and for the differences already mentioned for Lemma 3.

We prove first Proposition 12 and then Proposition 11.

**Proof.** (Proposition 12)

The finiteness of the $i$-trace, as well as items 2 to 5 follow from Lemma 4.
We prove item 1 by showing that it holds for the two possible cases of Proposition 10. Note that the value \( Y \), mentioned in that proposition, is the same as the element \( Y_{i_0} \) of the corresponding \( i \)-trace.

If item 1 of Proposition 10 holds, then

\[
Y_{i_0} \text{ is a stable model of } P \text{ such that } Y_{i_0} \neq X_j, \text{ and } X_j \not
s Y_{i_0} \text{ for all } X_j \in \mathcal{X}_{i-1}.
\] (A.39)

By this and item 2 of Lemma 3 we have that \( Y_{i_0} \) is not the last element of the trace, and therefore

\[ n > 0. \] (A.40)

By item 2, 4 and the transitivity of \( \succ_s \), it follows that \( Y_{i_n} \) is a \( \succeq_s \)-preferred stable model of \( P \) such that \( Y_{i_n} \succ_s Y_{i_0} \). By (A.39), \( Y_{i_n} \) has to be different to all \( X_j \in \mathcal{X}_{i-1} \). Hence, \( \mathcal{X}_{i-1} \) does not contain all \( \succeq_s \)-preferred stable models of \( P \), and with (A.40) we can conclude that item 1 holds in this case.

On the other hand, if item 2 of Proposition 10 holds, then

\[ Y_{i_0} \text{ is } \bot, \] (A.41)

and

\[
\text{there is no stable model } Y \text{ of } P \text{ such that } Y \neq X_j, \text{ and } X_j \not
s Y \text{ for all } X_j \in \mathcal{X}_{i-1}.
\] (A.42)

By (A.41), item 2 and item 4 of this proposition, it follows that

\[ n = 0. \] (A.43)

We also have that (A.42) implies that

\( \mathcal{X}_{i-1} \) contains all \( \succeq_s \)-preferred stable models of \( P \). (A.44)

Then, by (A.43) and (A.44) we can conclude that item 1 holds in this case.

We prove now item 6. If \( n = 0 \), then by item 4 we have that \( Y_{i_0} \) is \( \bot \). This means that in Line 3 of Algorithm 2 the variable \( Y \) is assigned the value \( \bot \), and therefore the procedure returns in the next line, leaving \( X_i \) undefined. On the other case, if \( n > 0 \), then by item 4 we have that \( Y_{i_n} \) is \( \bot \). This means that in Line 7 of Algorithm 2 the variable \( Y \) is assigned the value \( \bot \), and therefore the loop finishes in the next line. In this iteration, in Line 6 the variable \( X \) is assigned
the value $Y_{i_{n-1}}$, and this is precisely the element that is added to $\mathcal{X}$ in Line 9 after finishing the loop. Hence, in this case $X_i$ is $Y_{i_{n-1}}$.

For item 7, the case where $j = 0$ and $j < n$ is implied by Proposition 10, since the value $Y$, mentioned in that proposition, is the same as $Y_{i_0}$. For the other case where $j$ is such that $0 < j < n$, by item 3 and the transitivity of $\succ_s$, we have that

$$Y_{ij} \succ_s Y_{i_0}. \quad (A.45)$$

We can then conclude item 7 for this case because, for all $X_k \in \mathcal{X}_{i-1}$, it cannot be the case that either $X_k = Y_{ij}$ or $X_k \succ_s Y_{ij}$ hold. Given (A.45) and the transitivity of $\succ_n$, any of them would imply that $X_k \succ_s Y_{i_0}$ holds, which would contradict the case where $j = 0$.

**PROOF.** (Proposition 11)

We prove first by contradiction that the trace $(X_i)_{i \in I}$ is finite. Assume that the trace is infinite. Given that every $i$-trace is finite (by Proposition 12), the trace $(X_i)_{i \in I}$ can only be infinite if Algorithm 2 never returns in Line 4, and this is only possible if in Line 3 the variable $Y$ is never assigned the value $\bot$. This means that for every $i$-trace of the form $(Y_{i_0}, \ldots, Y_{i_n})$ it is never the case that $Y_{i_0}$ is $\bot$. Then, by item 4 of Proposition 12, for every $i$-trace it must hold that $n > 0$. Moreover, by items 5, 6 and 7 of that proposition, it follows that $X_i$ should be a $\succeq_s$-preferred stable model of $P$ that is different than all previous $X_k \in \mathcal{X}_{i-1}$. Assuming that the trace $(X_i)_{i \in I}$ is infinite, this implies that the set of $\succeq_s$-preferred stable models of $P$ is infinite, which is false, given that $P$ is finite.

Now we prove items 1 and 3. For every $i \in I$, let us consider the $i$-trace $(Y_{i_0}, \ldots, Y_{i_n})$. The size $n$ cannot be 0, because in that case, by item 4 of Proposition 12, we would have that $Y_{i_0}$ is $\bot$. This would imply that Algorithm 2 returns in iteration $i$ in Line 4, and then no $X_i$ would be added in Line 9, implying that $i \notin I$. Hence, we have that $n > 0$, and in this case items 1 and 3 follow from items 6 and 5 of Proposition 12.

Next, we prove items 2 and 4. Given that the trace $(X_i)_{i \in I}$ and all $i$-traces are finite, Algorithm 2 must return at some point. This has to be done at iteration $i + 1$ in Line 4, after the variable $Y$ is assigned the value $\bot$ in Line 3. In this case, we have that $n = 0$ and $\mathcal{X} = \mathcal{X}_i$. This, together with item 1 of Proposition 12, implies that $\mathcal{X}$ contains all $\succeq_s$-preferred stable models of $P$. Item 4 follows from this, and item 2 also follows because every $X_i \in \mathcal{X}$ is the element $Y_{i_{n-1}}$ of the corresponding $i$-trace (by item 6 of Proposition 12) that must be different to all the other elements of the set $\mathcal{X}$ by item 7 of that proposition.
PROOF. (Theorem 2)
Given that the trace \( (X_i)_{i \in I} \) and all \( i \)-traces are finite (by Propositions 12 and 11), Algorithm 2 must return at some point. This has to be done at iteration \( i + 1 \), where it returns at Line 4 the variable \( \mathcal{X} \), that has the value \( \bigcup_{i \in I} X_i \). The theorem follows from this and items 1 and 2 of Proposition 11.

Appendix A.4. Proofs of Section 5.4 (Computational Complexity)
We prove one lemma for membership and another for hardness for every problem that we consider. We do it in this order: Optimality (Lemmas 5 and 6), Model Finding (Lemmas 7 and 8), Query (Lemmas 9 and 11), and Non Dominance (Lemmas 12 and 13). The membership proof for Optimality is used in the membership proofs for Model Finding and Query. This is the only dependency between the proofs. Additionally, before the hardness proof for Query, we include Lemma 10, that proves part of the hardness results for both Query and for Non Dominance. In the end, Theorem 3 follows directly from the previous lemmas.

Lemma 5. Given a base program \( P \) over \( A \) of class \( i \in \{1, 2\} \), and a preference program \( P_s \) of class \( j \in \{0, 1, 2\} \) for some preference statement \( s \), the problem Optimality is in \( \Pi^p_{\text{max}\{i,j\}} \).

PROOF. Let \( X \) be an stable model of \( P \). By item 1 of Proposition 3 we have that \( X \) is not a \( \succeq_s \)-preferred stable model of \( P \) iff the program \( P \cup P_s \cup R(A) \cup H(X)' \) is satisfiable. Since that program is of class \( \text{max}\{i, j\} \), we can conclude that the problem of deciding whether \( X \) is not a \( \succeq_s \)-preferred stable model of \( P \) is in \( \Sigma^p_{\text{max}\{i,j\}} \). Then, the lemma follows from the fact that this is the complement of the Optimality problem.

Lemma 6. Given a base program \( P \) over \( A \) of class \( i \in \{1, 2\} \), and a preference program \( P_s \) of class \( j \in \{0, 1, 2\} \) for some preference statement \( s \), the problem Optimality is \( \Pi^p_{\text{max}\{i,j\}} \)-hard.

PROOF. Let \( X \) be an stable model of \( P \), and let Non Optimality be the problem of deciding whether \( X \) is not a \( \succeq_s \)-preferred stable model of \( P \). We show below that Non Optimality is both \( \Sigma^p_i \) and \( \Sigma^p_j \)-hard. Hence, Non Optimality is \( \Sigma^p_{\text{max}\{i,j\}} \)-hard, and the lemma follows from the fact that Non Optimality is the complement of the Optimality problem.

Non Optimality is \( \Sigma^p_i \)-hard. Given a logic program \( P \) of class \( i \), the problem of deciding whether \( P \) has some stable model is \( \Sigma^p_i \)-complete. We prove \( \Sigma^p_i \)-hardness
by reducing that problem to Non Optimality where the base program is of class \(i\) and the preference program is of class \(0 \leq j\). Take any logic program \(P\) over \(\mathcal{A}\) of class \(i\), let \(\alpha\) be a fresh atom such that \(\alpha \notin \mathcal{A}\), and let \(W\) be the set \(\{\alpha\}\). Consider the base program \(P'\) of class \(i\), that contains the choice rule \(\{\alpha\} \leftarrow\), as well as all the rules of \(P\) extended with one additional literal \(\neg \alpha\) in the body. It is easy to see that the stable models of \(P'\) are the stable models of \(P\) together with the stable model \(W\). Next, let \(Q\) be the logic program of class 0 that consists of the two integrity constraints \(\leftarrow \text{holds}(\alpha)\) and \(\leftarrow \neg \text{holds}'(\alpha)\). Consider also the preference relation \(\succeq = \{(X,Y \cup \{\alpha\}) \mid X,Y \subseteq \mathcal{A}\} \cup \{(X,X) \mid X \subseteq \mathcal{A} \cup \{\alpha\}\}\) that specifies that the sets without \(\alpha\) are better than those with \(\alpha\). Clearly, \(\succeq\) is a preference relation, and \(Q\) is a preference program for \(\succeq\). Note that, for every stable model \(X\) of \(P\), it holds that \(X \succ W\). Then, we can reduce the problem of deciding whether the program \(P\) has some stable model to the problem of deciding whether the stable model \(W\) of \(P'\) is not a \(\succeq\)-preferred stable model of \(P'\). The reduction works given that, on the one hand, if \(P\) has some stable model \(X\), then \(W\) is not a \(\succeq\)-preferred stable model of \(P'\) because in this case \(X\) is a stable model of \(P'\) such that \(X \succ W\). And on the other hand, if \(P\) has no stable model, then \(W\) is a \(\succeq\)-preferred stable model of \(P'\) because in this case \(W\) is the unique stable model of \(P'\) and therefore there is no stable model \(X\) of \(P'\) such that \(X \succ W\).

**Non Optimality is \(\Sigma^P_j\)-hard.** Given a logic program \(Q\) of class \(j\), the problem of deciding whether \(Q\) has some stable model is \(\Sigma^P_j\)-complete. We prove \(\Sigma^P_j\)-hardness by reducing that problem to Non Optimality where the base program is of class \(1 \leq i\) and the preference program is of class \(j\). Take any logic program \(Q\) over \(\mathcal{A}\) of class \(j\), and let \(\alpha\) be a fresh atom such that \(\alpha \notin \mathcal{A}\). Moreover, let \(P\) be the base program that consists only of the choice rule \(\{\alpha\} \leftarrow\). The stable models of \(P\) are \(\emptyset\) and \(\{\alpha\}\). Next, let \(Q'\) be the logic program that contains the integrity constraints \(\leftarrow \text{holds}(\alpha)\) and \(\leftarrow \neg \text{holds}'(\alpha)\) as well as all the rules of \(Q\). Let \(EQ\) be \(\{(\emptyset,\emptyset), (\{\alpha\}, \{\alpha\})\}\), and consider the preference relation \(\succeq\) such that \(\succeq_\emptyset = \{(\emptyset,\emptyset)\} \cup EQ\) if \(Q\) has some stable model, and \(\succeq_\emptyset = EQ\) otherwise. Clearly, for any \(Q\), the preference relation \(\succeq\) is reflexive and transitive, and \(Q'\) is a preference program for \(\succeq\). Then, we can reduce the problem of deciding whether the program \(Q\) has some stable model to the problem of deciding whether the stable model \(\{\alpha\}\) of \(P\) is not a \(\succeq\)-preferred stable model of \(P\) (given \(P\) and \(Q'\)). The reduction works given that, on the one hand, if \(Q\) has some stable model \(X\), then \(\{\alpha\}\) is not a \(\succeq\)-preferred stable model of \(P\) because in this case \(\succeq_\emptyset = \{(\emptyset,\emptyset)\} \cup EQ\) and therefore \(\emptyset\) is a stable model of \(P\) such that \(\emptyset \succ \{\alpha\}\). And on the other hand, if \(Q\) has no stable model, then \(\{\alpha\}\) is a \(\succeq\)-preferred stable model of \(P\) because in this case \(\succeq_\emptyset = EQ\) and therefore there is no stable model \(X\).
of $P$ such that $X \succeq \{\alpha\}$.

**Lemma 7.** Given a base program $P$ over $A$ of class $i \in \{1, 2\}$, and a preference program $P_s$ of class $j \in \{0, 1, 2\}$ for some preference statement $s$, the problem Model Finding is in $F^{\Sigma^P_{\text{max}(i,j)}}_{\text{max}(i,j)+1}$.

**Proof.** We prove the lemma by showing a nondeterministic Turing machine, with an oracle for any problem in $\Sigma^P_{\text{max}(i,j)}$, that can compute a solution to the Model Finding problem (if it exists) in polynomial time. The machine first guesses a set $X \subseteq A$. Then it checks whether $X$ is a stable model of $P$, and whether $X$ is a $\succeq_s$-preferred stable model of $P$. If both checks succeed, it prints the solution $X$. The first check is in $\Pi^P_{i-1}$ (see Table III of [3]) and the second is in $\Pi^P_{\text{max}(i,j)}$ (Lemma 5), hence both can be performed by the oracle for $\Sigma^P_{\text{max}(i,j)}$ problems.

**Lemma 8.** Given a base program $P$ over $A$ of class $i \in \{1, 2\}$, and a preference program $P_s$ of class $j \in \{0, 1, 2\}$ for some preference statement $s$, the problem Model Finding is $F^{\Delta^P_{i+1}}_{i+1}$-hard.

**Proof.** The problem of computing a stable model of a logic program of class $i \in \{1, 2\}$ with optimization statements is $F^{\Delta^P_{i+1}}_{i+1}$-complete. For $i = 1$, the result can be found in Theorem 3.6 of [2]. For $i = 2$, the result can be easily obtained by modifying the proof of Theorem 19 of [92], which states that the problem of deciding if a given atom belongs to some optimal stable model of a logic program of class 2 with optimization statements is $\Delta^P_3$-complete. Then, the lemma follows from the translation of logic programs with optimization statements to logic programs with preference specifications in asprin, presented in Section 7.1.

**Lemma 9.** Given a base program $P$ over $A$ of class $i \in \{1, 2\}$, and a preference program $P_s$ of class $j \in \{0, 1, 2\}$ for some preference statement $s$, the problem Query is in $\Sigma^P_{\text{max}(i,j)+1}$.

**Proof.** The proof is analogous to the proof of Lemma 7, except for the fact that the machine only guesses sets $X \subseteq A$ that contain $\alpha$, and that it should return only ‘yes’ or ‘no’ depending on whether the guess passed the check.

In the next proofs we will use some complexity results about abduction from logic programs presented in [32]. We introduce those results simplifying slightly the notation, since we only need to consider one type of reasoning (brave reasoning under the stable model semantics) among the many considered in that paper. Let $A$ be a set of atoms. A logic programming abduction problem (LPAP)
A is a tuple \( \langle H, M, P \rangle \) where \( H \subseteq A \) is a finite set of hypotheses, \( M \subseteq A \cup \{ \neg a \mid a \in A \} \) is a finite set of manifestations, and \( P \) is a propositional logic program over \( A \). We say that a set of atoms \( X \) satisfies a set of literals \( M \) if \( X \) contains all the atoms \( a \) such that \( a \) belongs to \( M \), and \( X \) does not contain any atom \( a \) such that \( \neg a \) belongs to \( M \). Let \( \models \) be the brave reasoning relation under the stable model semantics. Formally, if \( P \) is a logic program over \( A \) and \( M \) is a set of literals over \( A \), then \( P \models M \) if there is some stable model \( X \) of \( P \) such that \( X \) satisfies \( M \). Let \( P = \langle H, M, P \rangle \) be a LPAP and let \( S \subseteq H \). Then, \( S \) is a solution to \( P \) if \( P \cup S \models M \), and \( S \) is also a \( \subseteq \)-solution to \( P \) if there is no solution \( S' \) to \( P \) such that \( S' \subseteq S \). We say that a hypothesis \( h \in H \) is relevant for \( P \) if there is some solution \( S \) to \( P \) such that \( h \in S \), and it is \( \subseteq \)-relevant for \( P \) if there is some \( \subseteq \)-solution \( S \) to \( P \) such that \( h \in S \). Let \( \subseteq \)-Relevance be the problem of deciding if a given hypothesis is \( \subseteq \)-relevant for a LPAP \( P = \langle H, M, P \rangle \). This problem is \( \Sigma^p_j \)-complete if \( P \) is normal (part (4) of Theorem 15 of [32]) and it is \( \Sigma^p_j \)-complete if \( P \) is disjunctive (Theorem 23 of [32]). In other words, \( \subseteq \)-Relevance is \( \Sigma^p_{i+1} \)-complete if \( P \) is of class \( i \in \{1, 2\} \). We can represent the solutions to \( P \) by the following logic program \( lp(P) \):

\[
P \cup \{ \{h\} \leftarrow h \in H \} \cup \{ \leftarrow \neg a \mid a \in A, a \in M \} \cup \{ \leftarrow a \mid a \in A, \neg a \in M \}.
\]

The choice rules guess some possible solution \( S \subseteq H \), while \( P \) together with the integrity constraints enforce that \( P \cup S \models M \). It is easy to see that there is a one-to-many correspondence between the solutions to \( P \) and the stable models of \( lp(P) \):

- if \( S \) is a solution to \( P \) then there is at least one stable model \( X \) of \( lp(P) \) such that \( X \cap H = S \), (A.46)
- and if \( X \) is a stable model of \( lp(P) \) then \( X \cap H \) is a solution to \( P \). (A.47)

Note that \( lp(P) \) is normal if \( P \) is normal, and it is disjunctive if \( P \) is disjunctive.

**Lemma 10.** Given a base program \( P \) over \( A \) of class 1, and a preference program \( P_s \) of class \( j \in \{1, 2\} \) for some preference statement \( s \), the problems Query and Non Dominance are \( \Sigma^p_{j+1} \)-hard.

**Proof.** We reduce the problem \( \subseteq \)-Relevance of deciding whether a given hypothesis \( h \) is \( \subseteq \)-relevant for a LPAP \( P = \langle H, M, P \rangle \) over \( A \), where \( P \) is of class \( j \). As we have seen before, this problem is \( \Sigma^p_{j+1} \)-complete.
Let $\alpha$ be a new atom that does not belong to $A$. Consider the following logic program $Q$:
\[
\{\{\alpha\} \leftarrow\} \cup \{\{a\} \leftarrow \alpha \mid a \in A\} \cup \{h \leftarrow \alpha\}.
\]

It is easy to see that

the set of stable models of $Q$ is $\{\emptyset \cup \{W \cup \{h, \alpha\} \mid W \subseteq A\}\}$. (A.48)

Consider also the relation $\succ$ over $A \cup \{\alpha\}$ such that $X \succ Y$ iff $X = \emptyset$ and $Y = W \cup \{\alpha\}$ for some set $W \subseteq A$ such that:
1. either $W$ is not a stable model of $lp(P)$,
2. or $W \cap H$ is not a $\subseteq$-solution to $P$.

Let $\succeq$ be $\succ \cup \{(X.X) \mid X \subseteq A \cup \{\alpha\}\}$. Clearly, $\succ$ is a strict partial order, and $\succeq$ is a preorder.

We show that the three following statements are equivalent:
1. The hypothesis $h \in H$ is $\subseteq$-relevant for $P$.
2. There is some $\succeq$-preferred stable model $Y$ of $Q$ such that $\alpha \in Y$.
3. There is some stable model $Y$ of $Q$ such that for $X = \{\emptyset\}$ it holds that $Y \neq X$ and $X \not\succ Y$ for all $X \in \mathcal{X}$.

\textit{From statement 1 to statement 2.} If statement 1 holds, then by definition

there is some $\subseteq$-solution $S$ for $P$ such that $h \in S$. (A.49)

Hence, by (A.46), we have that

there is some stable model $W$ of $lp(P)$ such that $W \cap H = S$. (A.50)

Since $h \in H$, $h \in S$ and $W \cap H = S$, we have that $h \in W$. Let $Y = W \cup \{\alpha\}$.

By (A.48) it follows that $Y$ is a stable model of $Q$. Furthermore, $\emptyset \succ Y$ does not hold because $W$ is a stable model of $lp(P)$ by (A.50), and $W \cap H$ is a $\subseteq$-solution to $P$ by (A.49) and (A.50). Hence, $Y$ is a $\succeq$-preferred stable model of $Q$, and statement 2 follows given that $\alpha \in Y$.

\textit{From statement 2 to statement 1.} If statement 2 holds, then $Y$ is a stable model of $Q$ such that $\alpha \in Y$, and by (A.48) we have that $Y$ has the form $W \cup \{\alpha\}$ for some $W \subseteq A$ such that $h \in W$. Since $Y$ is $\succeq$-preferred, it follows that $\emptyset \succ Y$ does not hold, and therefore $W \cap H$ is a $\subseteq$-solution to $P$. Given that $h \in W$ and $h \in H$, we can conclude that $h \in W \cap H$. Hence, $W \cap H$ is a $\subseteq$-solution to $P$ such that $h \in W \cap H$, and this implies statement 1.

\textit{Statement 2 is equivalent to statement 3.} Consider the following statements:
4. There is some stable model \( Y \) of \( Q \) such that \( \alpha \in Y \) and there is no stable model \( X \) of \( Q \) such that \( X \succ Y \).

5. There is some stable model \( Y \) of \( Q \) such that \( Y \neq \emptyset \) and \( \emptyset \nprec Y \).

Statement 2 is equivalent to statement 4 by definition of \( \succeq \)-preferred. We prove that statement 4 is equivalent to statement 5. From 4 to 5, \( \alpha \in Y \) implies that \( Y \neq \emptyset \), and by (A.48) we have that \( \emptyset \nprec Y \). From 5 to 4, \( Y \neq \emptyset \) together with (A.48) imply that \( \alpha \in Y \), and if \( \emptyset \nprec Y \) then by definition of \( \succ \) there is no set \( X \) such that \( X \succ Y \). Next, it is easy to see that statement 5 is just a rewriting of statement 3. Finally, the equivalence between statements 2 and 3 is implied by the previous equivalences.

Now, we can reduce the problem \( \subseteq \)-Relevance of deciding whether statement 1 holds to the Query problem of deciding whether statement 2 holds, and to the Non Dominance problem of deciding whether statement 3 holds. Observe that the logic program \( Q \) is of class 1. Then, to finish the proof, we just have to show a preference program \( R \) for \( \succeq \) of class \( j \). In other words, we have to show a program \( R \) of class \( j \) such that for every \( X, Y \subseteq A \cup \{ \alpha \} \) we have that \( X \succ Y \) iff \( R \cup H(X) \cup H'(Y) \) is satisfiable.

Before doing this, we provide an alternative definition of the strict partial order \( \succ \). We say that \( X \succ Y \) iff \( X = \emptyset \) and \( Y = W \cup \{ \alpha \} \) for some set \( W \subseteq A \) such that:

1. either \( W \) is not a classical model of \( lp(P) \),
2. or there is some set \( W' \subset W \) such that \( W' \) is a minimal model of the reduct of \( lp(P) \) wrt \( W \),
3. or there is some \( W' \subseteq A \) such that \( W' \) is a stable model of \( lp(P) \) and \( W' \cap H \subset W \cap H \).

The equivalence between condition 1 of the original definition and the disjunction of conditions 1 and 2 of the alternative definition follows from the definition of stable models. If none of those conditions hold, then \( W \) is a stable model of \( lp(P) \). In this case, by (A.46) and (A.47), the condition 2 of the original definition is equivalent to condition 3 of the alternative definition. This shows that both definitions are equivalent, and from now on we continue using the alternative one.

The preference program \( R \) has the form \( I \cup C_1 \cup C_2 \cup C_3 \). The rules in \( I \) constrain the form of \( X \) and \( Y \), select one condition \( i \in \{1, 2, 3\} \), and require the derivation of an special atom \( ok \). The rules in every \( C_i \) check that condition \( i \) holds whenever it is selected, and in this case they derive the atom \( ok \). As we will see, the set of rules \( I \) is of class 1, the set of rules \( C_1 \) is of class 0, while \( C_2 \) and \( C_3 \) are both of class \( j \). Hence, the program \( R \) is of class \( j \).
The first part of $I$ consists of the following rules:

$$\{ \leftarrow \text{holds}(a) \mid a \in A \cup \{ \alpha \} \} \cup \{ \leftarrow \neg \text{holds}'(\alpha) \}.$$ 

The first set enforces that $X = \emptyset$, and the second set enforces that $Y$ has the form $W \cup \{ \alpha \}$. We immediately have that $W \subseteq A$ since $\triangleright$ is defined over $A \cup \{ \alpha \}$.

The set $I$ also contains the following rules, that use the atoms $c_i$ for $i \in \{ 1, 2, 3 \}$ to represent condition $i$:

$$\{ \{ c_i \} \leftarrow i \in \{ 1, 2, 3 \} \} \cup \{ \leftarrow \neg c_1, \neg c_2, \neg c_3 \} \cup \{ \leftarrow c_i, c_j \mid i, j \in \{ 1, 2, 3 \}, i \neq j \}.$$ 

They state that exactly one condition $i$ has to be chosen. Finally, the following integrity constraint from $I$ ensures that the chosen condition derives the atom $ok$:

$$\leftarrow \neg ok.$$ 

The set of rules $C_1$ contains for every rule $r \in lp(P)$ of the form

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o$$

one rule of the form

$$ok \leftarrow \neg \text{holds}'(a_1), \ldots, \neg \text{holds}'(a_m),$$

$$\text{holds}'(a_{m+1}), \ldots, \text{holds}'(a_n), \neg \text{holds}'(a_{n+1}), \ldots, \neg \text{holds}'(a_o), c_1$$

that derives $ok$ if $W$ does not satisfy $r$ and condition 1 is selected. All together, the rules of $C_1$ enforce condition 1 by deriving the atom $ok$ if and only if $W$ is not a classical model of $lp(P)$.

The set of rules $C_2$ contains for every rule $r \in lp(P)$ of the form

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o$$

one rule of the form

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg \text{holds}'(a_{n+1}), \ldots, \neg \text{holds}'(a_o), c_2$$

where the atoms occurring in the negative literals are replaced by their interpretation in $Y$ using the unary predicate $\text{holds}'$. Moreover, for every choice rule $r \in lp(P)$ of the form

$$\{ a_0 \} \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

97
the set $C_2$ also contains one rule of the form

$$a_0 \leftarrow \text{holds}'(a_0), a_1, \ldots, a_m, \neg \text{holds}'(a_{m+1}), \ldots, \neg \text{holds}'(a_n), c_2$$

where the rule is translated to a normal one and the atom $\text{holds}'(a_0)$ is added to the body to guarantee that the rule is only fired if $a_0$ belongs to $W$. This set of rules generates minimal models $W'$ of the reduct of $lp(P)$ wrt $W$, whenever condition 2 is activated. Now, to represent condition 2, we only have to enforce that $W' \subset W$. We do this in two parts. First, we guarantee that $W' \subseteq W$ with these rules:

$$\{ \leftarrow a, \neg \text{holds}'(a), c_2 \mid a \in A \}$$

and then derive the special atom $ok$ when $W'$ is strictly smaller that $W$:

$$\{ ok \leftarrow \neg a, \text{holds}'(a), c_2 \mid a \in A \}.$$ 

The set of rules $R_3$ contains all the rules of $lp(P)$ extended with the additional atom $c_3$ in the body. This set of rules generates stable models $W'$ of $lp(P)$ when condition 3 is activated. Now, to represent condition 3, we only have to enforce that $W' \cap H \subset W \cap H$, and we do that with similar rules to the ones we used in $R_2$:

$$\{ \leftarrow a, \neg \text{holds}'(a), c_3 \mid a \in H \} \cup \{ ok \leftarrow \neg a, \text{holds}'(a), c_3 \mid a \in H \}$$

All in all, this combination of rules ensures that the program $R \cup H(X) \cup \text{H}'(Y)$ is satisfiable iff $X$ and $Y$ have the right form and some condition $i$ holds. This implies that $R$ is a preference program for $\succeq$ (of class $j$) and concludes the proof.

**Lemma 11.** Given a base program $P$ over $A$ of class $i \in \{1, 2\}$, and a preference program $P_s$ of class $j \in \{0, 1, 2\}$ for some preference statement $s$, the problem $	ext{Query}$ is $\Sigma_{\text{max}((i,j))}^{p}+1$-hard.

**PROOF.** We prove that if $j = 0$ then the problem is $\Sigma_{i+1}^{p}$-hard. This gives us immediately $\Sigma_{\text{max}((i,j))}^{p}+1$-hardness for all the combinations of $i$ and $j$, except for the case where $i = 1$ and $j = 2$, that follows from Lemma 10.

$\Sigma_{i+1}^{p}$-hardness when $j = 0$. We reduce the problem $\subseteq$-Relevance of deciding whether a given hypothesis $h$ is $\subseteq$-relevant for a LPAP $\mathcal{P} = \langle H, M, P \rangle$. As we have seen above, this problem is $\Sigma_{i+1}^{p}$-complete if $P$ is of class $i \in \{1, 2\}$. Consider the logic program $lp(\mathcal{P})$ of class $i$, that generates the solutions to $\mathcal{P}$. 98
Consider also the preference statement \( \#\text{preference}(s, \text{subset})H \) that declares the preference relation \( X \succeq_s Y \) as \( \{ h \in H \mid X \models h \} \subseteq \{ h \in H \mid Y \models h \} \). It is easy to see that the \( \subseteq \)-solutions to \( P \) correspond one-to-many to the \( \succeq_s \)-preferred stable models of \( lp(P) \). Recall that the problem \( \subseteq \)-Relevance amounts to deciding whether there is some \( \subseteq \)-solution \( S \) to \( P \) such that \( h \in S \). Then, \( \subseteq \)-Relevance can be reduced to the Query problem of deciding whether there is some \( \succeq_s \)-preferred stable model \( X \) of \( lp(P) \) such that \( h \in X \). To finish the proof, we just have to show how to build a preference program of class 0 for \( \succeq_s \), and this can be done simply adapting to our preference statement \( s \) the encoding of class 0 presented in Listing 1.

**Lemma 12.** Given a base program \( P \) over \( A \) of class \( i \in \{ 1, 2 \} \), and a preference program \( P_s \) of class \( j \in \{ 0, 1, 2 \} \) for some preference statement \( s \), the problem Non Dominance is in \( \Sigma_{\max(i,j+1)}^p \).

**PROOF.** We prove the lemma by showing a nondeterministic Turing machine, with an oracle for any problem in \( \Sigma_{\max(i,j+1)}^p \), that can solve the Non Dominance problem in polynomial time. The machine first guesses a set \( Y \subseteq A \). Then it checks whether \( Y \) is a stable model of \( P \), and whether for all \( X \in X \) it holds that \( X \neq Y \) and \( X \not\succ_s Y \). The first check is in \( \Pi_{i-1}^p \) (see Table III of [3]). In the second check, for each \( X \in X \), the condition \( X \neq Y \) can be decided in polynomial time, and the condition \( X \not\succ_s Y \) is in \( \Pi_j^p \). To see the latter, note that \( P_s \) is a preference program for \( \succeq_s \) of class \( j \). Hence, deciding whether \( X \succ_s Y \) is in \( \Sigma_j^p \) (see again Table III of [3]) and its complement is in \( \Pi_j^p \). Since the set \( X \) is part of the input, we can conclude that the second check on all the elements of \( X \) is also in \( \Pi_j^p \). Putting all together, it follows that the checks are in \( \Pi_{\max(i-1,j)}^p \), and therefore they can be performed by the oracle for \( \Sigma_{\max(i-1,j)}^p \) problems.

**Lemma 13.** Given a base program \( P \) over \( A \) of class \( i \in \{ 1, 2 \} \), and a preference program \( P_s \) of class \( j \in \{ 0, 1, 2 \} \) for some preference statement \( s \), the problem Non Dominance is \( \Sigma_{\max(i,j+1)}^p \)-hard.

**PROOF.** Recall that given a logic program \( Q \) of class \( i \), the problem of deciding whether \( Q \) has some stable model is \( \Sigma_i^p \)-complete. Then, we prove \( \Sigma_i^p \)-hardness by reducing that problem to Non Dominance where the set \( X \) is empty. This implies \( \Sigma_{\max(i,j+1)}^p \)-hardness for the cases where \( i = 1 \) and \( j = 0 \), and where \( i = 2 \) and \( j \in \{ 0, 1 \} \). On the other hand, Lemma 10 states \( \Sigma_{j+1}^p \)-hardness for the cases where \( i = 1 \) and \( j \in \{ 1, 2 \} \), from which we also have \( \Sigma_{j+1}^p \)-hardness for the case
where $i = 2$ and $j = 2$. This implies $\Sigma_{\max(i,j+1)}^p$-hardness for the cases where $i = 1$ and $j \in \{1, 2\}$, and where $i = 2$ and $j = 2$. Putting all together, we have $\Sigma_{\max(i,j+1)}^p$-hardness for all $i \in \{1, 2\}$ and $j \in \{0, 1, 2\}$, concluding the proof.

**Proof.** (Theorem 3)

Follows from Lemmas 5 to 9 and 11 to 13.