A General Framework for Preferences in Answer Set Programming

Gerhard Brewka[a], James Delgrande[b], Javier Romero[c,*], Torsten Schaub[c]

[a] *University of Leipzig*
[b] *Simon Fraser University*
[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 *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 *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), `torsten@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 $Y \not\succeq X$. 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".

— 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:

$$\#\mathit{preference}(\mathit{costs}, \mathit{less}(\mathit{weight}))\{40 : \mathit{sauna}, 70 : \mathit{dive}\} \tag{1}$$

This statement[2] declares a preference relation named costs with type $\mathit{less}(\mathit{weight})$ and argument set $\{40 : \mathit{sauna}, 70 : \mathit{dive}\}$. 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 '$\mathit{dive} > \{\mathit{sauna}, \mathit{swim}\} \| \mathit{hot}$' 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:

$$\#\mathit{preference}(\mathit{all}, \mathit{pareto})\{\mathit{name}(\mathit{costs}), \mathit{name}(\mathit{fun}), \mathit{name}(\mathit{temps})\} \tag{2}$$

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 $\#\mathit{optimize}(s)$ 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, ..., 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, ..., 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, ..., 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

$$a_1 \;; \ldots ; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \tag{3}$$

and *choice rules* of the form

$$\{a_1\} \leftarrow a_2, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \tag{4}$$

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 $a_i \overset{+}{\leftarrow} a_j$ for $1 \leq i \leq m$ and $m+1 \leq j \leq n$, and an edge $a_i \overset{-}{\leftarrow} a_j$ 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 ($\overset{-}{\leftarrow}$). 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 $a_{m+1} \wedge \cdots \wedge a_n \wedge \neg a_{n+1} \wedge \cdots \wedge \neg a_o \rightarrow a_1 \vee \cdots \vee a_m$ and every choice rule (4) of P with the formula $a_2 \wedge \cdots \wedge a_n \wedge \neg a_{n+1} \wedge \cdots \wedge \neg a_o \rightarrow a_1 \vee \neg a_1$, 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 $\texttt{a:b}_1\texttt{,...,b}_m$, the latter can be written as $\texttt{s\{c}_1\texttt{;...;c}_n\texttt{\}t}$, where $\texttt{a}$ and $\texttt{b}_i$ 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`$_1$`, ..., 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 $\mathcal{A}$, and *preference specifications* S over $\mathcal{A}$. We define below preference specifications, and for now we just say that they define a preorder $\succeq\ \subseteq \mathcal{A} \times \mathcal{A}$ among the interpretations of P. We refer to $\succeq$ as a *preference relation*. Given two interpretations $X, Y \subseteq \mathcal{A}$, 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 $\mathcal{A}$. 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[4] $t : \varepsilon$ where t is a tuple of terms and ε is either a Boolean expression ϕ over $\mathcal{A}$ with logical connectives $\top$, $\neg$, $\wedge$, and $\vee$; or a naming atom (see below). We drop the colon and simply write ε whenever t is empty. For expressing composite preferences, we use a dedicated unary naming predicate $name$ that allows us to refer to auxiliary preferences. That is, a *naming atom*, $name(s)$, refers to relations associated with a preference statement s (see below). Examples of weighted formulas include: $1, 1, 4 : \mathit{cuboid}(x) \wedge a$, and $2 : name(q)$.

A *preference element* is of the form[5]

$$\Phi_1 > \cdots > \Phi_m \parallel \phi \tag{5}$$

where ϕ is a non-weighted formula giving the context, and each Φ_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 ϕ is tautological. Also, we drop "$>$" if $m = 1$. Hence $\{a, \neg b\} > c$ stands for $\{a, \neg b\} > \{c\} \parallel \top$. Similarly, $40 : \mathit{sauna}$ in (1) abbreviates $\{40 : \mathit{sauna}\} \parallel \top$.

Preferences are declared by *preference statements* of the form

$$\#\mathit{preference}(s, t)\{e_1, \ldots, e_n\} \tag{6}$$

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 $dom(t)$. For instance, $\mathit{less}(\mathit{weight})$ in (1) is restricted to

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

[5]This 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 $\mathit{less}(\mathit{weight})$ in (1) is primitive.

To take effect, a set of preference statements is accompanied by a single *optimization directive* of form

$$\#\mathit{optimize}(s)$$

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 $s \in S$ whenever $\mathit{name}(s)$ 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 $\#\mathit{optimize}(s)$ such that $s \in S$ and S is acyclic and closed. We call s the *primary* preference statement in S and refer to statements in $S \setminus \{s\}$ as *auxiliary*.

3.2. Semantics

A preference statement like $\#\mathit{preference}(s,t)E$ 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 $E \in \mathit{dom}(t)$ to a preorder $t(E) \subseteq \mathcal{A} \times \mathcal{A}$ over $\mathcal{A}$. For simplicity, we often denote the relation $t(E)$ by $\succeq_s$, and its strict version by $\succ_s$.

As a first example of a definition of a preference type, consider a class of simple cardinality-based relations, referred to as $\mathit{less}(\mathit{card})$: [6]

$$(X,Y) \in \mathit{less}(\mathit{card})(E) \text{ iff } |\{\ell \in E \mid X \models \ell\}| \leq |\{\ell \in E \mid Y \models \ell\}|$$

where $\mathit{dom}(\mathit{less}(\mathit{card})) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$ and $\mathcal{P}(S)$ 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] $X \models \phi$ is the standard satisfaction relation between an interpretation X and a formula ϕ.

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

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

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

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

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

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

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 name. Given a naming atom $\mathit{name}(s)$, 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:

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

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

$$\begin{aligned}(X,Y) \in \mathit{lexico}(E) \text{ iff } & \bigvee_{w:\mathit{name}(s)\in E} \Big((X \succ_s Y) \wedge \bigwedge_{\substack{v:\mathit{name}(s')\in E \\ v>w}} (X =_{s'} Y)\Big) \vee \\ & \bigwedge_{w:\mathit{name}(s)\in E} (X =_s Y)\end{aligned} \tag{8}$$

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

Combining several preference relations of type *more(weight)* via *lexico* amounts to the orderings induced by *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 *subset* relations with *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 *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:

$\#\mathit{preference}(1, \mathit{less}(\mathit{card}))\{a, \neg b, c\})$ declares $X \succeq_1 Y$ as

$$|\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| \leq |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$

$\#\mathit{preference}(2, \mathit{more}(\mathit{weight}))\{1 : a, 2 : \neg b, 3 : c\}$ declares $X \succeq_2 Y$ as

$$\sum_{\substack{w:\ell \in \{1:a,2:\neg b,3:c\} \\ X \models \ell}} w \geq \sum_{\substack{w:\ell \in \{1:a,2:\neg b,3:c\} \\ Y \models \ell}} w$$

$\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\}$ declares $X \succeq_3 Y$ as

$$\{\ell \in \{a, \neg b, c\} \mid X \models \ell\} \subseteq \{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}$$

$\#\mathit{preference}(4, \mathit{pareto})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\}$ declares $X \succeq_4 Y$ as

$$(X \succeq_1 Y) \wedge (X \succeq_2 Y) \wedge (X \succeq_3 Y)$$

$\#\mathit{preference}(5, \mathit{lexico})\{1 : \mathit{name}(1), 2 : \mathit{name}(2), 3 : \mathit{name}(3)\}$ declares $X \succeq_5 Y$ as

$$\begin{array}{l}(X \succ_3 Y) \vee ((X =_3 Y) \wedge (X \succ_2 Y)) \vee \\ ((X =_3 Y) \wedge (X =_2 Y) \wedge (X \succ_1 Y)) \vee \\ ((X =_3 Y) \wedge (X =_2 Y) \wedge (X =_1 Y))\end{array}$$

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 $w_1, \dots, w_l : \phi$ occurring in some set Φ_r of a preference element e_j in a preference statement s as in (6) is represented as a fact of form[7]

$$\texttt{preference(s,j,r,for(t}_\phi\texttt{),(w}_1\texttt{,...,w}_l\texttt{)).}$$

where each $\texttt{w}_i$ represents w_i for $i = 1, \dots, l$ and $\texttt{t}_\phi$ is a term representing ϕ 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 $name(s)$ is represented analogously, except that $\texttt{for(t}_\phi\texttt{)}$ is replaced by `name(s)`.

[7]For tangibility, we present the format in terms of ASP source as produced by *asprin*.

[8]In *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 $\#\mathit{preference}(s,t)\{e_1,\dots,e_n\}$ as

$$F_s = \{\,\texttt{preference(s,t).}\} \cup \textstyle\bigcup_{j=1,\dots,n} F_{s,j}$$

where `s` and `t` represent s and t, respectively.

For example, the previous preference statements are translated by *asprin* as follows.

$\#\mathit{preference}(1,\mathit{less}(\mathit{card}))\{a,\neg b,c\}$) yields

```
preference(1,less(cardinality)).

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

$\#\mathit{preference}(2,\mathit{more}(\mathit{weight}))\{1:a,2:\neg b,3:c\}$) yields

```
preference(2,more(weight)).

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

$\#\mathit{preference}(3,\mathit{subset})\{a,\neg b,c\}$) yields

```
preference(3,subset).

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

$\#\mathit{preference}(4,\mathit{pareto})\{\mathit{name}(1),\mathit{name}(2),\mathit{name}(3)\}$) yields

```
preference(4,pareto).

preference(4,1,1,name(1),()).
preference(4,2,1,name(2),()).
preference(4,3,1,name(3),()).
```

$\#\mathit{preference}(5,\mathit{lexico})\{1:\mathit{name}(1),2:\mathit{name}(2),3:\mathit{name}(3)\}$ 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 holds and 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:

$$H(X) = \{\mathit{holds}(a) \leftarrow \ \mid a \in X\} \text{ and } H'(X) = \{\mathit{holds}'(a) \leftarrow \ \mid a \in X\}.$$

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

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

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 '$\mathit{optimize}(s) \leftarrow$' is added to F_s to represent the optimization directive $\#\mathit{optimize}(s)$. 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 $\mathit{better}(s)$ 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 ϕ, the rules in G warrant that `holds(`$\mathtt{t}_\phi$`)` is obtained when ϕ is entailed by the stable model X captured in $H(X)$ where t_ϕ is the term representation of ϕ. 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:

$$\leftarrow \neg \mathit{better}(P), \mathit{optimize}(P) \tag{9}$$

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 $\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\}$ 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 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.[9] Altogether the three

[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 $\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\}$.

parts constitute the preference program $P_3 = F_3 \cup E_{\mathit{subset}} \cup G$. With it, we may check whether $\{a, b\} \succ_3 \{a\}$ 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 $\{a\} \subset \{a, \neg b\}$ 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 $\succeq_s \subseteq \mathcal{A} \times \mathcal{A}$ and P_s be a preference program for s. Then, we have that $\succ_s$ is given by*

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

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:

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

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 $\succeq_s \subseteq \mathcal{A} \times \mathcal{A}$ and P_s be a preference program for s. Then, we have that $\succ_s$ is given by*

$$\begin{aligned}\{(X, Y) \mid {} & Z \textit{ is a stable model of } P_s \cup C(\mathcal{A}) \cup C'(\mathcal{A}), \\ & X = \{a \mid \mathit{holds}(a) \in Z\} \textit{ and } Y = \{a \mid \mathit{holds}'(a) \in Z\}\} . \end{aligned} \qquad (10)$$

As an example, let us reconsider the above preference program P_3 implementing $\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\}$. The logic program $P_3 \cup C(\{a, b, c\}) \cup C'(\{a, b, c\})$ results in 19 stable models capturing relation $\succ_3$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ to $H(\mathcal{A})$ and $H(\mathcal{A})'$, respectively. To formulate our result, we need the following set of rules subject to a set X of atoms:

$$R(X) = \{\mathit{holds}(a) \leftarrow a \mid a \in X\}$$

$R(X)$ 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 $\mathcal{A}$, 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 $\big(P \cup P_s \cup R(\mathcal{A}) \cup H(X)'\big)$ is unsatisfiable*
2. *if Y is a stable model of $\big(P \cup P_s \cup R(\mathcal{A}) \cup H(X)'\big)$ for some $X \subseteq \mathcal{A}$, then $Y \cap \mathcal{A}$ is a stable model of P such that $(Y \cap \mathcal{A}) \succ_s X$.*

We use $(P \cup P_s \cup R(\mathcal{A}) \cup H(X)')$ to check whether there is a model dominating X. Note how the usage of program $P \cup R_{\mathcal{A}}$ restricts candidates to stable models of P, unlike arbitrary subsets of $\mathcal{A}$ 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 `holds(a)` and `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

```
better(P) :- preference(P,less(cardinality)),
         #sum{ -1,X: holds(X),  preference(P,_,_,for(X),_);
                1,X: holds'(X), preference(P,_,_,for(X),_) } > 0.
```

$more(weight)$ is implemented by[10]

```
better(P) :- preference(P,more(weight)),
         #sum{  W,X: holds(X), preference(P,_,_,for(X),(W));
               -W,X: holds'(X),preference(P,_,_,for(X),(W))} > 0.
```

The implementation of $subset$ is given above.

In general, the correctness of a preference program is the responsibility of the implementer, just as with regular ASP encodings. However, for *asprin*'s preference library, we can provide correctness results.

Proposition 4. *Let s be an admissible preference statement of the form $\#preference(s, subset)E$, E_{subset} consist of the single rule in lines 8 to 10 of Listing 1, and F_s and G be defined as above. Then*

$$E_{subset} \cup F_s \cup G \textit{ is a preference program for } s.$$

Proposition 5. *Let s be an admissible preference statement of the form $\#preference(s, more(weight))E$, $E_{more(weight)}$ consist of the previous rule implementing $more(weight)$, and F_s and G be defined as above. Then*

$$E_{more(weight)} \cup F_s \cup G \textit{ is a preference program for } s.$$

The difference between primitive and composite preferences reduces to an issue of modularity when it comes to preference programs. This is because a preference program for a primary composite preference can be obtained by simply including all logic programs defining referenced auxiliary preferences. Hence, we do not further elaborate on preference programs for composite preferences here and rather concentrate on the design of preference modules in *asprin* for importing auxiliary preference programs in Section 8.

Two examples of composite preference types are $pareto$ and $lexico$:

[10]In the *asprin* system, the domain of $more(weight)$ is slightly different, being closer to the syntax of optimization statements in ASP. For this reason, the implementation in *asprin* is also slightly different (see Section 7.1).

$pareto$ is implemented by

```
better(P) :- preference(P,pareto),
             better(P''),   preference(P,_,_,name(P''),_),
             bettereq(P') : preference(P,_,_,name(P'),_).
```

$lexico$ is implemented by

```
better(P) :- preference(P,lexico);
             better(P'), preference(P,_,_,name(P'), (W)),
             equal(P''): preference(P,_,_,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 $bettereq$ or $equal$. For instance, a non-strict variant for $less(cardinality)$ can be implemented by replacing `better` and '>' above by `bettereq` and '>=', respectively. Similarly, an equality-oriented variant is obtained by using `equal` and '='.

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 `better` for $pareto$ but moreover programs defining `better` as well as `bettereq` for $less(card)$, $more(weight)$, and $subset$.

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.

Algorithm 1: $solveOpt(P, P_s)$

```
Input  : A finite program $P$ over $\mathcal{A}$ and
         a preference program $P_s$ for preference statement $s$.
Output : A $\succeq_s$-preferred stable model of $P$, if $P$ is satisfiable,
         and $\bot$ otherwise.
1 $Y \leftarrow solve(P)$
2 if $Y = \bot$ then return $\bot$
3 repeat
4     $X \leftarrow Y$
5     $Y \leftarrow solve(P \cup P_s \cup R(\mathcal{A}) \cup H(X)') \cap \mathcal{A}$
6 until $Y = \bot$
7 return $X$
```

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 preference 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 $\mathcal{A}$ 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     $Y \leftarrow solve\big(P \cup \bigcup_{X_i \in \mathcal{X}} \big(N_{X_i} \cup (\overline{P_s} \cup H(X_i))^i \cup R'(\mathcal{A})^i\big)\big) \cap \mathcal{A}$
4     if $Y = \bot$ then return $\mathcal{X}$
5     repeat
6         $X \leftarrow Y$
7         $Y \leftarrow solve\big(P \cup P_s \cup R(\mathcal{A}) \cup H(X)'\big) \cap \mathcal{A}$
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

$$N_X = \{\leftarrow X \cup \{\neg a \mid a \in \mathcal{A} \setminus X\}\} \, . \tag{11}$$

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

$$(P \setminus \{r \in P \mid \mathit{head}(r) = \emptyset\}) \cup \\ \{u \leftarrow \mathit{body}(r) \mid r \in P, \mathit{head}(r) = \emptyset\} \cup \{ \leftarrow \neg u\} , \tag{12}$$

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

Proposition 7. *If program P is stratified, P is satisfiable iff $\overline{P}$ 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 $X, Y \subseteq \mathcal{A}$, we have*

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

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 $\mathcal{A}$ and P_s be a stratified preference program for preference statement s. If Y is a stable model of*

$$\left(P \cup \overline{P_s} \cup H(X) \cup R'(\mathcal{A})\right)$$

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

Note that we also have $(Y \cap \mathcal{A}) \not\succ_s X$ whenever X is preferred wrt $\succeq_s$.

Given that $\mathcal{X}$ contains several preferred models, we need to test each model in $\mathcal{X}$ 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 $R'_i(X) = \{\mathit{holds}'(a)^i \leftarrow a \mid a \in X\}$. 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 $\mathcal{X}$ 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 $\mathcal{X}$ 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 $\mathcal{X} = \emptyset$ (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 $(Y_{i_0}, Y_{i_1}, \dots)$ of sets of atoms successively computed as follows:

- Y_{i_0} is the value of Y in Line 3 and
- Y_{i_j} is the value of Y in Line 7 at the jth 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 $\mathcal{X}_i$ 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 $(Y_{i_0}, \dots, Y_{i_n})$ for some integer $n \geq 0$, and*

1. *$n > 0$ iff $\mathcal{X}_{i-1}$ does not contain all $\succeq_s$-preferred stable models of P*
2. *Y_{i_j} is a stable model of P for $0 \leq j < n$*
3. *$Y_{i_j} \succ_s Y_{i_{j-1}}$ for $0 < j < n$*
4. *$Y_{i_n} = \bot$*
5. *$Y_{i_{n-1}}$ is a $\succeq_s$-preferred stable model of P, if $n > 0$*
6. *X_i is undefined if $n = 0$, and is $Y_{i_{n-1}}$ if $n > 0$*
7. *Y_{i_j} is such that $X_k \neq Y_{i_j}$ and $X_k \not\succ_s Y_{i_j}$ for $0 \leq j < n$ and all $X_k \in \mathcal{X}_{i-1}$*

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 $\mathcal{X}$ 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

$$\mathcal{R}\big(P_s \cup C(\mathcal{A}) \cup C'(\mathcal{A})\big) \cup \mathcal{M}\ . \tag{13}$$

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

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

$$\big(P \cup \textstyle\bigcup_{X\in\mathcal{X}} N_X\big) \cup \mathcal{Q}_s \cup \mathcal{N}(\mathcal{X}) \cup \mathcal{R}'(\mathcal{A}) \cup \{\leftarrow \neg bot\} \tag{14}$$

[13]For instance, programs can be reified with *clingo* via option `--output=reify`.

[14]Specific meta-encodings for saturation come with the respective distributions of *clingo*, and are bundled at https://github.com/potassco/clingo/tree/master/examples/reify.

As in Algorithm 2, $\big(P \cup \bigcup_{X\in\mathcal{X}} N_X\big)$ generates stable model candidates different from those in $\mathcal{X}$.[15] Programs $\mathcal{N}(\mathcal{X})$ and $\mathcal{R}'(\mathcal{A})$ restrict the choices of $C(\mathcal{A})$ and $C'(\mathcal{A})$ in (13), respectively.

$$\begin{aligned}
\mathcal{N}(\mathcal{X}) &= \{bot \leftarrow \textstyle\bigcup_{X_i\in\mathcal{X}}\{u_i\}\} \cup \textstyle\bigcup_{X_i\in\mathcal{X}}\{u_i \leftarrow \mathcal{R}^-(holds(a)) \mid a \in X_i\} \\
&\qquad\qquad\qquad\qquad \cup \textstyle\bigcup_{X_i\in\mathcal{X}}\{u_i \leftarrow \mathcal{R}^+(holds(a)) \mid a \in \mathcal{A}\setminus X_i\} \\
\mathcal{R}'(\mathcal{A}) &= \{\mathcal{R}^+(holds'(a)) \leftarrow a \mid a \in \mathcal{A}\} \cup \{\mathcal{R}^-(holds'(a)) \leftarrow \neg a \mid a \in \mathcal{A}\}
\end{aligned}$$

Just like bot, $\mathcal{R}^+(a)$ and $\mathcal{R}^-(a)$ belong to the signature of the used saturation encoding $\mathcal{M}$; they encode reified positive and negative atoms, respectively.[16] Unlike this, u_i is a new atom for each $X_i \in \mathcal{X}$ (similar to atom u in (12)). While $\mathcal{N}(\mathcal{X})$ 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 $\mathcal{R}(H(\mathcal{A})')$.

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

[15]The definition of N_X is given in (11).

[16]For instance, in the meta encoding in [24] literals a and $\neg a$ are represented as $true(atom(a))$ and $fail(atom(a))$, 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

$$\mathcal{R}\big(P_s \cup (P \cup R(\mathcal{A})) \cup C'(\mathcal{A})\big) \cup \mathcal{M}\ . \tag{15}$$

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

This leads us to the following disjunctive logic program, $\mathcal{E}_{P,s}$, extending P.

$$P \cup \big(\mathcal{R}\big(P_s \cup P \cup R(\mathcal{A}) \cup C'(\mathcal{A})\big) \cup \mathcal{M}\big) \cup \mathcal{R}'(\mathcal{A}) \cup \{\leftarrow \neg bot\} \tag{16}$$

The stable models Y of P are mapped to the signature of $\mathcal{M}$ by $\mathcal{R}'(\mathcal{A})$, 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 $\mathcal{E}_{P,s}$ 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 $\mathcal{E}_{P,s} \cup \{\leftarrow \neg a\}$ 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 $\mathcal{E}_{P,s}$ 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 $\mathcal{E}_{P,s}$ 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.[17]

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.[18] 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 P^C (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 Σ^p_k, Π^p_k and Δ^p_k are defined as follows:

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

In particular, $NP = \Sigma^p_1$ and $co\text{-}NP = \Pi^p_1$. 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

[17]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.

[18]For instance, *clasp* offers both types of enumeration because "solution recording" is sometimes more effective [28].

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 $a \in \mathcal{A}$, 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 $X \not\succ Y$ 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 Σ_0^p-complete if P is stratified, is Σ_1^p-complete if P is normal, and is Σ_2^p-complete if P is disjunctive (cf. [3]).[19]

[19]The complexity results from [3], unlike ours, consider normal and disjunctive logic programs *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.

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 $F\Delta^p_{i+1}$-hard and belongs to $F\Sigma^p_{max(\{i,j\})+1}$.*
- *The problem Query is $\Sigma^p_{max(\{i,j\})+1}$-complete.*
- *The problem Optimality is $\Pi^p_{max(\{i,j\})}$-complete.*
- *The problem Non Dominance is $\Sigma^p_{max(\{i,j+1\})}$-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 Σ^p_{i-1} oracle, and their optimality with a $\Sigma^p_{max(\{i,j\})}$ 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 $\mathcal{X}$ using a Σ^p_j 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 $F\Delta^p_2$. In fact, for all preference types that we have studied that are decidable in polynomial time, the Model Finding problem is in $F\Delta^p_2$ (when $i = 1$). For example, for $more(weight)$, 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 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 exponential 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 disjunctive.

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 Σ_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 Π_i^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 program is stratified, then the problem is in Σ_i^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 preference program is normal, then the problem is Σ_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 Σ_3^p-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 '$\boldsymbol{t}$ `::` ε' where $\boldsymbol{t}$ 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 ε is either a Boolean formula or a naming atom. Both $\boldsymbol{t}$ and ε may contain variables. Boolean formulas are formed from atoms using connectives `not`, `&`, and `|`. If they have the form ϕ_1 `&` $\ldots$ `&` ϕ_n, then they can also we written using the body notation ϕ_1`,` $\ldots$`,` ϕ_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 $\boldsymbol{t}_1$ `::` $\varepsilon_1, \ldots, \boldsymbol{t}_n$ `::` ε_n are weighted formulas, then '`{`$\boldsymbol{t}_1$ `::` ε_1`;`$\ldots$`;`$\boldsymbol{t}_n$ `::` ε_n`}`' is a set of weighted formulas.

A *preference element* is of the form 'Φ_1 `>>...>>` Φ_n `||` ϕ `:` B' where each Φ_i is a set of weighted formulas, ϕ 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 `;...;` 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 '`#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 $\mathcal{D}$ 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 $\mathcal{D}$, and then we just remove the ground bodies from them. For example, given this logic program with preferences:

```
dom(1..2).
{ a(X,Y) : dom(X), dom(Y)}.

#preference(p(X),subset){ a(X,Y) : dom(Y) } : dom(X).
#optimize(p(X)) : dom(X), not dom(X+1).
```

the set $\mathcal{D}$ is {`dom(1),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 `do` and `hot` are domain predicates:

```
 1  #preference(costs,less(weight)){
 2    C,A :: do(A) : cost(A,C)
 3  }.
 4  #preference(fun,superset){
 5        do(A) : like(A);
 6    not do(A) : dislike(A)
 7  }.
 8  #preference(temps,aso){
 9    do(A) >> do(B) ||     hot : outdoor(A),  indoor(B);
10    do(A) >> do(B) || not hot :  indoor(A), outdoor(B)
11  }.
12  #preference(all,pareto){ **costs; **fun; **temps }.
13
14  #optimize(all).
```

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*.[20] A *#minimize* directive is of the form

$$\#minimize\{w_1@k_1, \vec{t_1} : \vec{\ell_1}, \dots, w_n@k_n, \vec{t_n} : \vec{\ell_n}\}$$

where each w_i and k_i is an integer, and $\vec{t_i} = t_{i_1}, \dots, t_{i_m}$ and $\vec{\ell_i} = \ell_{i_1}, \dots, \ell_{i_k}$ are tuples of terms and literals, respectively. For a set X of atoms and an integer k, let Σ_k^X denote the sum of weights w_i over all occurrences of elements $(w_i@k_i, \vec{t_i} : \vec{\ell_i})$ in M such that $X \models \vec{\ell_i}$. Then, for sets X, Y of atoms and minimize statement M

[20]The decomposition of weak constraints is analogous, and is omitted for brevity.

as above, we have that $X \succ_M Y$ if there is some integer k such that $\Sigma_k^X < \Sigma_k^Y$ and $\Sigma_{k'}^X = \Sigma_{k'}^Y$ for all $k' > k$; and $X \succeq_M Y$ if either $X \succ_M Y$ or $\Sigma_k^X = \Sigma_k^Y$ for all integers k.

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

#preference(s$_M$,lexico){ k :: **s$_k$ $\mid (w@k, \vec{t}\colon \vec{\ell}) \in M$ }.
#preference(s$_k$,less(weight)){ w, ($\vec{t}$) :: $\vec{\ell} \mid (w@k, \vec{t}\colon \vec{\ell}) \in M$ }.
#optimize(s$_M$).

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

```
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 $\vec{t}$ into $(\vec{t})$, we only deal with pairs w, ($\vec{t}$) rather than tuples of varying length. For the aggregation by lexico, the implementation includes also a rule definining 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 $\mathit{less}(\mathit{weight})$.

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, $E_{\mathit{less}(\mathit{weight})}$ be the previous rules for $\mathit{less}(\mathit{weight})$, 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 $E_{\mathit{lexico}} \cup E_{\mathit{less}(\mathit{weight})} \cup F_M \cup G$ 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

$$\phi_1 > \cdots > \phi_m \leftarrow B \tag{17}$$

where each ϕ_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 $X \not\models b$ 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 $r \in O$,

$v_X(r) \leq v_Y(r)$, 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 *asprin*, we can represent an ASO rule r as in (17) as a preference statement of the form

```
#preference($s_r$,aso){ $\phi_1$>>...>>$\phi_m$ || $B$ }.
```

A set $\{r_1, \ldots, r_n\}$ of ASO rules is represented by corresponding preference statements $\mathtt{s}_{r_1}$ to $\mathtt{s}_{r_n}$ along with an aggregating *pareto* preference subject to optimization.

```
#preference(paraso,pareto){ **$s_{r_1}$, ..., **$s_{r_n}$ }.
#optimize(paraso).
```

Note that other composite preferences instead of *pareto* could be used just as well.

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

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

 8 one'(P) :- preference(P,aso),
 9            not holds'(F) : preference(P,_,R,for(F),_), R>1.
10 one'(P) :- preference(P,aso),
11            holds'(F), preference(P,_,1,for(F),_).
12 one'(P) :- preference(P,aso),
13            not holds'(F), preference(P,_,0,for(F),_).
```

With these rules, we derive `better(`$\mathtt{s}_r$`)` in Line 15 whenever some ASO rule r has satisfaction degree 1 in X and one greater than 1 in Y. Otherwise, `better(`$\mathtt{s}_r$`)` is derivable in Line 16 whenever r has satisfaction degree `R` in X but none of the formulas ϕ_1 to $\phi_{\mathtt{R}}$ are true in Y. This is analogous for `bettereq` in lines 20-23.

```
15 better(P) :- preference(P,aso), one(P), not one'(P).
16 better(P) :- preference(P,aso),
17      preference(P,_,R,for(F),_), holds(F), R > 1, not one'(P),
18      not holds'(G) : preference(P,_,R',for(G),_), 1 < R',R' <= R.

20 bettereq(P) :- preference(P,aso), 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 $pareto$, 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_{pareto} be the preference program for $pareto$, E_{aso} be the previous rules for aso, and F_O be the set of facts for the corresponding preference specification. Then, it can be shown that $E_{pareto} \cup E_{aso} \cup F_O \cup G$ is a preference program for $\succeq_O$.

7.3. Posets

In [8], qualitative preferences are modeled by a strict partially ordered set $(\Phi, <)$ of literals, also called a *poset*. The literals in Φ represent propositions that are preferably satisfied and the strict partial order $<$ on Φ 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 $(\Phi, <)$, $X \succ_{(\Phi,<)} Y$ 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 $\succeq_{(\Phi,<)}$ as follows: $X \succeq_{(\Phi,<)} Y$ if either $X \succ_{(\Phi,<)} Y$, 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 $(\Phi, <)$ by a preference statement $s_{(\Phi,<)}$ of form:

#preference(s$_{(\Phi,<)}$,poset) $\Phi \cup \{\, \phi' >> \phi \mid \phi' < \phi \,\}$.

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

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

```
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),_).

6 better(P,F)  :- preference(P,poset),
7                 poset(P,F), holds(F), not holds'(F).
8 notbetter(P) :- preference(P,poset),
 9                  poset(P,F), not holds(F), holds'(F),
10                  not better(P,G) : poset(P,G,F).

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

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

Finally, we sketch how these rules capture the intended semantics. For this, given a strict partially ordered set $(\Phi, <)$, we let E_{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 $E_{poset} \cup F_{(\Phi,<)} \cup G$ 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 ϕ, [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_ϕ of form

`#preference(`s_ϕ`,basic){` ϕ `}.`

The preference type *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 $\lhd$ directly correspond to *asprin*'s preference types *neg*, *and*, *pareto*, 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).[21] Preference programs can be im-

[21] If 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[22]

`#heuristic` a `:` l_1`,...,`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>[,<v>]`, 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

```
#heuristic holds(X) : preference(_,_,_,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[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;[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*.[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.

[23]Other competition classes were either too easy or too difficult.

[24]This is originally expressed as a common minimize statement.

[25]A cutoff at 900 seconds brought only a handful of additional instances.

Benchmark class	Type	Instances	Elements
Ricochet	c	30	20.00
Valves	w	30	56.63
Crossing	c	24	211.92
Puzzle	c	7	580.57
Diagnosis	c	30	1669.00
Repair	c	30	6750.73
Expansion	c	30	7501.87
Timetabling	w	12	23687.75

Table 1: Benchmark classes of ASP optimization problems.

We ran all benchmark instances with *asprin-1* and *clingo-4*[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.[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

[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.

[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.

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

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 were *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 by *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

[28]We 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.

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*$_l$, 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 traditionally 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 disjunctive 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 improvements. 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.

	metasp	*asprin*	*asprin*$_s$	*asprin*$_l$	*asprin*$_f$
Ricochet	811 (24)	365 (3)	461 (10)	**69** (**0**)	71 (**0**)
Valves	900 (30)	**38** (**0**)	39 (**0**)	339 (6)	673 (21)
Crossing	62 (**0**)	**0** (**0**)	1 (**0**)	7 (**0**)	3 (**0**)
Puzzle	35 (**0**)	31 (**0**)	32 (**0**)	**21** (**0**)	51 (**0**)
Diagnosis	182 (6)	19 (**0**)	2 (**0**)	**0** (**0**)	**0** (**0**)
Repair	900 (30)	8 (**0**)	3 (**0**)	**1** (**0**)	**1** (**0**)
Expansion	900 (30)	64 (**0**)	14 (**0**)	4 (**0**)	**3** (**0**)
Timetabling	799 (10)	217 (2)	21 (**0**)	900 (12)	**5** (**0**)
Total	574 (130)	93 (**5**)	**72** (10)	168 (18)	101 (21)

Table 3: Subset optimization with *metasp* and *asprin*.

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

[29]In [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).

	satpref	*satpref$_s$*	*asprin*	*asprin$_s$*
0	**0** (**0**)	**0** (**0**)	1 (**0**)	**0** (**0**)
0.00621	**0** (**0**)	**0** (**0**)	1 (**0**)	1 (**0**)
0.01243	**1** (**0**)	**1** (**0**)	6 (**0**)	2 (**0**)
0.02486	8 (**0**)	**6** (**0**)	55 (**0**)	9 (**0**)
0.04972	67 (2)	**16** (**0**)	318 (16)	26 (**0**)
1	850 (88)	243 (10)	856 (92)	**174** (**0**)
Total	154 (90)	44 (10)	206 (108)	**35** (**0**)

Table 4: *Poset* optimization on random benchmarks with *satpref* and *asprin*.

	satpref	*satpref$_s$*	*asprin*	*asprin$_s$*
Maxsat	54 (**0**)	**9** (**0**)	835 (31)	109 (3)
Partial-minone	**14** (**0**)	**14** (**0**)	24 (**0**)	24 (**0**)
Pbo-mqc-nencdr	5 (**0**)	**2** (**0**)	150 (14)	9 (**0**)
Pbo-mqc-nlogencdr	3 (**0**)	**1** (**0**)	110 (3)	5 (**0**)
Pseudo-primes	110 (18)	110 (18)	215 (27)	**106** (**17**)
Pseudo-routing	346 (4)	49 (**0**)	85 (**0**)	**4** (**0**)
Total	88 (22)	**31** (**18**)	236 (75)	43 (20)

Table 5: *Poset* optimization on structured benchmarks with *satpref* and *asprin*.

improve the performance of both systems. In the random benchmarks, *asprin$_s$* is faster than *satpref$_s$* on the class with more preferences, while *satpref$_s$* is faster on the others. In the structured benchmarks, *asprin$_s$* 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$_s$* 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_l), which amount to an aggregation of *aso* preferences with *lexico* in $asprin_l$. The generator assigns a higher rank to half of the ASO rules to account for this.

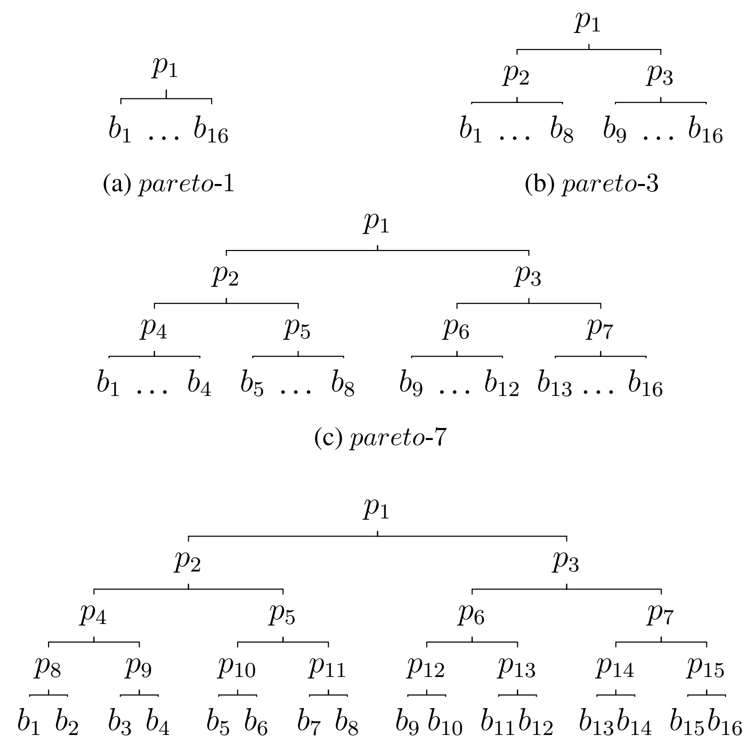
We created a benchmark set comprising 20 instances for every value of n ranging from 350 to 490 in increments of 10. Table 6 shows the results of the comparison of *asprin* and *aso* on this benchmark set. Overall, we observe that the general-purpose approach of *asprin* is comparable with the dedicated 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.

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

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 $pareto$-1, -3, -7 and -15, where the b_i's are basic statements, and the p_i's are pareto statements. The representation using *lexico* is similar, but the pareto statements p_i are replaced by lexico statements l_i, whose priority decreases as their subindex i increases.


p_1
$b_1 \dots b_{16}$
(a) $pareto$-1
p_1
p_2 p_3
$b_1 \dots b_8$ $b_9 \dots b_{16}$
(b) $pareto$-3
p_1
p_2 p_3
p_4 p_5 p_6 p_7
$b_1 \dots b_4$ $b_5 \dots b_8$ $b_9 \dots b_{12}$ $b_{13} \dots b_{16}$
(c) $pareto$-7
p_1
p_2 p_3
p_4 p_5 p_6 p_7
p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15} b_{16}

(d) $pareto$-15

Figure 1: Pareto aggregation using different numbers of composite preferences.

Tables 7 and 8 summarize the results for *pareto* and *lexico*, respectively, using *asprin*'s default configuration.

Overall, we find that the number of composite preferences does not significantly impact the performance or *asprin*, except in the classes *Repair* and *Expansion* using *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 *pareto* optimization is faster than *lexico* optimization, which is expected since, by design, every *lexico* optimal model is also *pareto* optimal.

	pareto-1	*pareto-3*	*pareto-7*	*pareto-15*
Ricochet	**115** (**0**)	118 (**0**)	137 (**0**)	131 (**0**)
Valves	**39** (**0**)	**39** (**0**)	**39** (**0**)	40 (**0**)
Crossing	**2** (**0**)	**2** (**0**)	**2** (**0**)	**2** (**0**)
Puzzle	**24** (**0**)	**24** (**0**)	**24** (**0**)	**24** (**0**)
Diagnosis	**900** (**30**)	**900** (**30**)	**900** (**30**)	**900** (**30**)
Repair	**5** (**0**)	14 (**0**)	16 (**0**)	17 (**0**)
Expansion	**122** (**0**)	425 (3)	451 (4)	481 (2)
Timetabling	629 (**8**)	**609** (**8**)	641 (**8**)	610 (**8**)
Total	**229** (**38**)	266 (41)	276 (42)	275 (40)

Table 7: Pareto optimization with *asprin* using different numbers of composite preferences.

	lexico-1	*lexico-3*	*lexico-7*	*lexico-15*
Ricochet	123 (**0**)	124 (**0**)	121 (**0**)	**112** (**0**)
Valves	50 (**0**)	43 (**0**)	**42** (**0**)	47 (**0**)
Crossing	**4** (**0**)	**4** (**0**)	**4** (**0**)	**4** (**0**)
Puzzle	58 (**0**)	56 (**0**)	**51** (**0**)	57 (**0**)
Diagnosis	**900** (**30**)	**900** (**30**)	**900** (**30**)	**900** (**30**)
Repair	**900** (**30**)	**900** (**30**)	**900** (**30**)	**900** (**30**)
Expansion	**900** (**30**)	**900** (**30**)	**900** (**30**)	**900** (**30**)
Timetabling	**824** (**10**)	826 (**10**)	841 (11)	831 (11)
Total	470 (**100**)	**469** (**100**)	470 (101)	**469** (101)

Table 8: Lexico optimization with *asprin* using different numbers of composite preferences.

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 preference handling in ASP. Our intention was not primarily to come up with new preference relations on stable models that have not been previously studied (although 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 optimization 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 preference 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 *asprin* 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 composite 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* preferences 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 feasible outcomes, and selectors that determine which of them are optimal.[30] 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. Succintness 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]

[30]The paper also studies formalisms like logic programs with ordered disjunctions ([6], see below) where the separation is not strict, and a single syntactic component (a rule, for example) takes part both in the generation of outcomes and in the selection of the optimal ones.

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:

$$\mathit{dive} \times \mathit{beach} \leftarrow \mathit{hot}$$

Together with the fact $\mathit{hot} \leftarrow$, that rule leads to the unique optimal solution $\{\mathit{hot}, \mathit{dive}\}$, while adding the constraint $\leftarrow \mathit{dive}$ the unique optimal solution becomes $\{\mathit{hot}, \mathit{beach}\}$. 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:

$$r(C) : \mathit{ab}(C) \overset{+}{\leftarrow} \mathit{component}(C)$$

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

$$\begin{aligned} \{\mathit{appl}(r(C))\} &\leftarrow \mathit{component}(C) \\ \mathit{ab}(C) &\leftarrow \mathit{component}(C), \mathit{appl}(r(C)) \\ \#\mathit{preference}(1, \mathit{subset})\{\mathit{appl}(r(C)) &: \mathit{component}(C)\} \end{aligned}$$

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 $\mathit{prefer}(r(c1), r(c2)) \leftarrow$ can represent our preference towards faults of component $c1$ over faults of component $c2$. The semantics of the language define a relation between models using these $\mathit{prefer}/2$ 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(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-

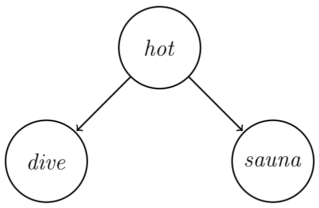


Figure 2: A preference tree.

erence 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 $(\{hot, hot \wedge dive, \neg hot \wedge sauna\}, >)$ where the partial order specifies that $hot > hot \wedge dive$ and $hot > \neg hot \wedge sauna$. 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:

$$dive > \neg dive \parallel hot \qquad\qquad sauna > \neg sauna \parallel \neg hot$$

The net represents the order where $\{dive, hot\}$ is preferred to $\{hot\}$, $\{dive, hot, sauna\}$ is preferred to $\{hot, sauna\}$, $\{sauna\}$ is preferred to $\emptyset$, and $\{dive, sauna\}$ is preferred to $\{dive\}$. 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:

$$gain(dive) : in(hot) \qquad\qquad gain(sauna) : \neg in(hot)$$

The second work introduces a preference logic where preference formulas are Boolean combinations of preference statements of the form $\alpha \rhd \beta \| F$ [31] where α and β 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 $\alpha \rhd \beta \parallel F$ if for every two sets X and Y such that (i) both interpret the same way F, (ii) X satisfies α and (iii) Y satisfies β, it holds that $X \succ Y$. Following [19], the previous CP-net can be represented by the next preference formula:

$$\big(dive \wedge hot > \neg dive \wedge hot \| \{sauna\}\big) \wedge \big(sauna \wedge \neg hot > \neg sauna \wedge \neg hot \| \{dive\}\big)$$

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 $\rhd$ is replaced by $\trianglerighteq$.

extension of the description logic $SROEL(\sqcup, \times)$ 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 $\mathcal{ALC}$ 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 $\mathcal{SROIQ}$ 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 *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 *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 *asprin* with *superset* preferences. The second focuses on representing and reasoning about the rule-based argumentation framework of ASPIC+, using *asprin* to represent the preferred semantics, again with the preference type *superset*.

11. Conclusion

In this work, we presented *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 *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 *asprin* in general compares well to dedicated implementations for preferences, although there was a gap with respect to *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 *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 *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 *asprin* to multi-shot solving and theory solving, paralleling the extension of *clingo* in the same directions [9, 83]. Additionally, we would like to study the application of *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

[1] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.

[2] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (1-2) (2002) 181–234.

[3] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.

[4] G. Brewka, I. Niemelä, M. Truszczyński, Answer set optimization, in: G. Gottlob, T. Walsh (Eds.), Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Morgan Kaufmann Publishers, 2003, pp. 867–872.

[5] G. Brewka, Complex preferences for answer set optimization, in: D. Dubois, C. Welty, M. Williams (Eds.), Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04), AAAI Press, 2004, pp. 213–223.

[6] G. Brewka, I. Niemelä, T. Syrjänen, Logic programs with ordered disjunction, Computational Intelligence 20 (2) (2004) 335–357.

[7] T. Son, E. Pontelli, Planning with preferences using logic programming, Theory and Practice of Logic Programming 6 (5) (2006) 559–608.

[8] E. Di Rosa, E. Giunchiglia, M. Maratea, Solving satisfiability problems with preferences, Constraints 15 (4) (2010) 485–515.

[9] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (1) (2019) 27–82.

[10] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3-4) (1999) 241–273.

[11] M. Banbara, M. Gebser, K. Inoue, M. Ostrowski, A. Peano, T. Schaub, T. Soh, N. Tamura, M. Weise, aspartame: Solving constraint satisfaction problems with answer set programming, in: Calimeri et al. [84], pp. 112–126.

[12] G. Brewka, J. Delgrande, J. Romero, T. Schaub, asprin: Customizing answer set preferences without a headache, in: B. Bonet, S. Koenig (Eds.), Proceedings of the Twenty-ninth National Conference on Artificial Intelligence (AAAI'15), AAAI Press, 2015, pp. 1467–1474.

[13] G. Brewka, J. Delgrande, J. Romero, T. Schaub, Implementing preferences with asprin, in: Calimeri et al. [84], pp. 158–172.

[14] K. Apt, H. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, 1987, Ch. 2, pp. 89–148.

[15] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–385.

[16] P. Ferraris, Answer sets for propositional theories, in: C. Baral, G. Greco, N. Leone, G. Terracina (Eds.), Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05), Vol. 3662 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2005, pp. 119–131.

[17] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory and Practice of Logic Programming 20 (2) (2020) 294–309.

[18] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, *Clingo* = ASP + control: Preliminary report, in: M. Leuschel, T. Schrijvers (Eds.), Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14), Vol. 14(4-5) of Theory and Practice of Logic Programming, Online Supplement, 2014.

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

[20] V. Pareto, Cours d'economie politique, Librairie Droz, 1964.

[21] J. McCarthy, Applications of circumscription to formalizing common-sense knowledge, Artificial Intelligence 28 (1986) 89–116.

[22] E. Giunchiglia, M. Maratea, Algorithms for solving satisfiability problems with qualitative preferences, in: E. Erdem, J. Lee, Y. Lierler, D. Pearce (Eds.), Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz, Vol. 7265 of Lecture Notes in Computer Science, Springer-Verlag, 2012, pp. 327–344.

[23] T. Eiter, G. Gottlob, On the computational cost of disjunctive logic programming: Propositional case, Annals of Mathematics and Artificial Intelligence 15 (3-4) (1995) 289–323.

[24] M. Gebser, R. Kaminski, T. Schaub, Complex optimization in answer set programming, Theory and Practice of Logic Programming 11 (4-5) (2011) 821–839.

[25] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, Theory and Practice of Logic Programming 23 (1) (2023) 299–361.

[26] T. Janhunen, E. Oikarinen, Capturing parallel circumscription with disjunctive logic programs, in: J. Alferes, J. Leite (Eds.), Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04), Vol. 3229 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 134–146.

[27] J. Romero, T. Schaub, P. Wanko, Computing diverse optimal stable models, in: Carro and King [86], pp. 3:1–3:14.

[28] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set enumeration, in: C. Baral, G. Brewka, J. Schlipf (Eds.), Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), Vol. 4483 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2007, pp. 136–148.

[29] C. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[30] C. Boutilier, R. Brafman, C. Domshlak, H. Hoos, D. Poole, CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements, Journal of Artificial Intelligence Research 21 (2004) 135–191.

[31] M. Alviano, J. Romero, T. Schaub, On the integration of cp-nets in ASPRIN, in: S. Kraus (Ed.), Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI'19), ijcai.org, 2019, pp. 1495–1501.

[32] T. Eiter, G. Gottlob, N. Leone, Abduction from logic programs: Semantics and complexity, Theoretical Computer Science 189 (1-2) (1997) 129–177.

[33] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco User Guide, version 2.2.0 Edition (2019).

[34] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

[35] M. Järvisalo, T. Junttila, I. Niemelä, Unrestricted vs restricted cut in a tableau method for Boolean circuits, Annals of Mathematics and Artificial Intelligence 44 (4) (2005) 373–399.

[36] M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, P. Wanko, Domain-specific heuristics in answer set programming, in: desJardins and Littman [87], pp. 350–356.

[37] E. Di Rosa, E. Giunchiglia, Combining approaches for solving satisfiability problems with qualitative preferences, AI Communications 26 (4) (2013) 395–408.

[38] Y. Zhu, M. Truszczyński, On optimal solutions of answer set optimization problems, in: Cabalar and Son [88], pp. 556–568.

[39] M. Gebser, H. Jost, R. Kaminski, P. Obermeier, O. Sabuncu, T. Schaub, M. Schneider, Ricochet robots: A transverse ASP benchmark, in: Cabalar and Son [88], pp. 348–360.

[40] S. Siddiqi, Computing minimum-cardinality diagnoses by model relaxation, in: T. Walsh (Ed.), Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11), IJCAI/AAAI Press, 2011, pp. 1087–1092.

[41] T. Schaub, S. Thiele, Metabolic network expansion with ASP, in: P. Hill, D. Warren (Eds.), Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), Vol. 5649 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 312–326.

[42] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, P. Veber, Repair and prediction (under inconsistency) in large biological networks with answer set programming, in: Lin and Sattler [85], pp. 497–507.

[43] M. Banbara, T. Soh, N. Tamura, K. Inoue, T. Schaub, Answer set programming as a modeling language for course timetabling, Theory and Practice of Logic Programming 13 (4-5) (2013) 783–798.

[44] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, T. Schaub, Progress in clasp series 3, in: Calimeri et al. [84], pp. 368–383.

[45] M. Alviano, J. Romero, T. Schaub, Preference relations by approximation, in: Thielscher et al. [89], pp. 2–11.

[46] E. Di Rosa, E. Giunchiglia, M. Maratea, A new approach for solving satisfiability problems with qualitative preferences, in: M. Ghallab, C. Spyropoulos, N. Fakotakis, N. Avouris (Eds.), Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08), IOS Press, 2008, pp. 510–514.

[47] M. Alviano, C. Dodaro, J. Marques-Silva, F. Ricca, Optimum stable model search: Algorithms and implementation, Journal of Logic and Computation 30 (4) (2020) 863–897.

[48] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, F. Ricca, ASP and subset minimality: Enumeration, cautious reasoning and MUSes, Artificial Intelligence 320 (2023) 103931.

[49] C. Domshlak, E. Hüllermeier, S. Kaci, H. Prade, Preferences in AI: an overview, Artificial Intelligence 175 (7-8) (2011) 1037–1052.

[50] R. Brafman, C. Domshlak, Preference handling — an introductory tutorial, AI Magazine 30 (1) (2009) 58–86.

[51] J. Delgrande, T. Schaub, H. Tompits, K. Wang, A classification and survey of preference handling approaches in nonmonotonic reasoning, Computational Intelligence 20 (2) (2004) 308–334.

[52] G. Brewka, I. Niemelä, M. Truszczyński, Preferences and nonmonotonic reasoning, AI Magazine 29 (4) (2008) 69–78.

[53] T. Eiter, E. Erdem, H. Erdogan, M. Fink, Finding similar/diverse solutions in answer set programming, Theory and Practice of Logic Programming 13 (3) (2013) 303–359.

[54] Y. Lierler, An abstract view on optimizations in propositional frameworks, Theory and Practice of Logic Programming (2022) 1–29.

[55] A. Ensan, E. Ternovska, A language–independent framework for reasoning about preferences for declarative problem solving, in: A. Herzig, A. Popescu (Eds.), Frontiers of Combining Systems - 12th International Symposium, FroCoS, Proceedings, Vol. 11715 of Lecture Notes in Computer Science, Springer-Verlag, 2019, pp. 57–73.

[56] G. Brewka, M. Truszczyński, S. Woltran, Representing preferences among sets, in: M. Fox, D. Poole (Eds.), Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10), AAAI Press, 2010, pp. 273–278.

[57] W. Faber, M. Truszczynski, S. Woltran, Abstract preference frameworks — a unifying perspective on separability and strong equivalence, in: desJardins and Littman [87], pp. 297–303.

[58] S. Coste-Marquis, J. Lang, P. Liberatore, P. Marquis, Expressive power and succinctness of propositional languages for preference representation, in: Lin and Sattler [85], pp. 203–212.

[59] M. Huelsman, M.Truszczynski, Relating preference languages by their expressive power, in: R. Barták, F. Keshtkar, M. Franklin (Eds.), Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, 2022.

[60] A. Charalambidis, P. Rondogiannis, A. Troumpoukis, A logical characterization of the preferred models of logic programs with ordered disjunction, Theory and Practice of Logic Programming 21 (5) (2021) 629–645.

[61] M. Bernreiter, J. Maly, S. Woltran, Choice logics and their computational properties, in: Z. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI'21), ijcai.org, 2021, pp. 1794–1800.

[62] J. Lee, Z. Yang, Computing logic programs with ordered disjunction using asprin, in: Thielscher et al. [89], pp. 57–61.

[63] M. Balduccini, M. Gelfond, Logic programs with consistency-restoring rules, in: P. Doherty, J. McCarthy, M. Williams (Eds.), Proceedings of the International Symposium on Logical Formalization of Commonsense Reasoning (COMMONSENSE'03), 2003, pp. 9–18.

[64] T. Eiter, R. Kiesel, Weighted LARS for quantitative stream reasoning, in: De Giacomo et al. [90], pp. 729–736.

[65] N. Fraser, Applications of preference trees, in: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, IEEE Computer Society Press, 1993, pp. 132–136.

[66] X. Liu, M. Truszczynski, Reasoning with preference trees over combinatorial domains, in: T. Walsh (Ed.), Algorithmic Decision Theory - 4th International Conference, ADT, Proceedings, Vol. 9346 of Lecture Notes in Computer Science, Springer-Verlag, 2015, pp. 19–34.

[67] R. Booth, Y. Chevaleyre, J. Lang, J. Mengin, C. Sombattheera, Learning conditionally lexicographic preference relations, in: H. Coelho, R. Studer, M. Wooldridge (Eds.), Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10), IOS Press, 2010, pp. 269–274.

[68] J. Goldsmith, J. Lang, M. Truszczynski, N. Wilson, The computational complexity of dominance and consistency in cp-nets, Journal of Artificial Intelligence Research 33 (2008) 403–432.

[69] D. Bigot, B. Zanuttini, H. Fargier, J. Mengin, Probabilistic conditional preference networks, in: A. E. Nicholson, P. Smyth (Eds.), Proceedings of the Twenty Ninth Conference on Uncertainty in Artificial Intelligence (UAI 2013), AUAI Press, 2013, pp. 72–81.

[70] T. Guns, P. Stuckey, G. Tack, Solution dominance over constraint satisfaction problems, CoRR abs/1901.09125.

[71] G. Koçak, Ö. Akgün, T. Guns, I. Miguel, Exploiting incomparability in solution dominance: Improving general purpose constraint-based mining, in: De Giacomo et al. [90], pp. 331–338.

[72] L. Giordano, D. Dupré, ASP for minimal entailment in a rational extension of SROEL, Theory and Practice of Logic Programming 16 (5-6) (2016) 738–754.

[73] L. Giordano, D. Dupré, An ASP approach for reasoning in a concept-aware multipreferential lightweight DL, Theory and Practice of Logic Programming 20 (5) (2020) 751–766.

[74] L. Giordano, D. Dupré, Weighted conditional $EL^{\perp}$ knowledge bases with integer weights: an ASP approach, in: A. Formisano, Y. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens, N. Zhou (Eds.), Technical Communications of the Thirty seventh International Conference on Logic Programming (ICLP'21), Vol. 345 of EPTCS, 2021, pp. 70–76.

[75] L. Giordano, D. Dupré, An ASP approach for reasoning on neural networks under a finitely many-valued semantics for weighted conditional knowledge bases, Theory and Practice of Logic Programming 22 (4) (2022) 589–605.

[76] L. Bozzato, T. Eiter, R. Kiesel, Reasoning on multirelational contextual hierarchies via answer set programming with algebraic measures, Theory and Practice of Logic Programming 21 (5) (2021) 593–609.

[77] T. Eiter, R. Kiesel, Asp($\mathcal{AC}$): Answer set programming with algebraic constraints, Theory and Practice of Logic Programming 20 (6) (2020) 895–910.

[78] S. Rudolph, L. Schweizer, S. Tirtarasa, Justifications for description logic knowledge bases under the fixed-domain semantics, in: C. Benzmüller, F. Ricca, X. Parent, D. Roman (Eds.), Proceedings of the Second International Joint Conference on Rules and Reasoning (RuleML+RR'18), Vol. 11092 of Lecture Notes in Computer Science, Springer-Verlag, 2018, pp. 185–200.

[79] T. Gao, P. Fodor, M. Kifer, Paraconsistency and word puzzles, Theory and Practice of Logic Programming 16 (5-6) (2016) 703–720.

[80] M. Gebser, T. Guyet, R. Quiniou, J. Romero, T. Schaub, Knowledge-based sequence mining with ASP, in: R. Kambhampati (Ed.), Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16), IJCAI/AAAI Press, 2016, pp. 1497–1504.

[81] T. Lehtonen, J. Wallner, M. Järvisalo, Declarative algorithms and complexity results for assumption-based argumentation, Journal of Artificial Intelligence Research 71 (2021) 265–318.

[82] T. Lehtonen, J. Wallner, M. Järvisalo, An answer set programming approach to argumentative reasoning in the ASPIC+ framework, in: D. Calvanese, E. Erdem, M. Thielscher (Eds.), Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'21), AAAI Press, 2020, pp. 636–646.

[83] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: Carro and King [86], pp. 2:1–2:15.

[84] F. Calimeri, G. Ianni, M. Truszczyński (Eds.), Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15), Vol. 9345 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2015.

[85] F. Lin, U. Sattler (Eds.), Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10), AAAI Press, 2010.

[86] M. Carro, A. King (Eds.), Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16), Vol. 52 of Open

Access Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[87] M. desJardins, M. Littman (Eds.), Proceedings of the Twenty-seventh National Conference on Artificial Intelligence (AAAI'13), AAAI Press, 2013.

[88] P. Cabalar, T. Son (Eds.), Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13), Vol. 8148 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2013.

[89] M. Thielscher, F. Toni, F. Wolter (Eds.), Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'18), AAAI Press, 2018.

[90] G. De Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, J. Lang (Eds.), Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20), IOS Press, 2020.

[91] V. Lifschitz, H. Turner, Splitting a logic program, in: Proceedings of the Eleventh International Conference on Logic Programming, MIT Press, 1994, pp. 23–37.

[92] F. Buccafurri, N. Leone, P. Rullo, Enhancing disjunctive datalog by constraints, IEEE Transactions on Knowledge and Data Engineering 12 (5) (2000) 845–860.

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 $X, Y \subseteq \mathcal{A}$ and the program $P_s \cup H(X) \cup H(Y)'$ is satisfiable (by Definition 1) iff (X, Y) belongs to $\{(X, Y) \mid X, Y \subseteq \mathcal{A}, P_s \cup H(X) \cup H(Y)'$ 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 $X = \{a \mid \mathit{holds}(a) \in Z\}$ and $Y = \{a \mid \mathit{holds}'(a) \in Z\}$.
4. There is a stable model Z of $P_s \cup C(\mathcal{A}) \cup C'(\mathcal{A})$ such that $X = \{a \mid \mathit{holds}(a) \in Z\}$ and $Y = \{a \mid \mathit{holds}'(a) \in Z\}$.
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

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

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

$$H(X) \subseteq \{\mathit{holds}(a) \mid \mathit{holds}(a) \in Z\}. \tag{A.2}$$

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

$$H(X) \supseteq \{\mathit{holds}(a) \mid \mathit{holds}(a) \in Z\}$$

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

$$H(X) = \{\mathit{holds}(a) \mid \mathit{holds}(a) \in Z\}.$$

By a simple rewriting, we obtain that

$$X = \{a \mid \mathit{holds}(a) \in Z\}. \tag{A.3}$$

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

$$Y = \{a \mid \mathit{holds}'(a) \in Z\}. \tag{A.4}$$

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

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

$$P_s \cup C(\mathcal{A}) \cup C'(\mathcal{A}) \tag{A.5}$$

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

> Z is a stable model of (A.5) iff Z is a stable model of $P_s \cup Z_1$ for some stable model Z_1 of $C(\mathcal{A}) \cup C'(\mathcal{A})$. (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 is a stable model of (A.5) iff Z is a stable model of $P_s \cup H(V) \cup H(W)'$ for some sets $V, W \subseteq \mathcal{A}$. (A.7)

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

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

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 $V = \{a \mid \mathit{holds}(a) \in Z\}$, $W = \{a \mid \mathit{holds}'(a) \in Z\}$, $X = \{a \mid \mathit{holds}(a) \in Z\}$ and $Y = \{a \mid \mathit{holds}'(a) \in Z\}$. (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 is a stable model of Q iff $M = M_1 \cup M_2$ 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 $P_s \cup H(M_1) \cup H(X)'$. (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 holds or 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, $R(\mathcal{A})$, 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 $M_1 \cup R(\mathcal{A})$, and M is a stable model of $M_3 \cup P_s \cup H(X)'$.

It is easy to see that the program $M_1 \cup R(\mathcal{A})$ 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 is a stable model of Q iff there is a set of atoms M_1 such that M_1 is a stable model of P, and M is a stable model of $M_1 \cup H(M_1) \cup P_s \cup H(X)'$. (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 $H(M_1) \cup P_s \cup H(X)'$, and therefore M has the form $M_1 \cup M_2$ for some stable model M_2 of $H(M_1) \cup P_s \cup H(X)'$. 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 $X \subseteq \mathcal{A}$. 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 $P_s \cup H(Y) \cup H(X)'$ is satisfiable.
4. There are some sets of atoms Y and Z such that is Y is a stable model of P, and Z is a stable model of $P_s \cup H(Y) \cup H(X)'$.
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 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 $X \subseteq \mathcal{A}$, 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 $X \subseteq \mathcal{A}$, by (A.9) 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 $P_s \cup H(M_1) \cup H(X)'$. (A.11)

Given that the atoms of $\mathcal{A}$ do not occur in the program $P_s \cup H(M_1) \cup H(X)'$, the set $M_2 \cap \mathcal{A}$ is empty, and since we have that $Y = M_1 \cup M_2$ by (A.11), we can conclude that

$$Y \cap \mathcal{A} = M_1. \tag{A.12}$$

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 $Y \cap \mathcal{A}$ is a stable model of P by (A.11) and (A.12), and that $Y \cap \mathcal{A} \succ_s X$ by (A.12) and (A.13).

PROOF. (Proposition 4, sketch)
Let $X, Y \subseteq \mathcal{A}$, then by $Q(X, Y)$ we denote the logic program

$$E_{subset} \cup F_s \cup G \cup H(X) \cup H(Y)'.$$

The set G consists of the set of rules defining $holds$ and $holds'$ 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 $X, Y \subseteq \mathcal{A}$ the program $Q(X, Y)$ is satisfiable iff $X \succ_s Y$. (A.14)

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

$$E_{subset} \cup F_s \cup G' \cup H(X) \cup H(Y)'.$$

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). (A.15)

Observe that the program $Q'(X, Y)$ consists of a set of facts along with the rule E_{subset}, that defines the predicate $better$, 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 $better$. (A.16)

More precisely, below we prove that

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

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

M violates the constraint (9) iff $better(s) \notin O$.

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 $N \cup \{better(s)\}$ if $X \succ_s Y$, and it has no stable models otherwise. This implies (A.14) and completes the main part of the proof.

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

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

- $preference(p, subset) \in N$ iff $s = p$;
- there are some terms t_1, t_2 and t_3 such that $preference(p, t_1, t_2, for(x), t_3) \in N$ iff $x \in E$;
- for every $x \in E$ 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. $X \not\models x$, $Y \models x$, 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 $\{better(s)\}$ 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

$$(X, Y) \in subset(E),$$

which, given the form of s, is equal to

$$X \succ_s Y.$$

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

that we prove next using the notation introduced in Proposition 4.
Given the form of $E_{more(weight)}$, O is the set of atoms of the form $better(p)$ satisfying these conditions:

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

- the integer terms $-w$ for every pair of terms (w, x) such that $holds'(x) \in N$ and there are some terms t_1 and t_2 such that $preference(p, t_1, t_2, for(x), w) \in N$.

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:

- $preference(p, more(weight)) \in N$ iff $s = p$;
- there are terms t_1 and t_2 such that $preference(p, t_1, t_2, for(x), w) \in N$ 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 $\{better(s)\}$ 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 that 0 iff

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

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

$$(X, Y) \in more(weight)(E),$$

which, given the form of s, is equal to

$$X \succ_s Y.$$

Hence, (A.17) follows: O is the set of atoms $\{better(s)\}$ 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 $(Y_0, Y_1, \dots)$ 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

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

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

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

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 $(Y_0, Y_1, \dots)$ 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 $(Y_0, \dots, Y_n)$ 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 $\perp$, or a stable model M of P. The first case is not possible because, by Lemma 1, if Y_i is $\perp$ 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 $\perp$, 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

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

By item 3 we have that Y_n is $\perp$, 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 $\perp$, 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 $Y_0 = \perp$, hence Y has the value $\perp$ in Line 1, and the algorithm returns $\perp = Y_0$ 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 $\perp$, 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 $\succeq_s$-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 is a stable model of P iff N is a stable model of Q that satisfies all constraints in C. (A.20)

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

> Q has a unique stable model, that we denote by M. (A.21)

Consider the next statement, that we will use afterwards:

> M satisfies $\mathit{body}(r)$ for some $r \in C$. (A.22)

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

> P is satisfiable iff (A.22) does not hold, (A.23)

and

> $\overline{P}$ is satisfiable iff (A.22) holds. (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 $\mathit{body}(r)$ 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 $\{u \leftarrow \mathit{body}(r) \mid r \in C\}$. 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 $\{u \leftarrow \mathit{body}(r) \mid r \in C\}$ such that $M \cup \{u\}$ satisfies $\mathit{body}(r)$. Finally, statement 6 is equivalent to statement 7 because the atom u does not occur in any $\mathit{body}(r)$ for $r \in C$.

PROOF. (Proposition 8)

We prove the proposition by showing that, for all $X, Y \subseteq \mathcal{A}$, the following statements are equivalent:

1. $X \not\succ_s Y$.
2. $P_s \cup H(X) \cup H(Y)'$ is not satisfiable.
3. $\overline{P_s \cup H(X) \cup H(Y)'}$ is satisfiable.
4. $\overline{P_s} \cup H(X) \cup H(Y)'$ 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 $\overline{P_s \cup H(X) \cup H(Y)'}$ and $\overline{P_s} \cup H(X) \cup H(Y)'$ are the same.

PROOF. (Proposition 9)

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

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

> M is a stable model of Q iff $M = M_1 \cup M_2$ 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.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 $X \subseteq \mathcal{A}$. 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 $\mathcal{A}$ do not occur in the program $\overline{P_s} \cup H(X) \cup H(M_1)'$, the set $M_2 \cap \mathcal{A}$ is empty, and since we have that $Y = M_1 \cup M_2$ by (A.26), we can conclude that

$$Y \cap \mathcal{A} = M_1. \tag{A.27}$$

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

$$X \not\succ_s M_1. \tag{A.28}$$

Putting all together, we can prove that $Y \cap \mathcal{A}$ is a stable model of P by (A.26) and (A.27), and that $X \not\succ_s Y \cap \mathcal{A}$ 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 $X_i \subseteq \mathcal{A}$. (A.29)

Observe that the set $\mathcal{X}$ 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 $\bot$. If (i), then Y is assigned the value $M \cap \mathcal{A}$. Hence, $Y \subseteq \mathcal{A}$ holds, and X will be assigned in Line 6 some subset of $\mathcal{A}$. If (ii), then Y is assigned the value $\bot$. 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 $\mathcal{A}$, and (A.29) holds.

Step 2. Let $\{X_1, \ldots, X_n\}$ be the value of $\mathcal{X}$ in Line 2, I be $\{1, \ldots, n\}$, and Q denote the program $P \cup \bigcup_{X_i \in \mathcal{X}} \left(N_{X_i} \cup (\overline{P_s} \cup H(X_i))^i \cup R'(\mathcal{A})^i\right)$ 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, …, 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 $X_i \in \mathcal{X}$. (A.30)

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
- $Q_2 = \bigcup_{X_i \in \mathcal{X}} \big((\overline{P_s} \cup H(X_i))^i \cup R'(\mathcal{A})^i \big)$.

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 splitted in two parts:

- $Q_3 = \bigcup_{X_i \in \mathcal{X}} R'(\mathcal{A})^i$, 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 $\mathcal{A}$ or have the form $holds'(a)^i$ for some $a \in \mathcal{A}$ 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 $holds'(a)^i$ for some $a \in \mathcal{A}$ and $i \in I$, given that no atoms over $holds'$ 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 $\bigcup_{X_i \in \mathcal{X}} H(M_0)'^i \cup Q_4$. (A.31)

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

M'' is a stable model of $\bigcup_{X_i \in \mathcal{X}} H(M_0)'^i \cup Q_4$ 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 $(\overline{P_s} \cup H(X_i) \cup H(M_0)')^i$ 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 $(\overline{P_s} \cup H(X_i) \cup H(M_0)')^i$ for every $X_i \in \mathcal{X}$. (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

$$Y = M \cap \mathcal{A}. \tag{A.33}$$

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 $X_i \in \mathcal{X}$. (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:

$$M_0 \cap \mathcal{A} = M_0. \tag{A.35}$$

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:

$$M_i \cap \mathcal{A} = \emptyset \text{ for } X_i \in \mathcal{X}. \tag{A.36}$$

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

$$Y = M_0. \tag{A.37}$$

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 $(\overline{P_s} \cup H(X_i) \cup H(Y)')^i$ for every $X_i \in \mathcal{X}$ are satisfiable. This implies that the programs $\overline{P_s} \cup H(X_i) \cup H(Y)'$ for every $i \in I$ are satisfiable, and then by (A.29) and Proposition 9 it follows that $X_i \not\succ_s Y$ 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 $X_i \not\succ_s Y$ 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 $\bigcup_{X_i \in \mathcal{X}} N_{X_i}$. (A.38)

Given that $X_i \not\succ_s Y$ for all $i \in I$, by Proposition 9 and (A.29), it follows that the programs $\overline{P_s} \cup H(X_i) \cup H(Y)'$ for every $i \in I$ are satisfiable. This implies that the programs $(\overline{P_s} \cup H(X_i) \cup H(Y)')^i$ for every $X_i \in \mathcal{X}$ are satisfiable. Then, let M_i be the stable model of $(\overline{P_s} \cup H(X_i) \cup H(Y)')^i$ 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 $(Y_{i_0}, Y_{i_1}, \dots)$ 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 $Y_{i_j} \succ_s Y_{i_{j-1}}$ 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 $(Y_{i_0}, Y_{i_1}, \dots)$ instead of a trace $(Y_0, Y_1, \dots)$, 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 $(Y_{i_0}, \dots, Y_{i_n})$ for some integer $n \geq 0$, and*

1. *Y_{i_j} is a stable model of P for $0 \leq j < n$*
2. *$Y_{i_j} \succ_s Y_{i_{j-1}}$ for $0 < j < n$*
3. *$Y_{i_n} = \bot$*
4. *$Y_{i_{n-1}}$ is a $\succeq_s$-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

$$Y_{i_0} \text{ is a stable model of } P \text{ such that } Y_{i_0} \neq X_j \text{, and } X_j \not\succ_s Y_{i_0} \text{ for all } X_j \in \mathcal{X}_{i-1}. \tag{A.39}$$

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

$$n > 0. \tag{A.40}$$

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

$$Y_{i_0} \text{ is } \bot, \tag{A.41}$$

and

$$\text{there is no stable model } Y \text{ of } P \text{ such that } Y \neq X_j \text{, and } X_j \not\succ_s Y \text{ for all } X_j \in \mathcal{X}_{i-1}. \tag{A.42}$$

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

$$n = 0. \tag{A.43}$$

We also have that (A.42) implies that

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

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

$$Y_{i_j} \succ_s Y_{i_0}. \tag{A.45}$$

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 $X_k = Y_{i_j}$ or $X_k \succ_s Y_{i_j}$ 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 $\mathcal{A}$ 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 $\Pi^p_{max(\{i,j\})}$.*

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 $P \cup P_s \cup R(\mathcal{A}) \cup H(X)'$ is satisfiable. Since that program is of class $max(\{i,j\})$, we can conclude that the problem of deciding whether X is not a $\succeq_s$-preferred stable model of P is in $\Sigma^p_{max(\{i,j\})}$. Then, the lemma follows from the fact that this is the complement of the Optimality problem.

Lemma 6. *Given a base program P over $\mathcal{A}$ 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 $\Pi^p_{max(\{i,j\})}$-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 Σ^p_i and Σ^p_j-hard. Hence, Non Optimality is $\Sigma^p_{max(\{i,j\})}$-hard, and the lemma follows from the fact that Non Optimality is the complement of the Optimality problem.

Non Optimality is Σ^p_i-hard. Given a logic program P of class i, the problem of deciding whether P has some stable model is Σ^p_i-complete. We prove Σ^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 α 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 $\leftarrow \mathit{holds}(\alpha)$ and $\leftarrow \neg\mathit{holds}'(\alpha)$. 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 α are better than those with α. 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 Σ^p_j-hard. Given a logic program Q of class j, the problem of deciding whether Q has some stable model is Σ^p_j-complete. We prove Σ^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 α 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 $\leftarrow \mathit{holds}(\alpha)$ and $\leftarrow \neg\mathit{holds}'(\alpha)$ as well as all the rules of Q. Let EQ be $\{(\emptyset, \emptyset), (\{\alpha\}, \{\alpha\})\}$, and consider the preference relation $\succeq$ such that $\succeq = \{(\{\}, \{\alpha\})\} \cup EQ$ if Q has some stable model, and $\succeq = EQ$ 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 $\succeq = \{(\{\}, \{\alpha\})\} \cup EQ$ 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 $\succ = EQ$ and therefore there is no stable model X

of P such that $X \succ \{\alpha\}$.

Lemma 7. *Given a base program P over $\mathcal{A}$ 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 $F\Sigma^p_{max(\{i,j\})+1}$.*

PROOF. We prove the lemma by showing a nondeterministic Turing machine, with an oracle for any problem in $\Sigma^p_{max(\{i,j\})}$, that can compute a solution to the Model Finding problem (if it exists) in polynomial time. The machine first guesses a set $X \subseteq \mathcal{A}$. 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 Π^p_{i-1} (see Table III of [3]) and the second is in $\Pi^p_{max(\{i,j\})}$ (Lemma 5), hence both can be performed by the oracle for $\Sigma^p_{max(\{i,j\})}$ problems.

Lemma 8. *Given a base program P over $\mathcal{A}$ 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 $F\Delta^p_{i+1}$-hard.*

PROOF. The problem of computing a stable model of a logic program of class $i \in \{1,2\}$ with optimization statements is $F\Delta^p_{i+1}$-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 Δ^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 $\mathcal{A}$ 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 $\Sigma^p_{max(\{i,j\})+1}$.*

PROOF. The proof is analogous to the proof of Lemma 7, except for the fact that the machine only guesses sets $X \subseteq \mathcal{A}$ that contain a, 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 $\mathcal{A}$ be a set of atoms. A logic programming abduction problem (LPAP)

$\mathcal{P}$ over $\mathcal{A}$ is a tuple $\langle H, M, P\rangle$ where $H \subseteq \mathcal{A}$ is a finite set of hypotheses, $M \subseteq \mathcal{A} \cup \{\neg a \mid a \in \mathcal{A}\}$ is a finite set of manifestations, and P is a propositional logic program over $\mathcal{A}$. 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 $\mathcal{A}$ and M is a set of literals over $\mathcal{A}$, then $P \models M$ iff there is some stable model X of P such that X satisfies M. Let $\mathcal{P} = \langle H, M, P\rangle$ be a LPAP and let $S \subseteq H$. Then, S is a solution to $\mathcal{P}$ if $P \cup S \models M$, and S is also a $\subseteq$-solution to $\mathcal{P}$ if there is no solution S' to $\mathcal{P}$ such that $S' \subset S$. We say that a hypothesis $h \in H$ is relevant for $\mathcal{P}$ if there is some solution S to $\mathcal{P}$ such that $h \in S$, and it is $\subseteq$-relevant for $\mathcal{P}$ if there is some $\subseteq$-solution S to $\mathcal{P}$ such that $h \in S$. Let $\subseteq$-Relevance be the problem of deciding if a given hypothesis is $\subseteq$-relevant for a LPAP $\mathcal{P} = \langle H, M, P\rangle$. This problem is Σ_2^p-complete if P is normal (part (4) of Theorem 15 of [32]) and it is Σ_3^p-complete if P is disjunctive (Theorem 23 of [32]). In other words, $\subseteq$-Relevance is Σ_{i+1}^p-complete if P is of class $i \in \{1, 2\}$. We can represent the solutions to $\mathcal{P}$ by the following logic program $lp(\mathcal{P})$:

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

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 $\mathcal{P}$ and the stable models of $lp(\mathcal{P})$:

> if S is a solution to $\mathcal{P}$ then there is at least one stable model X of $lp(\mathcal{P})$ such that $X \cap H = S$, (A.46)

and

> if X is a stable model of $lp(\mathcal{P})$ then $X \cap H$ is a solution to $\mathcal{P}$. (A.47)

Note that $lp(\mathcal{P})$ is normal if P is normal, and it is disjunctive if P is disjunctive.

Lemma 10. *Given a base program P over $\mathcal{A}$ 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 Σ_{j+1}^p-hard.*

PROOF. We reduce the problem $\subseteq$-Relevance of deciding whether a given hypothesis h is $\subseteq$-relevant for a LPAP $\mathcal{P} = \langle H, M, P\rangle$ over $\mathcal{A}$, where P is of class j. As we have seen before, this problem is Σ_{j+1}^p-complete.

Let α be a new atom that does not belong to $\mathcal{A}$. Consider the following logic program Q:

$$\{\{\alpha\} \leftarrow\} \cup \{\{a\} \leftarrow \alpha \mid a \in \mathcal{A}\} \cup \{h \leftarrow \alpha\}.$$

It is easy to see that

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

Consider also the relation $\succ$ over $\mathcal{A} \cup \{\alpha\}$ such that $X \succ Y$ iff $X = \emptyset$ and $Y = W \cup \{\alpha\}$ for some set $W \subseteq \mathcal{A}$ such that:

1. either W is not a stable model of $lp(\mathcal{P})$,
2. or $W \cap H$ is not a $\subseteq$-solution to $\mathcal{P}$.

Let $\succeq$ be $\succ \cup \{(X.X) \mid X \subseteq \mathcal{A} \cup \{\alpha\}\}$. 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 $\mathcal{P}$.
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 $\mathcal{X} = \{\emptyset\}$ it holds that $Y \neq X$ and $X \not\succ Y$ for all $X \in \mathcal{X}$.

From statement 1 to statement 2. If statement 1 holds, then by definition

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

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

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

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 $lp(\mathcal{P})$ by (A.50), and $W \cap H$ is a $\subseteq$-solution to $\mathcal{P}$ by (A.49) and (A.50). Hence, Y is a $\succeq$-preferred stable model of Q, and statement 2 follows given that $\alpha \in Y$.

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 $W \subseteq \mathcal{A}$ 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 $\mathcal{P}$. 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 $\mathcal{P}$ such that $h \in W \cap H$, and this implies statement 1.

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 $\emptyset \not\succ Y$.

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 $\emptyset$ is a stable model of Q, from which we can conclude that $\emptyset \not\succ Y$. From 5 to 4, $Y \neq \emptyset$ together with (A.48) imply that $\alpha \in Y$, and if $\emptyset \not\succ Y$ 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 $X, Y \subseteq \mathcal{A} \cup \{\alpha\}$ 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 $W \subseteq \mathcal{A}$ such that:

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

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

The first set enforces that $X = \emptyset$, and the second set enforces that Y has the form $W \cup \{\alpha\}$. We immediately have that $W \subseteq \mathcal{A}$ since $\succ$ is defined over $\mathcal{A} \cup \{\alpha\}$. The set I also contains the following rules, that use the atoms c_i for $i \in \{1,2,3\}$ to represent condition i:

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

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 $r \in \mathit{lp}(\mathcal{P})$ of the form

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

one rule of the form

$$\begin{aligned} ok \leftarrow & \neg \mathit{holds}'(a_1), \dots, \neg \mathit{holds}'(a_m), \\ & \quad \mathit{holds}'(a_{m+1}), \dots, \mathit{holds}'(a_n), \neg \mathit{holds}'(a_{n+1}), \dots, \neg \mathit{holds}'(a_o), c_1 \end{aligned}$$

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 $\mathit{lp}(\mathcal{P})$. Note that the choice rules of $\mathit{lp}(\mathcal{P})$ are not considered for C_1.

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

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

one rule of the form

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

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

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

the set C_2 also contains one rule of the form

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

where the rule is translated to a normal one and the atom $\mathit{holds}'(a_0)$ 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 $\mathit{lp}(\mathcal{P})$ 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:

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

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

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

The set of rules R_3 contains all the rules of $\mathit{lp}(\mathcal{P})$ extended with the additional atom c_3 in the body. This set of rules generates stable models W' of $\mathit{lp}(\mathcal{P})$ 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:

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

All in all, this combination of rules ensures that the program $R \cup H(X) \cup H'(Y)$ 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 $\mathcal{A}$ 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 $\Sigma^p_{\mathit{max}(\{i,j\})+1}$-hard.*

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

Σ^p_{i+1}-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 Σ^p_{i+1}-complete if P is of class $i \in \{1, 2\}$. Consider the logic program $\mathit{lp}(\mathcal{P})$ of class i, that generates the solutions to $\mathcal{P}$.

Consider also the preference statement $\#preference(s, subset)H$ 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 $\mathcal{P}$ correspond one-to-many to the $\succeq_s$-preferred stable models of $lp(\mathcal{P})$. Recall that the problem $\subseteq$-Relevance amounts to deciding whether there is some $\subseteq$-solution S to $\mathcal{P}$ 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 $lp(\mathcal{P})$ 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 $\mathcal{A}$ 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 $\Sigma^p_{max(\{i,j+1\})}$.*

Proof. We prove the lemma by showing a nondeterministic Turing machine, with an oracle for any problem in $\Sigma^p_{max(\{i-1,j\})}$, that can solve the Non Dominance problem in polynomial time. The machine first guesses a set $Y \subseteq \mathcal{A}$. Then it checks whether Y is a stable model of P, and whether for all $X \in \mathcal{X}$ it holds that $X \neq Y$ and $X \not\succ_s Y$. The first check is in Π^p_{i-1} (see Table III of [3]). In the second check, for each $X \in \mathcal{X}$, the condition $X \neq Y$ can be decided in polynomial time, and the condition $X \not\succ_s Y$ is in Π^p_j. 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 Σ^p_j (see again Table III of [3]) and its complement is in Π^p_j. Since the set $\mathcal{X}$ is part of the input, we can conclude that the second check on all the elements of $\mathcal{X}$ is also in Π^p_j. Putting all together, it follows that the checks are in $\Pi^p_{max(\{i-1,j\})}$, and therefore they can be performed by the oracle for $\Sigma^p_{max(\{i-1,j\})}$ problems.

Lemma 13. *Given a base program P over $\mathcal{A}$ 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 $\Sigma^p_{max(\{i,j+1\})}$-hard.*

Proof. Recall that given a logic program Q of class i, the problem of deciding whether Q has some stable model is Σ^p_i-complete. Then, we prove Σ^p_i-hardness by reducing that problem to Non Dominance where the set $\mathcal{X}$ is empty. This implies $\Sigma^p_{max(\{i,j+1\})}$-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 Σ^p_{j+1}-hardness for the cases where $i = 1$ and $j \in \{1, 2\}$, from which we also have Σ^p_{j+1}-hardness for the case where $i = 2$ and $j = 2$. This implies $\Sigma^p_{max(\{i,j+1\})}$-hardness for the cases where $i = 1$ and $j \in \{1, 2\}$, and where $i = 2$ and $j = 2$. Putting all together, we have $\Sigma^p_{max(\{i,j+1\})}$-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.