"Are Preferences Giving You a Headache?"
"Take asprin!"

Gerhard Brewka[1] James Delgrande[2] Javier Romero[3] Torsten Schaub[3]

[1]Universität Leipzig [2]Simon Fraser University [3]Universität Potsdam

Abstract. In this paper we introduce *asprin*[1], a general, flexible, and extensible framework for handling preferences among the stable models of a logic program. 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 give algorithms for computing one, or all, optimal stable models of a logic program. Notably, the algorithms depend on the complexity of the dominance tests and make use of incremental answer set solving technology.

1 Introduction

Preferences are pervasive. The identification of preferred, or optimal, solutions is indispensable in many real-world applications. Often this involves the combination of various qualitative and quantitative preferences. Although preferences have been widely studied in Answer Set Programming (ASP; [2]) in various contexts (cf. [3]), today's ASP systems are limited to optimization statements over sum or count aggregates, as given by $\#\mathit{minimize}$ statements or weak constraints [4, 5].

We address this shortcoming and present *asprin*, a general and flexible framework for implementing preferences among the stable models of logic programs. Our framework captures numerous existing approaches to preference. Moreover, it allows for an easy implementation of new or extended approaches to preference handling in a uniform setting. Our framework builds upon recent control capabilities for incremental ASP solving. First, this technology allows us to direct the search for specific preferred solutions without modifying the ASP solver. Second, its incrementality significantly reduces redundancies found in an iterated setting. Finally, this technology paves the way for the high customizability of our framework by offering an implementation of preferences via ordinary ASP encodings.

From an abstract point of view, we are interested in distinguishing the *preferred* stable models of a logic program given an underlying preference relation. That is, we determine a strict partial order $\succ$ on the stable models of a logic program P, representing a *preference relation* such that $X \succ Y$ means that X is preferred to Y. Accordingly, a stable model X of P is $\succ$*-preferred*, if there is no other stable model Y such that $Y \succ X$. In other words, X is not $\succ$*-dominated* by any stable model Y. A *preference type* is a class of preference relations. Restricting preferences to (strict) partial orders has the advantage that a satisfiable program has preferred stable models.

[1] *asprin* stands for "*AS*P for *Pr*eference handl*in*g".

As a motivating example, consider the following statements:

$$\#\mathit{preference}(\mathit{costs}, \mathit{less}(\mathit{weight}))\{40 : \mathit{sauna}, 70 : \mathit{dive}\} \quad (1)$$
$$\#\mathit{preference}(\mathit{fun}, \mathit{subset})\{\mathit{sauna}, \mathit{dive}, \mathit{hike}, \neg \mathit{bunji}\} \quad (2)$$
$$\#\mathit{preference}(\mathit{temps}, \mathit{aso})\{\mathit{dive} > \mathit{sauna} \parallel \mathit{hot}, \mathit{sauna} > \mathit{dive} \parallel \neg \mathit{hot}\} \quad (3)$$
$$\#\mathit{preference}(\mathit{all}, \mathit{pareto})\{\mathit{name}(\mathit{costs}), \mathit{name}(\mathit{fun}), \mathit{name}(\mathit{temps})\} \quad (4)$$
$$\#\mathit{optimize}(\mathit{all})$$

The keyword $\#\mathit{preference}$ declares a preference while $\#\mathit{optimize}$ is the optimization directive. A preference relation has an identifier, a type, and a set of arguments. The arguments make use of weights (eg. 40), rankings (via $>$), conditionalization (via $\parallel$), and naming (via name). Terms like "*fun*" and "*less(weight)*" are user-defined preference types and relations, respectively, similar to a term or atom in an ASP encoding. While their implementation is given in terms of *preference programs*, their semantics, that is, the underlying preference type or relation, lies with the user. We illustrate this via some preference types and relations taken from the preference library of our system.

As with $\#\mathit{minimize}$ statements [4], preference types are instantiated to preference relations by concrete arguments. Thus, applying $\#\mathit{minimize}$ to the set $\{40 : \mathit{sauna}, 70 : \mathit{dive}\}$ instantiates the (implicit) preference type to a preference relation which prefers stable models with a smaller sum over those with a greater one. This is similar to (1), except that we make the type and relation, viz. *less(weight)* and *costs*, explicit. Also, $\#\mathit{minimize}$ statements are always subject to optimization, while this must be explicitly stated by an $\#\mathit{optimize}$ directive in our approach.

The combinations of problems, programs, and preference types result in a wide range of possible computational complexities. For uniformly addressing this range of problems, we use ASP's meta interpretation capacities, and design computations in a branch-and-bound manner by building upon the control capacities for incremental ASP solving. The basic building block is the *preference program*, which allows for testing whether a current optimal solution is dominated by a new one. While a simple arithmetic comparison is sufficient for $\#\mathit{minimize}$, a much more elaborate technique is needed for complex preferences, such as the one shown above. We address this intricacy by means of ASP: we encode the elementary dominance tests either as a standard or a saturation-based ASP encoding, depending on the respective problem complexity.

We assume a general familiarity with ASP. For a comprehensive introduction, we refer the reader to [6]. Our notation also follows that in [6]. Our ASP encodings rely upon the new ASP language standard [7]. Other notation is introduced when first used.

2 Expressing Preferences

We provide a simple generic preference language for expressing a wide range of preference relations. To keep our framework open for extensions, we do not fix a predefined set of preferences. Rather we give some examples of how well-known preferences can be expressed and implemented. Many of them are included in *asprin*'s preference library which provides basic building blocks for defining new preferences.

Syntax. Let $\mathcal{A}$ be a fixed alphabet. A *weighted (propositional) formula* is of the form[2] $w_1, \dots, w_l : \phi$ where each w_i is a term and ϕ is a Boolean expression over $\mathcal{A}$ with logical connectives $\top$, $\neg$, $\wedge$, and $\vee$. We write ϕ whenever $l = 0$. 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).

A *preference element* is of the form[3]

$$\Phi_1 > \cdots > \Phi_m \parallel \Phi \qquad (5)$$

where each Φ_r is a set of weighted formulas for $r = 1, \dots, m$ and $m \geq 1$, and Φ is a non-weighted formula. 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 pre-orders (in addition to using weights). For convenience, we usually drop the surrounding braces of such sets and omit "$\parallel \Phi$" if Φ is tautological. Also, we drop all "$>$" if $m = 1$. Hence $a, \neg b, c$ stands for $\{a, \neg b, c\} \parallel \top$, and $\{a, \neg b\} > c \parallel \neg d$ stands for $\{a, \neg b\} > \{c\} \parallel \neg d$.

A *preference statement* is of the form

$$\#preference(s, t)\{e_1, \dots, e_n\} \qquad (6)$$

where s and t are ground terms giving the preference name and its type, respectively, and each e_j is a preference element. In what follows, we often identify a preference statement with its identifier s and refer to its type by t_s. The preference type determines the set of admissible preference elements. For instance, $subset$ in (2) is restricted to (unweighted) literals, while aso in (3) takes general preference elements. Analogously, a preference type may or may not require naming atoms, depending on whether it has a *composite* or *primitive* nature. For instance, the preference type $subset$ in (2) is primitive, while $pareto$ in (4) is composite.

As described in the next section, each preference statement s is associated with a strict partial order $\succ_s$. The relation $\succ_s$ is of the preference type denoted by t_s. If s is not subject to optimization, it may serve as an auxiliary element in a composite preference statement. To this end, some preference types may belong to a preference library, and provide definitions of the non-strict, equal, and inverse variants of the type at hand. In such a case, a naming atom $name(s)$ provides the importing preference statement also with the non-strict, equal, and inverse counterparts of $\succ_s$, namely, $\succeq_s$, $=_s$, and $\prec_s$, $\preceq_s$. For instance, in (1)-(4), the preference relation $\succ_{all}$ relies on the auxiliary relations $\succ_{costs}$ and $\succeq_{costs}$, among others. On the implementation side, this is reflected by the distinction between a *preference program* and a *preference module* (see Section 3). The former is importing preference implementations, while the latter is being imported.

A set of preference statements is accompanied by a single *optimization directive* $\#optimize(s)$, telling a solver to restrict its reasoning mode to the preference relation declared by s. A set of preference statements S is

- *closed*, if $s \in S$ whenever $name(s)$ occurs in S, and

[2] This syntax follows that of *aggregate elements* [7, Section 1.3].
[3] This syntax follows the one in [8].

- *acyclic*, if the dependency relation induced among preference statements in S by naming atoms is acyclic (no $s \in S$ refers directly or indirectly to itself via naming).

A *preference specification* is a set of preference statements S and a single directive $\#optimize(s)$ such that S is an acyclic and closed, and $s \in S$. We call s the primary preference statement in S and refer to all statements in $S \setminus \{s\}$ as secondary ones.
Semantics. A preference statement as in (6) declares a preference relation obtained by instantiating its preference type with a specific interpretation of the preference elements. More formally, a preference type, t, is a function, mapping a set of preference elements, E, to a preference relation, that is,

$$t : E \mapsto \{(X, Y) \mid def_t(E, X, Y), X, Y \subseteq \mathcal{A}\}$$

where $def_t(E, X, Y)$ holds iff X is preferred to Y in the relation obtained by applying t to E; and $dom(t)$ is the domain of t fixing admissible preference elements for t. [4] Accordingly, a preference statement $\#preference(s,t)E$ is said to be admissible, if $E \subseteq dom(t)$. In what follows, we assume that all preference statements (and so specifications) are admissible. Note that each preference type comes implicitly with a recipe for interpreting its preference elements, no matter how these elements are determined. These details must be fixed and controlled for each type t by def_t and $dom(t)$.

As an example, we can define the preference type $less(cardinality)$ by letting

- $def_{less(cardinality)}(E, X, Y) = |\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$ and
- $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$, for $\mathcal{P}(S)$ the power set of S.

Our approach centers on the implementation of $def_t(E, X, Y)$, as described in Section 3. The resulting algorithmic framework developed in Section 4 requires that the induced preference relations be strict partial orders. We can still define strict preference relations in terms of non-strict or other auxiliary preference relations, so long the preference relation subject to optimization is irreflexive and transitive. Let us illustrate this with some examples from *asprin*'s preference library. First, we provide some examples of primitive types and then show how composite types can be expressed via naming. In turn, we illustrate below how preference types are used for defining specific preference relations.

For two further examples, in *asprin* the preference type

- $more(weight)$ is defined by
  - $def_{more(weight)}(E, X, Y) = \sum_{(w:\ell) \in E, X \models \ell} w > \sum_{(w:\ell) \in E, Y \models \ell} w$
  - $dom(more(weight)) = \mathcal{P}(\{w : a, w : \neg a \mid w \in \mathbb{Z}, a \in \mathcal{A}\})$; and
- $subset$ is defined by
  - $def_{subset}(E, X, Y) = \{\ell \in E \mid X \models \ell\} \subset \{\ell \in E \mid Y \models \ell\}$
  - $dom(subset) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$.

All previous preference types are primitive since they do not refer to auxiliary preferences. Such preferences are free of naming predicate $name$. Given that our optimization

[4] Thus, $dom(t)$ also delineates the sub-language of preference elements admissible for preference statements of type t.

approach is centered upon strict preference relations, we only provide explicit definitions of strict preference types, and omit explicit definitions of any variations.

Whereas $\mathit{less}(\mathit{cardinality})$ corresponds to the common *#minimize* directive applied to (regular) literals, and $\mathit{more}(\mathit{weight})$ matches *#maximize* applied to weighted literals, subset goes already beyond existing preferences in ASP solvers. Obviously many more types of preferences can be defined.

Composite preferences are formed by aggregation. The naming predicate name is used to refer to auxiliary preferences. For a naming atom $\mathit{name}(s')$, we let $\succeq_{s'}$, $\succ_{s'}$, $=_{s'}$, $\prec_{s'}$, $\preceq_{s'}$ be the non-strict, strict, equal, and inverse preference relations associated with preference statement s'.

For example, letting N be the set of naming atoms, in *asprin* the preference type

- neg is defined by
  - $\mathit{def}_{\mathit{neg}}(E, X, Y) = (X \prec_s Y)$ where $E = \{\mathit{name}(s)\}$
  - $\mathit{dom}(\mathit{neg}) = \{\{n\} \mid n \in N\}$;
- and is defined by
  - $\mathit{def}_{\mathit{and}}(E, X, Y) = \bigwedge_{\mathit{name}(s)\in E}(X \succ_s Y)$
  - $\mathit{dom}(\mathit{and}) = \mathcal{P}(\{n \mid n \in N\})$;
- pareto is defined by
  - $\mathit{def}_{\mathit{pareto}}(E, X, Y) = \bigwedge_{\mathit{name}(s)\in E}(X \succeq_s Y) \wedge \bigvee_{\mathit{name}(s)\in E}(X \succ_s Y)$
  - $\mathit{dom}(\mathit{pareto}) = \mathcal{P}(\{n \mid n \in N\})$;
- lexico is defined by
  - $\mathit{def}_{\mathit{lexico}}(E, X, Y) = \bigvee_{w:\mathit{name}(s)\in E}((X \succ_s Y) \wedge \bigwedge_{v:\mathit{name}(s')\in E, v<w}(X =_{s'} Y))$
  - $\mathit{dom}(\mathit{lexico}) = \mathcal{P}(\{n \mid n \in N\})$.

Many other composite preference types can be defined. For instance, we could define a more symbolic lexicographic preference type by using a single preference element $\mathit{name}(s_1) > \cdots > \mathit{name}(s_m)$ rather than m preference elements being weighted literals $1 : \mathit{name}(s_1), \ldots, m : \mathit{name}(s_m)$.

Although the above preference types only accept sets of naming atoms, there is no restriction on mixing them with weighted literals. However, care must be taken to guarantee that the preference gives a strict partial order. For instance, we ignore plain disjunctive preferences like $\bigvee_{s\in E}(X \succ_s Y)$ since they may yield non-strict partial orders.

A preference relation is obtained by applying a preference type to an admissible set of preference elements; thus, $\#\mathit{preference}(s, t)\, E$ declares the preference relation $t(E)$. For simplicity, however, let us denote the resulting preference relation by $\succ_s$ rather than $t(E)$. Accordingly, the preference statement $\#\mathit{preference}(1, \mathit{less}(\mathit{cardinality}))\{a, \neg b, c\})$ declares

$$X \succ_1 Y \text{ as } |\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$

rather than $\mathit{less}(\mathit{cardinality})(\{a, \neg b, c\})$.

For further illustration, consider the preference relations induced by the following preference statements.

- $\#\mathit{preference}(2, \mathit{more}(\mathit{weight}))\{1 : a, 2 : \neg b, 3 : c\})$ declares
  $X \succ_2 Y$ as $\sum_{(w:\ell)\in\{1:a,2:\neg b,3:c\},X\models\ell} w > \sum_{(w:\ell)\in\{1:a,2:\neg b,3:c\},Y\models\ell} w$

– $\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\})$ declares
    $X \succ_3 Y$ as $\{\ell \in \{a, \neg b, c\} \mid X \models \ell\} \subset \{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}$
– $\#\mathit{preference}(4, \mathit{and})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\})$ declares
    $X \succ_4 Y$ as $(X \succ_1 Y) \wedge (X \succ_2 Y) \wedge (X \succ_3 Y)$
– $\#\mathit{preference}(5, \mathit{pareto})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\})$ declares
    $X \succ_5 Y$ as
    $(X \succeq_1 Y) \wedge (X \succeq_2 Y) \wedge (X \succeq_3 Y)$ and $(X \succ_1 Y) \vee (X \succ_2 Y) \vee (X \succ_3 Y)$
– $\#\mathit{preference}(6, \mathit{lexico})\{1\!:\mathit{name}(1), 2\!:\mathit{name}(2), 3\!:\mathit{name}(3)\})$ declares
    $X \succ_6 Y$ as
    $(X \succ_1 Y) \vee (X =_1 Y) \wedge (X \succ_2 Y) \vee (X =_1 Y) \wedge (X =_2 Y) \wedge (X \succ_3 Y)$

We note that preference relations of a primitive type are uniquely determined by their type and the result of evaluating the underlying preference elements, whereas composite ones additionally depend on the auxiliary preference relations. These must be provided with an encompassing preference specification.

3 Handling Preferences

We consider logic programs over a set $\mathcal{A}$ of atoms and refer to them as *base programs*. For implementing preferences among the stable models of base programs, we introduce the notion of a *preference program* formed over a set $\mathcal{E} \cup \mathcal{F}$ of atoms disjoint from $\mathcal{A}$. While $\mathcal{F}$ provides a fixed set of internal atoms (formed from dedicated predicate symbols), $\mathcal{E}$ can be customized as long as it complies with restrictions (see below).[5]
Instance format. 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

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

where each $\texttt{w}_\texttt{i}$ represents w_i for $i = 1, \dots, l$ and $\texttt{t}_\phi$ is a term representing ϕ by using function symbols `neg/1`, `and/2`, and `or/2`. For simplicity, we use indexes, r and j, for identifying the respective structural components. For representing Condition C of e_j, we set `r` to `0`. A naming atom $\mathit{name}(s)$ is represented analogously, except that $\texttt{for}(\texttt{t}_\phi)$ is replaced by `name(s)`.

We let $F_{s,j}$ denote the set of all facts obtained for all weighted formulas and naming atoms contained in a preference element e_j belonging to some preference statement s. With this, we define the translation of a preference statement $\#\mathit{preference}(s, t)\{e_1, \dots, e_n\}$ as

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

All atoms over predicates $\texttt{preference}/n$ for $n = 2, 5$ are internal and so belong to $\mathcal{F}$.

For example, the previous preference statements are translated as follows. [6]

[5] *asprin* makes all atoms in $\mathcal{E} \cup \mathcal{F}$ internal by adding a leading underscore "_".
Such atoms can be suppressed in the output via *clingo*'s option `--outf`.
[6] The set of facts F_3 and F_4 are described below.

– $\#\mathit{preference}(1, \mathit{less}(\mathit{cardinality}))\{a, \neg b, c\})$ yields F_1 containing

```
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 F_2 containing

```
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)).
```

– $\#\mathit{preference}(5, \mathit{pareto})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\})$ yields F_5 containing

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

– $\#\mathit{preference}(6, \mathit{lexico})\{1{:}\mathit{name}(1), 2{:}\mathit{name}(2), 3{:}\mathit{name}(3)\}$ yields F_6 containing

```
preference(6,lexico).                   preference(6,1,1,name(1),(1)).
                                        preference(6,2,1,name(2),(2)).
                                        preference(6,3,1,name(3),(3)).
```

A full preference specification S is represented as the set of facts $F_S = \bigcup_{s\in S} F_s$.
Encoding. We first consider a base program P over $\mathcal{A}$ along with a single primitive preference statement as in (6) subject to an optimization directive. Later this will be extended to composite preference statements. In both cases, the semantics of preference statements is captured by ASP encodings.
Encoding primitive preference statements. For deciding whether one stable model is preferable to another one, we implement each preference type t by an ASP encoding E_t over a set $\mathcal{E} \cup \mathcal{F}$ of atoms disjoint from $\mathcal{A}$ such that[7] $\mathit{head}(E_t) \subseteq \mathcal{E} \setminus \mathcal{F}$. This encoding is accompanied by the facts F_s representing preference statement s along with auxiliary rules, A, furnishing basic internal concepts.[8] Both F_s and A are programs over $\mathcal{F}$.

For comparing stable models by means of such an ASP encoding, we rely on ASP's meta interpretation capacities and reify atoms to constants by using unary predicates holds and holds'. To this end, we define for $X \subseteq \mathcal{A}$ the following sets.

$$\begin{array}{lll} H_X = \{\mathit{holds}(a) \mid a \in X\} & \text{and} & H'_X = \{\mathit{holds}'(a) \mid a \in X\} \\ R_X = \{\mathit{holds}(a) \leftarrow a \mid a \in X\} & \text{and} & R'_X = \{\mathit{holds}'(a) \leftarrow a \mid a \in X\} \\ G_X = \{\{h\} \leftarrow \mid h \in H_X\} & \text{and} & G'_X = \{\{h'\} \leftarrow \mid h' \in H'_X\} \end{array}$$

Note that R_X provides the dynamic counterpart of H_X within an encompassing model X. Atoms formed by using predicates holds and holds' are internal and thus belong to $\mathcal{F}$, that is, $H_{\mathcal{A}} \cup H'_{\mathcal{A}} \subseteq \mathcal{F}$.

The next definition captures the central building block of our approach.

Definition 1. *Let s be a preference statement declaring preference relation $\succ_s$ and let the programs E_{t_s}, F_s, and A be defined as above. We call $E_{t_s} \cup F_s \cup A$ a* preference program *for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have*

$$X \succ_s Y \quad \textit{iff} \quad E_{t_s} \cup F_s \cup A \cup H_X \cup H'_Y \textit{ is satisfiable.}$$

[7] $\mathit{head}(P)$ gives the set of all atoms occurring in the heads of rules in P.
[8] As shown below, this includes enabling optimization wrt primary preferences and satisfaction of Boolean expressions.

Note that preference programs refer only to atom sets and are thus at first independent of any base programs. This changes once a program P together with reifying or generating rules, like R_X or G_X, is considered (instead of H_X).

Base and preference programs are formed over disjoint sets of atoms. Interactions among them are controlled by mapping atoms in $\mathcal{A}$ to $H_\mathcal{A}$ and $H'_\mathcal{A}$, respectively. The next proposition makes precise how preference programs capture the semantics of preference statements.

Proposition 1. *Let $E_{t_s} \cup F_s \cup A$ be a preference program for preference statement s.*

1. *If Z is a stable model of $E_{t_s} \cup F_s \cup A \cup G_\mathcal{A} \cup G'_\mathcal{A}$,*
   then $\{a \mid \mathit{holds}(a) \in Z\} \succ_s \{a \mid \mathit{holds}'(a) \in Z\}$.
2. *If $X \succ_s Y$, then there is a stable model Z of $E_{t_s} \cup F_s \cup A \cup G_\mathcal{A} \cup G'_\mathcal{A}$ such that*
   $X = \{a \mid \mathit{holds}(a) \in Z\}$ and $Y = \{a \mid \mathit{holds}'(a) \in Z\}$.

The above implies that $\succ_s = \{(X,Y) \mid X,Y \subseteq \mathcal{A}, (E_{t_s} \cup F_s \cup A \cup H_X \cup H'_Y)$ is satisfiable$\}$.

Next, we show how preference programs can be used for deciding whether a stable model of a base program is preferred, and how a dominating model is obtainable.

Proposition 2. *Let P be a program over $\mathcal{A}$ and let s be a preference statement.*

1. *If X is a stable model of P, then X is $\succ_s$-preferred iff $\left(P \cup E_{t_s} \cup F_s \cup A \cup R_\mathcal{A} \cup H'_X\right)$ is unsatisfiable.*
2. *If Y is a stable model of $\left(P \cup E_{t_s} \cup F_s \cup A \cup R_\mathcal{A} \cup H'_X\right)$ 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 $(P \cup E_{t_s} \cup F_s \cup A \cup R_\mathcal{A} \cup H'_X)$ for checking 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 1).

Encoding composite preference statements. Both primitive and composite preference statements are implemented by preference programs. Their difference boils down to whether they rely upon secondary preferences or not.

For convenience, *asprin* allows us to expand the range of preference statements from defining strict partial orders to the corresponding strict, non-strict, equal, and inverse counterparts. The latter are not involved when optimizing wrt the preference statement at hand but can be used for defining other preference relations. In this setting, the concepts of preference types and relations are extended to families of preference types and relations, and preference programs become preference modules. As before, specific relations are obtained by instantiating the corresponding types with preference elements. However, while a preference program for a preference statement s is used for optimizing s, the preference module for s is added to a preference program of another preference statement importing s's family of relations via naming atom $\mathit{name}(s)$. In other words, a preference statement s induces both a preference program as well as a preference module; the former is used when s is the primary statement in a preference specification, the latter whenever s is a secondary one.

Implementation in *asprin*. Let us consider how our selected preference statements are implemented in *asprin*. To begin with, we detail *asprin*'s auxiliary rules A.

First, the directive $\#\mathit{optimize}(s)$ is turned into a fact

```
1   optimize(s).
```

This fact enables optimization wrt the primary preference s together with the integrity constraint, O_0:

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

This is complemented by the following program H, used for implementing the satisfaction of formulas.

```
3   formula(F) :- preference(_,_,_,for(F),_).
4   formula(F) :- formula(neg(F)).
5   formula(F) :- formula(and(F,G)).  formula(G) :- formula(and(F,G)).
6   formula(F) :- formula(or(F,G)).   formula(G) :- formula(or(F,G)).

8   holds(neg(F))     :- formula(neg(F)),  not holds(F).
9   holds(and(F,G))   :- formula(and(F,G)),    holds(F), holds(G).
10  holds(or(F,G))    :- formula(or(F,G)), 1 { holds(F); holds(G)}.
```

Looking at Line 3, we observe that the instantiation of the rules in H is driven by the occurrence of formulas in F_s. The satisfaction of a formula is checked wrt to the interpretation expressed by the set of atoms a for which `holds`(a) holds. The set H' is obtained from H by replacing all occurrences of `holds/1` by `holds'/1`. All atoms formed by using predicates `formula/1` (as well as `holds/1` and `holds'/1` applied to composite terms) are internal and thus belong to $\mathcal{F}$.

Next, we show how our selected preference types (and relations; marked with ▷) are implemented in *asprin*. Each preference type is captured by a (non-ground) encoding.

– $\mathit{less}(\mathit{cardinality})$ is implemented by the rule, $O^{\succ}_{\mathit{less}(\mathit{cardinality})}$,

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

▷ $\#\mathit{preference}(1, \mathit{less}(\mathit{cardinality}))\{a, \neg b, c\}$
is implemented by program[9] $\{O^{\succ}_{\mathit{less}(\mathit{cardinality})}\} \cup F_1 \cup \{O_0\} \cup H \cup H'$

– $\mathit{more}(\mathit{weight})$ is implemented by the rule, $O^{\succ}_{\mathit{more}(\mathit{weight})}$,

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

▷ $\#\mathit{preference}(2, \mathit{more}(\mathit{weight}))\{1 : a, 2 : \neg b, 3 : c\})$
is implemented by program $\{O^{\succ}_{\mathit{more}(\mathit{weight})}\} \cup F_2 \cup \{O_0\} \cup H \cup H'$

– subset is implemented by the rule, $O^{\succ}_{\mathit{subset}}$,

```
better(P) :- preference(P,subset),
          not holds(X) : preference(P,_,_,for(X),_), not holds'(X);
          1 #sum { 1,X : not holds(X), holds'(X), preference(P,_,_,for(X),_)}.
```

▷ $\#\mathit{preference}(3, \mathit{subset})\{a, \neg b, c\}$
is implemented by program $\{O^{\succ}_{\mathit{subset}}\} \cup F_3 \cup \{O_0\} \cup H \cup H'$ and F_3 is obtained from F_1 by replacing `less(cardinality)` by `subset` as well as the first argument of all facts by `3`.

– and is implemented by the rule, $O^{\succ}_{\mathit{and}}$,

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

[9] This is not yet a preference program since it lacks the fact `optimize(1)`.

▷ $\#\mathit{preference}(4, \mathit{and})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\}$
is implemented by program $\{O^{\succ}_{\mathit{and}}\} \cup F_4 \cup \{O_0\} \cup H \cup H'$ and F_4 is obtained from F_5 by replacing `pareto` by `and` as well as the first argument of all facts by `4`.

Note that for the primitive preference statements $s = 1, 2, 3$, each program $\{O^{\succ}_{t_s}\} \cup F_s \cup \{O_0\} \cup H \cup H'$ only becomes a preference program once the fact `optimize(s)` is added. This activates the preference implementation via integrity constraint O_0. Since preference statement 4 relies on the composite type *and*, its implementation requires in addition to `optimize(4)` all programs $\{O^{\succ}_{t_s}\} \cup F_s$ for $s = 1, 2, 3$ to constitute a preference program. This preference program nicely illustrates why the above programs are at first free of any `optimize` facts. In this way, preference implementations can be modularly activated and used for implementing composite preference statements. [10]

Basic preference types as well as *and* only involve strict preference relations. Unlike this, *pareto* and *lexico* also make use of non-strict and equal auxiliary relations. Such auxiliary relations can either be found in the preference modules of *asprin*'s library or are supplied by the user. In what follows, we omit explicit definitions of the various counterparts of `better/1`, and refer to the definitions of `bettereq/1`, `better/1`, `equal/1`, `worse/1`, and `worseeq/1` by $O^{\succeq}_{t_s}$, $O^{\succ}_{t_s}$, $O^{=}_{t_s}$, $O^{\prec}_{t_s}$, $O^{\preceq}_{t_s}$, respectively.

– *pareto* is implemented by the rule, $O^{\succ}_{\mathit{pareto}}$,

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

▷ $\#\mathit{preference}(5, \mathit{pareto})\{\mathit{name}(1), \mathit{name}(2), \mathit{name}(3)\}$
is implemented by program $\{O^{\succ}_{\mathit{pareto}}\} \cup F_5 \cup \{O_0\} \cup H \cup H'$

– *lexico* is implemented by the rule, $O^{\succ}_{\mathit{lexico}}$,

```
better(P) :- preference(P,lexico);
             better(P'), preference(P,_,_,name(P'),(N)),
             equal(P''): preference(P,_,_,name(P''),(M)), M < N.
```

▷ $\#\mathit{preference}(6, \mathit{lexico})\{1 : \mathit{name}(1), 2 : \mathit{name}(2), 3 : \mathit{name}(3)\}$
is implemented by program $\{O^{\succ}_{\mathit{lexico}}\} \cup F_6 \cup \{O_0\} \cup H \cup H'$

Note that 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.

4 Computing Preferences

Our algorithms rely upon consecutive calls to an incremental ASP solver (viz. *clingo* 4). For a (normal or disjunctive) program P, define

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

Computing one preferred model. Given a program P and a preference statement s, Algorithm 1 computes a $\succ_s$-preferred stable model of P (and thus implicitly addresses

Algorithm 1: $solveOpt(P, s)$

```
Input  : A program $P$ over $\mathcal{A}$ and preference statement $s$.
Output : A $\succ_s$-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     $Y \leftarrow solve(P \cup E_{t_s} \cup F_s \cup R_{\mathcal{A}} \cup H'_X) \cap \mathcal{A}$
6 until $Y = \bot$
7 return $X$
```

the decision problem). Note that we put no restrictions on the base program P or on the preference program; we are even free to use disjunctive programs at both ends.

Algorithm 1 is a branch-and-bound algorithm implementing the non-dominance test for candidate models as prescribed by Proposition 1. 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 $\succ_s$-preferred. If this fails, we obtain with Y a counterexample dominating X. Then we proceed with Y as new candidate model.

Theorem 1. *Given a program P and a preference statement s, Algorithm 1 computes some $\succ_s$-preferred stable model of P, if P is satisfiable, and $\bot$ otherwise.*

Computing all preferred models. Next, we address the problem of enumerating preferred models. While we still impose no restriction on base programs, we first limit ourselves to preferences for which we can decide whether $X \succ Y$ holds for sets X, Y in polynomial time. Given this, we assume without loss of generality that preference programs are stratified [9] because each problem decidable in polynomial time can be represented as a stratified logic program.

Given a program P and a preference statement s, Algorithm 2 computes all $\succ_s$-preferred stable models of P. The idea 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. We accumulate preferred models in an 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 of Algorithm 2, 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 clearly 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 $i \in I$. Note that we also have $Y \not\succ_s X_i$ because each $X_i \in \mathcal{X}$ is $\succ_s$-preferred.

The condition "$X_i \neq Y$" is guaranteed by the addition of an integrity constraint N_{X_i} of form $N_X = \{\leftarrow X \cup \{\sim a \mid a \in \mathcal{A} \setminus X\}\}$ for each $i \in I$. Although such solution recording is exponential in space, it is non-intrusive to the solver.

[10] Note that the addition of any other fact `optimize(s)` for $s = 1, 2, 3$ rather than $s = 4$ also yields a preference program yet only for the indicated preference statement s.

Algorithm 2: $solveOptAll(P, s)$

```
Input  : A program $P$ over $\mathcal{A}$ and preference statement $s$.
Output : The set of all $\succ_s$-preferred stable models of $P$.
1 $\mathcal{X} \leftarrow \emptyset$
2 loop
3     $Y \leftarrow solve\big(P \cup \bigcup_{X_i \in \mathcal{X}} \big(N_{X_i} \cup (\overline{E_{t_s}} \cup F_s \cup H_{X_i})^i \cup R'^{i}_{\mathcal{A}}\big)\big) \cap \mathcal{A}$
4     if $Y = \bot$ then return $\mathcal{X}$
5     repeat
6         $X \leftarrow Y$
7         $Y \leftarrow solve\big(P \cup E_{t_s} \cup F_s \cup R_{\mathcal{A}} \cup H'_X\big) \cap \mathcal{A}$
8     until $Y = \bot$
9     $\mathcal{X} \leftarrow \mathcal{X} \cup \{X_{|\mathcal{X}|+1}\}$
```

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. Unlike this, 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 can be accomplished as follows: Given a program P, define $\overline{P}$ as the program

$$(P \setminus \{r \in P \mid head(r) = \emptyset\}) \cup \{u \leftarrow body(r) \mid r \in P, head(r) = \emptyset\} \cup \{\leftarrow {\sim} u\} \ ,$$

where u is a new atom. Now if program P is stratified, P is satisfiable iff $\overline{P}$ is unsatisfiable. Moreover, let $E_{t_s} \cup F_s$ be a stratified preference program for preference statement s. Then, for all sets X, Y of atoms over $\mathcal{A}$, we have

$$X \not\succ_s Y \ \text{ iff } \ \overline{E_{t_s}} \cup F_s \cup H_X \cup H'_Y \text{ is satisfiable.}$$

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

Proposition 3. *Let P be a program over $\mathcal{A}$ and let s be a preference statement with a stratified preference program $E_{t_s} \cup F_s$. If Y is a stable model of $\big(P \cup \overline{E_{t_s}} \cup F_s \cup H_X \cup R'_{\mathcal{A}}\big)$ 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 $\succ_s$-preferred.

Given that $\mathcal{X}$ contains several preferred models, we need to accomplish the check in Proposition 3 for each model in $\mathcal{X}$. For this, we denote by P^i the program obtained from P by replacing each atom a in P by a^i. Moreover, we generalise the definition of R'_X to $R'^{i}_X = \{holds'(a)^i \leftarrow a \mid a \in X\}$. With this, the next proposition captures the functioning of Line 3 of Algorithm 2.

Proposition 4. *Let $\{X_i \mid i \in I\}$ be the value of $\mathcal{X}$ in Line 2 of Algorithm 2 and let Y be the set of atoms subsequently returned in Line 3 of Algorithm 2. Then, Y is a stable model of P such that $Y \neq X_i$, $Y \not\succ_s X_i$, and $X_i \not\succ_s Y$ for all $i \in I$.*

All in all, we obtain the following soundness and completeness result.

Theorem 2. *Given a program P and a preference statement s, Algorithm 2 computes the set of all $\succ_s$-preferred stable models of P.*

Computing all preferred models for complex preferences. We now remove the restriction of polynomially decidable preference relations, and consider preferences decidable in *NP*. 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 [10] the *saturation* technique. The idea is to re-express the problem as a positive disjunctive logic program, containing a special-purpose atom bot. Whenever bot is obtained, saturation derives all atoms (belonging to a "guessed" model). Intuitively, this is a way to materialize unsatisfiability. For automating this process, we build upon the meta-interpretation-based approach in [11]. The idea is to map a program R onto a set $\mathcal{R}(R)$ of facts via reification. The set $\mathcal{R}(R)$ of facts is then combined with a meta-encoding $\mathcal{M}$ from [11] implementing saturation.

We consider for a preference statement s the positive disjunctive logic program

$$\mathcal{R}\big(E_{t_s} \cup F_s \cup G_{\mathcal{A}} \cup G'_{\mathcal{A}}\big) \cup \mathcal{M} \ .$$

In analogy to Proposition 1, this program has a stable model (excluding bot) for each pair $X, Y \subseteq \mathcal{A}$ such that $X \succ_s Y$, and it has a saturated stable model (including bot) if there is no such pair. Note that X and Y are subsets of $\mathcal{A}$, not necessarily stable models.

Given this, it is sufficient to replace the program passed to $solve$ in Line 3 of Algorithm 2 by the following disjunctive logic program:

$$\big(P \cup \textstyle\bigcup_{X \in \mathcal{X}} N_X\big) \cup \big(\mathcal{R}(E_{t_s} \cup F_s \cup G_{\mathcal{A}} \cup G'_{\mathcal{A}}) \cup \mathcal{M}\big) \cup \mathcal{N}_{\mathcal{X}} \cup \mathcal{R}'_{\mathcal{A}} \cup \{\leftarrow {\sim} bot\}$$

As above, $\big(P \cup \bigcup_{X \in \mathcal{X}} N_X\big)$ generates stable model candidates different from those in $\mathcal{X}$. The programs $\mathcal{N}_{\mathcal{X}}$ and $\mathcal{R}'_{\mathcal{A}}$ restrict the choices of $G_{\mathcal{A}}$ and $G'_{\mathcal{A}}$, respectively. Note that [11] represent an atom a as $true(atom(a))$ and ${\sim}a$ as $fail(atom(a))$.

$$\begin{aligned}
\mathcal{N}_{\mathcal{X}} &= \{bot \leftarrow \textstyle\bigcup_{X_i \in \mathcal{X}}\{u_i\}\} \cup \textstyle\bigcup_{X_i \in \mathcal{X}}\{u_i \leftarrow fail(atom(holds(a))) \mid a \in X_i\} \\
&\qquad\qquad\qquad\qquad \cup \textstyle\bigcup_{X_i \in \mathcal{X}}\{u_i \leftarrow true(atom(holds(a))) \mid a \in \mathcal{A} \setminus X_i\} \\
\mathcal{R}'_{\mathcal{A}} &= \{true(atom(holds'(a))) \leftarrow a \mid a \in \mathcal{A}\} \cup \{fail(atom(holds'(a))) \leftarrow {\sim}a \mid a \in \mathcal{A}\}
\end{aligned}$$

While $\mathcal{N}_{\mathcal{X}}$ eliminates all non-preferred models (outside of $\mathcal{X}$) from the candidate sets generated by $G_{\mathcal{A}}$ via saturation, program $\mathcal{R}'_{\mathcal{A}}$ maps all candidate models generated by $(P \cup \bigcup_{X \in \mathcal{X}} N_X)$ to $\mathcal{R}(H'_{\mathcal{A}})$.

Computing preferred models by extension. The axiomatic way of computing preferred models is to extend a program so that the stable models of the extended program correspond to the preferred stable models of the original program. In ASP, this extension can be formulated via saturation (cf. [11]) because one has to combine the generation of model candidates with the failure to generate dominating "counter"-models.

This is easily accomplished by means of the above building blocks. We consider for a base program P and a preference statement s the positive disjunctive program

$$\mathcal{R}\big(E_{t_s} \cup F_s \cup (P \cup R_{\mathcal{A}}) \cup G'_{\mathcal{A}}\big) \cup \mathcal{M} \ .$$

However, instead of generating arbitrary sets of atoms via $G_{\mathcal{A}}$ as above, we now only generate stable models of P. Hence, the above program has a stable model (excluding bot) for each pair $X, Y \subseteq \mathcal{A}$ such that $X \succ_s Y$ and X is a stable model of P; 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.

$$P \cup \big(\mathcal{R}\big(E_{t_s} \cup F_s \cup P \cup R_{\mathcal{A}} \cup G'_{\mathcal{A}}\big) \cup \mathcal{M}\big) \cup \mathcal{R}'_{\mathcal{A}} \cup \{\leftarrow {\sim} bot\}$$

For computing one or all $\succ_s$-preferred models of P, it is sufficient to pass $\mathcal{E}(P, s)$ to a disjunctive ASP solver along with the appropriate option. Similarly, checking whether a query is true in some $\succ_s$-preferred model of P can be done by passing the program $\mathcal{E}(P, s) \cup \{\leftarrow {\sim} a\}$ to the solver. To do the same with Algorithm 2, one had to enumerate preferred models until one comprising a is found.

The major difference between the algorithmic and axiomatic approach is that the former relies on a sequence of problems successively passed to a solver, while the latter encapsulates all into solving a single problem. Hence, in ASP, the axiomatic approach is restricted to problems at the second level of the polynomial hierarchy, while branch-and-bound can go beyond this because only each solver invocation is restricted to such problems. Also, the complete problem captured by $\mathcal{E}(P, s)$ is often more complex than the successive problems considered in Algorithm 1 and 2. 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.

5 Discussion

We introduced in this paper a general, flexible and extendable 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 studied before (although *asprin* certainly allows one to introduce such new relations). Rather our goal was to provide ASP technology matching the substantial amount of research on preference handling in ASP and beyond. In a nutshell, we want to put this research to 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.

We expect two types of *asprin* users. Those who are happy with the preference relations available in the *asprin* library, and those who want to benefit from the extendability of the system and define their own domain-specific preference orderings. For the former, much of the technical detail we described in this paper, e.g. the internal representation of preference statements and the way preference programs work, are not essential. In fact, they can use *asprin* as a ready to use preference handling framework where all one needs to know are the available preference types and their arguments. For the latter type of users, let's call them preference engineers, the system provides all the additional functionality to define interesting new preference orderings.

We already mentioned the large body of work on preferences in logic programming. In fact, the literature is too large to be discussed here in detail and we refer the reader to the article [3] for a detailed overview. However, we want to mention that our algorithms are inspired by ideas in [12, 13, 11]. Moreover, we emphasize that the approaches one typically encounters in the literature can be modelled in our system. Arguably, the approach closest to ours is the one proposed in [14]. Therein, Brewka introduces a specific preference language with a predefined set of preference relations and combination methods. He also uses preference programs to identify optimal stable models. However, his preference language is fixed and does not have the flexibility of our approach. The preference programs are somewhat "naive" and certainly not up to modern ASP solving technology. Moreover, the only reasoning problem addressed is computing a single preferred model.

We are currently performing an empirical evaluation of our system. We report our results at [1] once a more substantial amount of testing has been done.

The *asprin* system is freely available at [1].

References

1. *asprin*: `http://potassco.sourceforge.net/labs.html#asprin`
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence **20**(2) (2004) 308–334
4. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
5. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
7. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.0.pdf` (2012)
8. Bienvenu, M., Lang, J., Wilson, N.: From preference logics to preference languages, and back. In Lin, F., Sattler, U., eds.: Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10), AAAI Press (2010)
9. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1987) 89–148
10. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence **15**(3-4) (1995) 289–323
11. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming **11**(4-5) (2011) 821–839
12. Brewka, G., Niemelä, I., Truszczyński, M.: Answer set optimization. In Gottlob, G., Walsh, T., eds.: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Morgan Kaufmann Publishers (2003) 867–872
13. Giunchiglia, E., Maratea, M.: Algorithms for solving satisfiability problems with qualitative preferences. In Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz. Springer-Verlag (2012) 327–344
14. Brewka, G.: Complex preferences for answer set optimization. In Dubois, D., Welty, C., Williams, M., eds.: Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04), AAAI Press (2004) 213–223