Implementing preferences with *asprin*[⋆]

Gerhard Brewka[3], James Delgrande[2], Javier Romero[4], and Torsten Schaub[1,4⋆⋆]

[1] INRIA Rennes [2] Simon Fraser University [3] Universität Leipzig [4] Universität Potsdam

Abstract. *asprin* offers a framework for expressing and evaluating combinations of quantitative and qualitative preferences among the stable models of a logic program. In this paper, we demonstrate the generality and flexibility of the methodology by showing how easily existing preference relations can be implemented in *asprin*. Moreover, we show how the computation of optimal stable models can be improved by using declarative heuristics. We empirically evaluate our contributions and contrast them with dedicated implementations. Finally, we detail key aspects of *asprin*'s implementation.

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]). On the other hand, such quantitative ways of optimization are often insufficient for applications and in stark contrast to the vast literature on qualitative and hybrid means of optimization [4–8].

This gulf is bridged by the *asprin* system, which offers a flexible and general framework for implementing complex combinations of quantitative and qualitative preferences. The primary contribution of this paper is to substantiate this claim by showing how easily selected approaches from the literature can be realized with *asprin*. In particular, we detail how answer set optimization [5], minimization directives [2], strict partially ordered sets [8], and the non-temporal part of the preference language in [7] can be implemented with *asprin*. Moreover, we sketch how the implementations of ordered disjunctions [4] and penalty-based answer set optimization [6] are obtained. In fact, *asprin*'s simple interface and straightforward methodology reduces the implementation of customized preferences to defining whether one model is preferred to another. This also lays bare *asprin*'s expressive power, which is delineated by the ability to express a preference's decision problem within ASP. In view of the practical relevance of preferences, we also investigate the use of ASP's declarative heuristics for boosting the search for optimal stable models. In particular, we are interested how the combination of ASP's two general-purpose frameworks for preferences and heuristics compares empirically with dedicated implementations.

The paper [9] introduced *asprin*'s approach and focused on fundamental aspects. Apart from a formal elaboration of *asprin*'s propositional language, it provided semantics and encodings for basic preferences from *asprin*'s library (like `subset` or

⋆ This work was funded by DFG (BR 1817/5; SCHA 550/9) and NSERC.

⋆⋆ Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

`less(weight)`). As well, it empirically contrasted the implementation of such basic preferences with the dedicated one in *clasp* and analyzed *asprin*'s scalability in view of increasingly nested preference structures. Here, we build upon this work and focus on engineering aspects. First, we introduce the actual first-order preference modeling language of *asprin*, including its safety properties, and carve out its simple interfaces and easy methodology. Second, we demonstrate how existing preferences from the literature can be implemented via *asprin*. In doing so, we provide best practice examples for preference engineers. Third, we show how declarative heuristics can be used for boosting the computation of optimal models. And last but not least, we detail aspects of *asprin*'s implementation and contrast it with dedicated ones.

In what follows, we rely upon a basic acquaintance with ASP [1, 10].

2 *asprin*'s approach at a glance

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 `s`, a type `t`, and an argument set: '`#preference(s,t){`e_1`;...;`e_n`} : `B`.`' The identifier names the preference relation, whereas its type and arguments define the relation; the set B of built-in or domain predicates is used for instantiation.[1] Identifiers and types are represented by terms, while each argument e_j is a *preference element*. For safety, variables appearing in `s` or `t` must occur in a positive atom of B. Let us consider an example before delving into further details:

```
#preference(costs,less(weight)){ C :: activity(A) : cost(A,C) }.
```

This statement declares a preference relation `costs` with type `less(weight)`. Given atoms `cost(sauna,40)` and `cost(dive,70)`, grounding results in one of the simplest form of preference elements, namely the weighted literals `40::activity(sauna)` and `70::activity(dive)`. Informally, the resulting preference relation prefers stable 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. We refer the reader to [9] on how preference statements induce preference relations by applying preference types to preference elements. And we focus in what follows on *asprin*'s syntactic features.

Preference elements 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. Moreover, preference types may refer to other preferences in their arguments, which is used for preference composition. For instance, the statement

```
#preference(all,pareto){ name(costs), name(temps) }.
```

defines a preference relation `all`, which is the Pareto ordering of preference relations `costs` and `temps`.

More formally, a *preference element* is of the form '$\boldsymbol{F}_1 > \ldots > \boldsymbol{F}_m$ `||` F `:` B' where each $\boldsymbol{F}_r$ is a set of weighted formulas, F is a non-weighted Boolean formula, and B is as above. We drop '>' if $m = 1$, and '`||` F' and '`:` B' whenever F and/or

[1] Just as with bodies, we drop curly braces from such sets.

B are empty, respectively. Intuitively, r gives the rank of the respective set of weighted formulas. This can be made subject to condition F by using the conditional '`||`'. 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.

A set of weighted formulas $\boldsymbol{F}_r$ is represented as '$\{F_1;\ldots;F_m\}$'. We drop the curly braces if $m = 1$. And finally, a *weighted formula* is of the form '$\boldsymbol{t}$`::`F' where $\boldsymbol{t}$ is a term tuple and F is a either a Boolean formula or a naming atom. We may drop `::` and simply write F whenever $\boldsymbol{t}$ is empty. Boolean formulas are formed from atoms, possibly preceded by strong negation ('$-$'), using the connectives `not` (negation), `&` (conjunction) and `|` (disjunction). Parentheses can be written as usual, and when omitted, negation has precedence over conjunction, and conjunction over disjunction. Naming atoms of form `name(s)` refer to the preference associated with preference statement `s` (cf [9]). For safety, variables appearing in a weighted formula must occur in a positive atom of the set B from either the corresponding preference element or preference statement. Examples of preference elements include '`a(X)`', '`42::b(X)`', '`{1::name(p);2::name(q)}`', '`{a(X);b(X)} > {c(X);d(X)}`', and '`a(X) > b(X) || c(X) : dom(X)`'.

Since preference statements may only be auxiliary, a preference relation must be distinguished for optimization. This is done via an optimization statement of form '`#optimize(s).`' with the name of the respective preference statement as argument.

Finally, a *preference specification* is a set of preference statements along with an optimization directive. It is valid if grounding results in acyclic and closed naming dependencies along with a single optimization directive (see [9] for details).

Once a preference specification is given, the computation of preferred stable models is done via a branch-and-bound process relying on *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. An optimal one is computed iteratively by repeated calls to an ASP solver. First, an arbitrary stable model of the underlying program is generated; then, this stable model is "fed" to the preference program to produce a better one, etc. Once the preference program becomes unsatisfiable, the last stable model obtained is an optimal one. The basic algorithm is described in [9]; it is implemented via *clingo 4*'s Python library, providing operative grounder and solver objects.

asprin also 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 relations. In this case, they also have to provide the preference programs *asprin* needs to cope with the new preference relations.

3 Embedding existing approaches

The implementation of customized preference types in *asprin* boils down to furnishing a preference program for the preference that is subject to optimization. For the sake of generality, this is usually done for the preference type, which then gets instantiated to the specific preference relation of interest.

The purpose of a preference program is to decide whether one stable model is strictly preferred to another wrt the corresponding preference relation. To this end, we reify stable models and represent them via the unary predicates `holds/1` and `holds'/1`. More formally, we define for a set X of atoms, the following sets of facts:

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

Then, given a preference statement identified by `s`, the program P_s is a preference program implementing preference relation $\succ_s$, if for sets X, Y of atoms, we have

$$X \succ_s Y \text{ iff } P_s \cup H(X) \cup H'(Y) \text{ is satisfiable.} \quad (1)$$

See [9] for a formal elaboration of preference programs.

In what follows, we explain *asprin*'s interfaces and methodology for implementing preference programs.

To begin with, *asprin* represents preference specifications in a dedicated fact format. Each optimization directive '`#optimize(s).`' is represented as a fact

`optimize(s).`

Next, each preference statement '`#preference(s,t){`e_1`;...;`e_n`}:`B`.`' gives rise to n rules encoding preference elements along with one rule of form

`preference(s,t) :-` B`.`

In turn, preference elements are represented by several facts, each representing a comprised weighted formula. Recall that a weighted formula F_k of form '$\boldsymbol{t}$`::`F' occurs in some set $\boldsymbol{F}_i$ of form '`{`F_1`;...;`F_m`}`' (or equals F_0) of a preference element e_j of form '$\boldsymbol{F}_1$ `>...>` $\boldsymbol{F}_n$ `||` F_0 `:` B_j' that belongs itself to a preference statement `s` as given above. Given this, the weighted formula F_k is translated into a rule of the form

`preference(s,(`j`,`$\boldsymbol{v}$`),`i`,for(`t_F`),`$\boldsymbol{t}$`) :-` B_j`,` B`.`

where j and i are the indices of e_j and $\boldsymbol{F}_i$, respectively, $\boldsymbol{v}$ is a term tuple containing all variables appearing in the rule, and t_F is a term representing the Boolean formula F by using function symbols `_not/1`, `_and/2`, and `_or/2` in prefix notation. For example, the formula `(not a(X) | b(X)) & c(X)` is translated into `_and(c(X),_or(_not(a(X)),b(X)))`. For representing condition F_0, we set i to `0`. A naming atom `name(s)` is represented analogously, except that `for(`t_F`)` is replaced by `name(s)`.

For instance, the earlier preference statement `costs` is translated as follows.

```
preference(costs,(1,(A,C)),1,for(activity(A)),(C)) :- cost(A,C).
preference(costs,less(weight)).
```

Grounding the first rule in the presence of `cost(sauna,40)` and `cost(dive,70)` yields two facts, representing the weighted literals `40::activity(sauna)` and `70::activity(dive)`.

Second, *asprin* extends the basic truth assignment to atoms captured by `holds/1` and `holds'/1` to all Boolean formulas occurring in the preference specification at hand. To this end, formulas are represented as terms as described above. Hence, for any formula F occurring in the preference specification, *asprin* warrants that `holds(`t_F`)` is true whenever F is entailed by the stable model X captured in $H(X)$, where t_F is the term representation of F. This is analogous for `holds'/1`.

Third, in *asprin*'s methodology a preference program is defined generically for the preference type, and consecutively instantiated to the specific preference in view of its preference elements. Concretely, *asprin* stipulates that preference programs define the unary predicate `better/1`, taking preference identifiers as arguments. The user's implementation is required to yield `better(s)` whenever the stable model captured by $H(X)$ is strictly better than that comprised in $H'(X)$. For illustration, consider the preference program for *asprin*'s pre-fabricated preference type `less(weight)`.

```
1 better(P) :- preference(P,less(weight)),
2       1 #sum { -W,X : holds(X),  preference(P,_,_,for(X),(W));
3                 W,X : holds'(X), preference(P,_,_,for(X),(W))}.
```

asprin complements this by the generic integrity constraint

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

ensuring that `better(P)` holds whenever `P` is subject to optimization and enforces the fundamental property of preference programs in (1).

All in all, a preference program thus consists of (i) facts representing preference and optimization statements, (ii) auxiliary rules, extending predicates `holds/1` and `holds'/1` to Boolean formulas as well as the above integrity constraint, and finally (iii) the definition of the preference type(s). While parts (i) and (ii) are provided by *asprin*, only part (iii) must be provided by the "preference engineer". Our methodology accounts for this by defining predicate `better/1`. However, this is not strictly necessary as long as all three parts constitute a preference program by fulfilling (1).

Additionally, the customization of preferences can draw upon *asprin*'s library containing various pre-defined preference types. This includes the primitive types `subset` and `superset`, `less(cardinality)` and `more(cardinality)`, `less(weight)` and `more(weight)`, along with the composite types `neg`, `and` `pareto`, and `lexico`. In fact, for these types, *asprin* not only provides definitions of `better(s)` but also its non-strict, and equal counterparts, namely, `bettereq/1`, `equal/1`, `worse/1`, and `worseeq/1`. Such definitions are very useful in defining aggregating preference types such as `pareto` (see below).

Answer set optimization. For capturing answer set optimization (ASO; [5]), we consider ASO rules of form

$$\phi_1 > \cdots > \phi_m \leftarrow B \tag{2}$$

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 (2). 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 $\sim b \in B$, 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 (2), $X \succeq_O Y$ if for all rules $r \in O$, $v_X(r) \leq v_Y(r)$, and $X \succ_O Y$ is defined as $X \succeq_O Y$ but $Y \not\succeq_O X$.

In *asprin*, we can represent an ASO rule r as in (2) as preference statement of form

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

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

`#preference(paraso,pareto){name(`s_{r_1}`),...name(`s_{r_n}`)}.`

```
#optimize(paraso).
```

Note that aggregating preferences other than `pareto` could be used just as well.

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

```
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(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(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/1` 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).
21 bettereq(P) :- preference(P,aso),
22    preference(P,_,R,for(F),_), holds(F), R > 1, not one'(P),
23    not holds'(G) : preference(P,_,R',for(G),_), 1 < R',R' < R.
```

The remaining rules implement the composite preference type `pareto`.

```
25 better(P) :- preference(P,pareto),
26              better(R),    preference(P,_,_,name(R),_),
27              bettereq(Q) : preference(P,_,_,name(Q),_).
```

Note how `pareto` makes use of both the strict and non-strict preference types, viz. `better/1` and `bettereq/1`. In fact, we only list this here for completeness since the definition could be imported from *asprin*'s library.

Altogether, the rules in Line 1-27 capture the semantics of ASO. To see this, consider a set O of ASO rules and the program P_O consisting of Line 1-27 along with the facts for the preference and optimization statements corresponding to O and the auxiliary rules in (ii) mentioned above. Then, we can show that $X \succ_O Y$ holds iff $P_O \cup H(X) \cup H'(Y)$ is satisfiable.

asprin also includes an implementation of the ASO extension with penalties introduced in [6]. Here, each formula ϕ_i in (2) is extended with a weight and further weight-oriented cardinality- and inclusion-based composite preference types are defined. The

implementation in *asprin* extends the one presented above by complex weight handling and is thus omitted for brevity. Similarly, logic programs with ordered disjunction [4] are expressible in *asprin* via the translation to ASO described in [5].

Partially ordered sets. In [8], qualitative preferences are modeled as a strict partially ordered set $(\Phi, <)$ of literals. The literals in Φ represent propositions that are preferably satisfied and the strict partial order $<$ on Φ gives their relative importance. We (slightly) generalize 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 $Y \not\models \phi'$.

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 all preference relations $\succ_{(\Phi,<)}$ 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/1` and `holds'/1`, we derive an instance of `better(P,F)` whenever $X \models \phi_{\mathtt{F}}$ but $Y \not\models \phi_{\mathtt{F}}$ (and `F` is the representation of $\phi_{\mathtt{F}}$). Similarly, we derive `notbetter(P)` whenever there is a formula $\phi_{\mathtt{F}}$ such that $Y \models \phi_{\mathtt{F}}$ and $X \not\models \phi_{\mathtt{F}}$ but `better(P,G)` fails to hold for all $\phi_{\mathtt{G}}$ preferred to $\phi_{\mathtt{F}}$ by the strict partial order $<$. Finally, 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 consider the program $P_{(\Phi,<)}$ consisting of the rules in Line 1-13, the facts for the preference and optimize statements corresponding to $(\Phi, <)$, and the auxiliary rules (ii) described above. Then, we can show that for two sets of atoms X and Y, $X \succ_{(\Phi,<)} Y$ holds iff $P_{(\Phi,<)} \cup H(X) \cup H'(Y)$ is satisfiable.

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 $X \succ_{\phi} Y$ by 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 `}.`

And 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 $\triangleleft$ directly correspond to `neg`, `and`, `pareto`, and `lexico`. For brevity, we refrain from further details and refer the reader to [7, 9] for formal definitions.

Optimization statements. Finally, it is instructive to see how common optimization statements are expressed in *asprin*.[2] A *#minimize* directive is of the form

$$\#minimize\{w_1@k_1, \boldsymbol{t}_1 : \boldsymbol{\ell}_1, \ldots, w_n@k_n, \boldsymbol{t}_n : \boldsymbol{\ell}_n\}$$

where each w_i and k_i is an integer, and $\boldsymbol{\ell}_i = \ell_{i_1}, \ldots, \ell_{i_k}$ and $\boldsymbol{t}_i = t_{i_1}, \ldots, t_{i_m}$ are tuples of literals and terms, 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 $(w_i@k_i, \boldsymbol{t}_i : \boldsymbol{\ell}_i)$ in M such that $X \models \boldsymbol{\ell}_i$. Then, for sets X, Y of atoms and minimize statement M as above, $X \succ_M Y$ if $\Sigma_k^X < \Sigma_k^Y$ and $\Sigma_{k'}^X = \Sigma_{k'}^Y$ for all $k' > k$.

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

```
#preference(sM,lexico){ -k :: name(sk) | (w@k,t:ℓ) ∈ M }.
#preference(sk,less(weight)){ w,(t) :: ℓ | (w@k,t:ℓ) ∈ M }.
#optimize(sM).
```

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

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

Note that by wrapping tuples $\boldsymbol{t}$ into $(\boldsymbol{t})$, we only deal with pairs $w, (\boldsymbol{t})$ rather than tuples of varying length.

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

4 Heuristic support in *asprin*

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 [11], it has already indicated great prospects by improving regular optimization in ASP [12] as well as qualitative preferences [8]. While the latter had to be realized by modifications to a SAT solver, in *asprin* we draw upon the integration with *clingo 4*'s declarative

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

heuristic framework [12] . Heuristic information is represented in a logic program by means of the dedicated predicate `_heuristic`. Different types of heuristic information can be controlled with *clingo 4*'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 choice 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 [12] for a details.

This framework seamlessly integrates into *asprin* by means of so-called *heuristic programs*, where heuristics for concrete preference types may be specified. For example, consider the following heuristic program for `less(weight)`:

```
_heuristic(holds(X),false,1) :- preference(P,less(weight)),
                                preference(P,_,_,for(X),_).
```

This tells the solver to decide first on formulas appearing in preference statements of type `less(weight)` and to assign *false* to them. As another example, we can replicate the modification of the sign heuristic proposed in [8] for `poset` as follows:

```
_heuristic(holds(X),sign,1) :- preference(P,poset),
                               preference(P,_,_,for(X),_).
```

The idea is to assign *true* when deciding on formulas of a `poset` preference statement. In general, the goal of these heuristic programs is to direct the search towards optimal solutions in such a way that fewer intermediate solutions have to be computed.

For activating the domain heuristics, the option `--heuristic=Domain` must be supplied. In addition, *asprin* provides an easy way to modify it from the command line via option `--domain-heuristic=<m>[,<v>]`; it turns on the domain heuristic and applies heuristic modifier `m` with value `v` (1 by default) to the formulas occurring in preference statements. For example, instead of adding the previous heuristic program for `poset`, we could have issued the option `--domain-heuristic=sign`.

As put forward in [13, 8], even domain heuristics alone may be used to compute optimal models by a single call to a solver. In other words, preference types may actually be implemented by domain heuristics. For example, the preference type `subset` can alternatively be implemented by the following heuristic program

```
_heuristic(holds(X),false,1) :- preference(P,subset),
                                preference(P,_,_,for(X),_).
```

which guarantees that the first answer set computed is (already) optimal. Similarly, the following heuristic program implements a more sophisticated version of `poset`:

```
_heuristic(holds(F),true,1) :- preference(P,poset),
                               preference(P,_,_,for(F),_),
                               assigned(P,G) : poset(P,G,F).

assigned(P,G) :- poset(P,G,_),     holds(G).
assigned(P,G) :- poset(P,G,_), not holds(G).
```

With `poset/3` defined as in Section 3, predicate `assign/2` represents that a formula is assigned by the solver. Given this heuristic program, the solver prefers to satisfy

formulas whose dominating formulas are already assigned. As shown in [8], such a heuristic guarantees that the first optimal model computed is optimal. The *asprin* library includes such heuristic program also for `aso`. Using them, there is no need for checking the optimality of a solution. For this case, option `--mode=heuristic` tells *asprin* to avoid the check and activate the domain heuristic.

5 Using the *asprin* System

asprin is implemented in Python (2.7) and consists of a parser along with a solver that uses *clingo 4*'s Python library. This library 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 approach 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 input of *asprin* consists of a set of ASP files structured by means of *clingo 4*'s `#program` directives into base, preference, and heuristic programs. Base programs consist typically of a problem instance and encoding, and may contain a preference specification (just as with $\#minimize$ statements).[3] 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. Similarly for '`#program heuristic.`' and '`#program heuristic(t).`'. Among all the type-specific preference and heuristic programs in the input files, *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)`', and when using option `--mode=heuristic` there must also be a corresponding heuristic program '`heuristic(t)`'.[4] *asprin*'s implementation relies on the correctness of preference and heuristic programs. In other words, if the preference (or heuristic) programs implement correctly the corresponding preference types, then *asprin* also functions correctly, as shown in [9].

asprin's parser starts translating preference and optimization statements as explained in Section 3. Then every atom a appearing in a weighted formula is reified into `_holds(`a`,0)` adding a rule of form '`_holds(`a`,0) :-` a`.`' to the base program. Similar auxiliary rules are added for handling Boolean formulas. The successive answer sets computed by *asprin* are reified into atoms of the form `_holds(`t_F`,n)` where `n` takes successively increasing integer values starting with 1. Next, preference programs are slightly modified for comparing answer sets numbered `m1` and `m2`. Atoms of the form `holds(t)` and `holds'(t)` are translated into `_holds(t,m1)` and `_holds(t,m2)`, respectively. After parsing, the base program generated by the parser is solved by *clingo*. If the program is unsatisfiable, then *asprin* terminates and returns `UNSAT`. Otherwise, *asprin* enters a loop, where the last generated answer set is reified into facts of form `_holds(`t_F`,n)`; the preference program is grounded setting `m1` to `0` and `m2` to `n`; and *clingo* solves the resulting program. If a new answer set is found,

[3] If no preference specification is given, *asprin* computes answer sets of the base program.

[4] In this case, for computing a single optimal model no preference program is needed.

asprin returns to the beginning of the loop, and otherwise it returns the last answer set found. By construction, this last answer set is optimal wrt the preference in focus.

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). 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.

6 Empirical evaluation of *asprin*

In [9], we contrasted *asprin* (1.0) 's performance with that of *clingo 4.4* on basic weight- and subset-based preferences. We begin with extending this study to investigate the impact of heuristic information on *asprin*'s performance. To this end, we consider in Table 1 the benchmarks from [9] and solve them by increasing the heuristic influence on *asprin*'s search. We ran all benchmarks with *asprin 1.1* using *clingo 4.5* on a Linux

Benchmark \ System	$asprin_w$	$asprin_w$+s	$asprin_w$+l	$asprin_w$+f	$asprin_s$	$asprin_s$+s	$asprin_s$+l	$asprin_s$+f
Ricochet (30) 20.00	432 (8, 4)	407 (7,4)	68 (1, 0)	71 (1, 0)	365 (8,3)	461 (7,10)	69 (1, 0)	71 (1, 0)
Timetabling(12)23687.75	345 (285, 3)	255 (202,2)	900 (4,12)	6 (1, 0)	217 (144,2)	21 (18, 0)	900 (2, 12)	5 (1, 0)
Puzzle (7) 580.57	82 (2, 0)	112 (2,0)	136 (2, 0)	416 (2, 1)	31 (1,0)	32 (1, 0)	21 (1, 0)	51 (1, 0)
Crossing (24) 211.92	104 (42, 1)	98 (35,0)	805 (19,20)	387 (6, 6)	0 (6,0)	1 (6, 0)	7 (9, 0)	3 (1, 0)
Valves (30) 56.63	69 (7, 0)	65 (6,0)	460 (8,11)	715 (0, 22)	38 (4,0)	39 (4, 0)	339 (4, 6)	673 (0, 21)
Expansion (30) 7501.87	216 (299, 0)	10 (15,0)	38 (7, 0)	12 (3, 0)	64 (295,0)	14 (54, 0)	4 (4, 0)	3 (1, 0)
Repair (30) 6750.73	76 (48, 0)	15 (47,0)	71 (3, 2)	8 (2, 0)	8 (43,0)	3 (11, 0)	1 (1, 0)	1 (1, 0)
Diagnosis (30) 1669.00	196 (341, 3)	76 (66,0)	43 (4, 0)	118 (3, 2)	19 (338,0)	2 (39, 0)	0 (1, 0)	0 (1, 0)
$\varnothing(\varnothing, \Sigma)$	190 (129,11)	130 (48,6)	315 (6,45)	217 (2, 31)	93 (105,5)	72 (18,10)	168 (3, 18)	101 (1, 21)

Table 1. Comparing *asprin* with different heuristic settings

machine with an Intel Dual-Core Xeon 3.4 GHz processor, imposing a limit of 900 s and 4 GB of memory per run. A timeout is counted as 900 s. Each entry in Table 1 gives average time and in parentheses the number of enumerated models and timeouts. The number of enumerated models reflects how well *asprin* converges to the optimum. Each group of four data columns contains results from running *asprin* in its default setting and heuristics modifying `sign`, `level` and both, viz. `false`. The first group deals with *w*eight-based optimization and the second with *s*ubset-based optimization. Overall each heuristic modification improves over the standard in terms of runtime and convergence. This somewhat holds for timeouts as well, though certain heuristic settings degenerate on specific classes. Generally, we observe that the stronger the heuristic influence, the better *asprin* converges to the optimum. Interestingly, the best runtime is however obtained with the least interfering strategy, simply preferring negative signs for preference elements. On the other hand, classes like *Puzzle* are resistant to heuristic manipulations and weight-based optimization is even worse in this case. Here, convergence is immediate and cannot be improved by heuristic means. The bad performance can thus be explained by the interference of the heuristics with the final UNSAT problem needed for establishing optimality. Just modifying the `sign` thus appears as the best overall compromise, boosting convergence without overly hindering the final UNSAT proof. Otherwise, the best heuristic modification must be decided case-by-case.

Our next series of experiments aims at comparing the general-purpose approach of *asprin* with dedicated implementations of `aso` [14] and `poset` [8] preferences. In both cases, we use their benchmark generators and sets to contrast the approaches. The experimental settings are the same as above.

First, we compare *asprin* with the system for `aso` preferences [14]; it implements a branch-and-bound approach in C++ and calls *clingo* each time from scratch via a system call. We refer to it as *aso*. We also used the benchmark generator from [14] 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 amounts to an aggregation of `aso` preferences with `lexico` in *asprin*$_{l+a}$. The generator accounts for this by assigning a higher rank to half of the `aso` rules. The results of comparing both systems on both sets of random benchmarks are shown in Table 2. Each cell gives average runtime and number of timeouts. We see that the general-purpose approach of *asprin* is comparable with the dedicated approach of [14] on their benchmark set. On the other hand, we observed with *asprin* a very fast convergence, so that no real difference can be expected on these benchmarks.

n	*aso*	*aso*$_l$	*asprin*$_a$	*asprin*$_{l+a}$
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)
$\varnothing(\Sigma)$	84 (14)	95 (14)	89 (14)	91 (14)

Table 2. Comparing *asprin* with *aso*

Benchmark\System	*satpref*	*satpref*+`s`	*satpref*+`H`	*asprin*$_p$	*asprin*$_p$+`s`	*asprin*$_p$+`H`
0.0	0 (29, 0)	0 (1, 0)	0 (1, 0)	1 (16, 0)	0 (2, 0)	0 (1, 0)
0.00621	0 (35, 0)	0 (1, 0)	90 (1, 6)	1 (17, 0)	1 (2, 0)	1 (1, 0)
0.01243	1 (75, 0)	1 (3, 0)	118 (1, 7)	6 (26, 0)	2 (3, 0)	3 (1, 0)
0.02486	8 (388, 0)	6 (10, 0)	635 (1, 38)	55 (74, 0)	9 (8, 0)	64 (1, 4)
0.04972	67 (1463, 2)	16 (36, 0)	900 (0,100)	318 (203, 16)	26 (17, 0)	176 (1, 14)
1.0	850 (10315,88)	243 (590,10)	177 (1, 12)	856 (323, 92)	174 (96, 0)	280 (1, 24)
$\varnothing(\varnothing, \Sigma)$	154 (2051,90)	44 (107,10)	320 (1,163)	206 (110,108)	35 (21, 0)	88 (1, 42)
MAXSAT	54 (8849, 0)	9 (7, 0)	62 (1, 0)	835 (957, 31)	109 (31, 3)	171 (1, 6)
PBO/pbo-mqc-nencdr	5 (267, 0)	2 (2, 0)	664 (1, 88)	150 (207, 14)	9 (2, 0)	244 (1, 20)
PBO/pbo-mqc-nlogencdr	3 (228, 0)	1 (2, 0)	237 (1, 21)	110 (214, 3)	5 (2, 0)	141 (1, 15)
PSEUDO/primes	110 (396,18)	110 (1,18)	110 (1, 18)	215 (334, 27)	106 (5,17)	110 (1, 17)
PSEUDO/routing	346 (409, 4)	49 (1, 0)	50 (1, 0)	85 (475, 0)	4 (1, 0)	86 (1, 1)
Partial-MINONE	14 (2, 0)	14 (2, 0)	7 (1, 0)	24 (2, 0)	24 (1, 0)	25 (1, 0)
$\varnothing(\varnothing, \Sigma)$	88 (1692,22)	31 (2,18)	188 (1,127)	236 (365, 75)	43 (7,20)	129 (1, 59)

Table 3. Comparing *satpref* and *asprin* under different heuristic settings

Next, we compare *asprin* with the system *satpref* for `poset` preferences [15]. Interestingly, *satpref* not only extends the SAT solver *minisat* with branch-and-bound-based optimization but also uses heuristic support for boosting optimization. Table 3 contains the results of our comparison on benchmarks from [15]. The first six lines of data stem from 600 random instances (each with 500 variables and 1750 clauses), in which an order $a > b$ between variables a and b is generated with the probabilities given in the left column. The second six lines of data stem from instances taken from various competitions (cf. [15]). As above, we compare both systems in their basic setting and with `sign`-based heuristics. In addition, we contrast the declarative heuristics from the end of Section 4 (*asprin*$_p$+`H`) with its hard-coded counterpart in *satpref*+`H`. Such a heuris-

tic ensures that the first found model is optimal. In fact, as above, the best results with both systems are obtained with a light `sign`-based heuristics. The slight edge of *satpref* over *asprin*$_p$ is due to additional grounding efforts (given that problems are expressed in ASP). Despite this, the experiments show that the general-purpose approach of *asprin* is overall comparable with the dedicated approach of *satpref*.

7 Discussion

We have presented *asprin*, a general and flexible ASP-based system for representing and evaluating combinations of quantitative and qualitative preferences. We presented *asprin*'s first-order modeling language and showed how existing (and future) preferences can be expressed in *asprin*. We showed that our general-purpose approach matches the performance of dedicated systems for `aso` and `poset` preferences. Moreover, we demonstrated how well-chosen heuristics can boost the optimization process.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL **7**(3) (2006) 499–562
4. Brewka, G.: Logic programming with ordered disjunction. In Proceedings of AAAI, AAAI Press (2002) 100–105
5. Brewka, G., Niemelä, I., Truszczyński, M.: Answer set optimization. In Proceedings of IJCAI, Morgan Kaufmann (2003) 867–872
6. Brewka, G.: Complex preferences for answer set optimization. In Proceedings of KR, AAAI Press (2004) 213–223
7. Son, T., Pontelli, E.: Planning with preferences using logic programming. Theory and Practice of Logic Programming **6**(5) (2006) 559–608
8. Di Rosa, E., Giunchiglia, E., Maratea, M.: Solving satisfiability problems with preferences. Constraints **15**(4) (2010) 485–515
9. Brewka, G., Delgrande, J., Romero, J., Schaub, T.: asprin: Customizing answer set preferences without a headache. In Proceedings of AAAI, AAAI Press (2015) 1467–1474
10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
11. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. Annals of Mathematics and Artificial Intelligence **44**(4) (2005) 373–399
12. Gebser, M., Kaufmann, B., Otero, R., Romero, J., Schaub, T., Wanko, P.: Domain-specific heuristics in answer set programming. In Proceedings of AAAI, AAAI Press (2013) 350–356
13. Castell, T., Cayrol, C., Cayrol, M., Le Berre, D.: Using the Davis-Putnam procedure for an efficient computation of preferred models. In Proceedings of ECAI, Wiley (1996) 350–354
14. Zhu, Y., Truszczyński, M.: On optimal solutions of answer set optimization problems. In Proceedings of LPNMR. Springer (2013) 556–568
15. Di Rosa, E., Giunchiglia, E.: Combining approaches for solving satisfiability problems with qualitative preferences. AI Communications **26**(4) (2013) 395–408