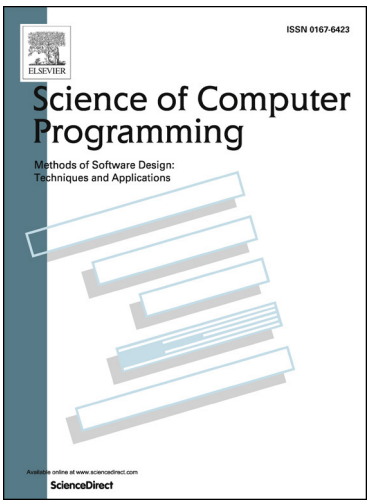
Accepted Manuscript



Higher-Order Logic Programming: an Expressive Language for Representing Qualitative Preferences

Angelos Charalambidis, Panos Rondogiannis, Antonis Troumpoukis

PII: S0167-6423(17)30190-9
DOI: http://dx.doi.org/10.1016/j.scico.2017.09.002
Reference: SCICO 2142

To appear in: *Science of Computer Programming*

Received date: 7 January 2017
Revised date: 5 September 2017
Accepted date: 7 September 2017

Please cite this article in press as: A. Charalambidis et al., Higher-Order Logic Programming: an Expressive Language for Representing Qualitative Preferences, *Sci. Comput. Program.* (2017), http://dx.doi.org/10.1016/j.scico.2017.09.002

Highlights

- A unifying approach for representing qualitative preferences is introduced.
- The proposed approach uses a simple fragment of higher-order logic programming.
- The new approach can represent a broad class of preference relations and operators.
- Apart from tuple-preferences, the framework can represent preferences over sets.

Higher-Order Logic Programming: an Expressive Language for Representing Qualitative Preferences

Angelos Charalambidis[a,b], Panos Rondogiannis[a], Antonis Troumpoukis[a,b]

[a] *Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece*
[b] *Institute of Informatics and Telecommunications, NCSR Demokritos, Athens, Greece*

Abstract

We consider the problem of concisely representing and handling preferences in logic programming and relational databases. Our starting point is the well-known proposal developed in [J. Chomicki, *Preference formulas in relational queries*, ACM TODS, 28(4):427-466, 2003] which advocates the embedding of first-order preference formulas into relational algebra through a *winnow* operator that is parameterized by a database relation and a preference formula. We argue that despite its elegance, the above proposal has certain shortcomings: only *intrinsic* preference formulas are supported, the preference relations and the preference queries are expressed in two different languages, and there is no direct way to define alternative operators beyond winnow. We propose the use of higher-order logic programming as a logical framework that remedies the above deficiencies. In particular, the proposed framework supports both intrinsic and extrinsic preference formulas, it can represent both preference relations as-well-as queries, and it can be used to define a variety of interesting alternative operators beyond winnow. We demonstrate the feasibility of our approach by presenting an implementation and an experimental evaluation of all the proposed concepts in the higher-order logic programming language Hilog.

1. Introduction

Preferences are being studied in many sciences (such as philosophy, economics, psychology, and so on) because they are a vital part of everyday human life. Research in preferences is also very active in computer science, mostly in areas such as artificial intelligence, machine learning, databases, and logic programming. This paper contributes to the area of *preference representation* and our results can be applied for designing and implementing more expressive preference representation languages that are based on higher-order logic programming. Alternatively, our results can be perceived as a logical framework for expressing and manipulating preferences.

Email addresses: a.charalambidis@di.uoa.gr (Angelos Charalambidis), prondo@di.uoa.gr (Panos Rondogiannis), antru@di.uoa.gr (Antonis Troumpoukis)

Preference representation formalisms are divided in two main categories [1]: in the *qualitative* approach preferences are expressed by comparing different objects ("I prefer beer over wine"), while in the *quantitative* one numerical values are used to express our degree of preference ("My preference in beer is 0.9 while in wine it is 0.2"). It has been argued that the qualitative approach is more general than the quantitative one (see Example 1.2 in [2][page 428]). A discussion of the techniques that have been developed in databases and logic programming regarding these two main approaches, will be given in Section 8. In this paper we focus exclusively on the qualitative approach.

The starting point of our research is an influential proposal by J. Chomicki [2] for representing preferences in a qualitative way in the context of relational database systems. Chomicki's approach is based on the following two ideas:

- Preferences between tuples of a database relation are specified using binary *preference relations*; these relations are defined using first-order formulas, called *preference formulas*.
- A new relational algebra operator, called *winnow*, is introduced which takes two parameters: a database relation and a preference formula. The winnow operator selects from its input relation the most preferred tuples according to the given preference formula.

The approach advocated by Chomicki, despite groundbreaking, has certain limitations (some of which are recognized and discussed in [2]). First of all, in the framework of [2] only *intrinsic* preference formulas can be defined, namely formulas that establish the preference relation between two tuples solely on the basis of the values occurring in these tuples. Second, the preference relations and the preference queries are expressed in two different languages, namely, first-order logic and SQL extended with the winnow operator, which makes the approach less uniform. Finally, there is no way to define *directly* other operators apart from winnow (such as, for example, an operator that returns the second-best tuples from a given relation according to a preference formula).

In this paper we propose the use of higher-order logic programming as a logical framework that remedies all the above deficiencies. The key idea behind our proposal is that *since preferences are relations, and since we need to define operators over relations (such as* winnow*), and some times even preferences over sets, a higher-order language can offer increased representation capabilities.* In particular, we demonstrate that higher-order logic programming can be used to express both intrinsic and extrinsic preference formulas, it can represent both preference relations as-well-as queries, and it can be used to define a variety of interesting alternative operators beyond winnow. We demonstrate the feasibility of our approach by presenting an implementation of all the proposed concepts in the higher-order logic programming language Hilog [3]. The main contributions of the paper can be summarized as follows:

1. We argue that higher-order logic programming is a very expressive framework for representing and manipulating qualitative preferences. A significant advantage of our approach is that preference formulas as-well-as

operators that are parameterized with such formulas can be expressed in the same language. Moreover, the seemingly more demanding case of *preferences over sets* [4] can be handled without extra notational overhead, because preferences over sets are essentially second-order relations and can therefore be encoded easily in our higher-order language.

2. We identify a new and significant (in our opinion) application area for higher-order logic programming. We substantiate this claim by providing an implementation of all the proposed ideas in Hilog. Higher-order logic programming has been around for many years [5, 3, 6], it has recently been given a standard denotational and proof-theoretic semantics [7, 8], and we feel that it deserves to be further developed and used since it extends in an elegant way the classical logic programming framework.
3. We provide a declarative and uniform framework for representing preferences, which attempts to bridge the (largely) independent research results on preference representation that have been developed by the communities of logic programming and relational databases (see Section 8 for a detailed account of the main results in these two areas). Moreover, we pave the road for the development of specialized techniques that would enhance higher-order logic programming so as that it can better handle and manipulate preferences. Such techniques include custom tailored implementation strategies, optimizations, specific language extensions, and so on; such issues are further discussed in the concluding section.
4. We provide experimental results that suggest that the proposed technique has a potential for further development. In particular we show that by adopting some simple optimizations, the performance of our implementation can be significantly enhanced. This leaves space for the development of more sophisticated optimizations that would further boost the performance of the proposed approach.

The rest of the paper is organized as follows: in Section 2 we present the main concepts behind the work of J. Chomicki as introduced in [2] and as extended in [4] in order to handle preferences over sets of tuples. Section 3 outlines the basic notions regarding higher-order logic programming. In Section 4 we demonstrate how higher-order logic programming can be used to concisely represent and manipulate preferences and how it bypasses the main shortcomings of [2]. In Section 5 we demonstrate that the proposed higher-order logic programming approach can also be used to elegantly represent preferences over sets of tuples. In Section 6 we present the details of an implementation of the proposed ideas in the higher-order logic programming language Hilog, together with the optimizations that we have adopted. Section 7 discusses the performance of the proposed approach on various experiments we have performed. Section 8 discusses related work regarding other preference representation formalisms that have been proposed in both databases and logic programming. Finally, Section 9 concludes the paper with discussion on possible future research directions.

The present article extends and revises the conference paper [9]. More specifically, this article extends Section 6 with new optimizations, it contains the new Section 7 which gives a performance evaluation of the proposed technique, it

has a more detailed comparison with related work (Section 8), and it contains new examples and more polished material in most of the remaining sections.

2. Qualitative Preferences and Databases

In this section we present the main ideas for representing preferences in a relational (SQL) database context following the approach proposed in [2]. We start by the definition of a "preference formula". In the following, by "built-in predicates" we mean any standard SQL predicate such as equality, inequality, arithmetic comparison operations, and so on.

Definition 1 ([2]). Let t_1, t_2 denote tuples of a given database relation. A *preference formula* $C(t_1, t_2)$ is a first-order formula defining a preference relation $\succ_C$ in the standard sense, namely, $t_1 \succ_C t_2$ iff $C(t_1, t_2)$ holds. An *intrinsic preference formula* (or *ipf*) is a preference formula that uses only built-in predicates.

Intuitively, an intrinsic preference formula is one that establishes the preference relation between two tuples solely on the basis of the values occurring in these tuples. *Extrinsic* preference formulas may use not only built-in predicates, but also other constructs (such as for example database relations, properties of the relations from which the tuples have been selected, and so on). For example, if we prefer every tuple of a given relation r over every tuple of another relation s, then this is an extrinsic preference because it depends also on the origin of the tuples and not only on their attributes. The framework of [2] focuses almost exclusively on intrinsic preference formulas.

Example 1. Consider the `movie(ID,Name,Director,Genre,Runtime,Rating)` relation represented by the following table:

ID	Name	Director	Genre	Runtime (min)	Rating
m1	The Godfather	Coppola	drama	175	9.2
m2	The Green Mile	Darabont	drama	189	8.5
m3	Goodfellas	Scorsese	drama	146	8.7
m4	The Big Lebowski	Coen	comedy	117	8.2
m5	Forrest Gump	Zemeckis	comedy	142	8.8
m6	Inception	Nolan	sci-fi	148	8.8

Suppose we want to express the following preference $\succ_{C_1}$: "I prefer one `movie` tuple over another iff their genres are the same and the rating of the first is higher". This can be defined by the following ipf formula C_1:

$$(i, n, d, g, m, r) \succ_{C_1} (i', n', d', g', m', r') \equiv (g = g') \wedge (r > r')$$

The above preference means that, for example, I prefer "`The Godfather`" from the "`Goodfellas`", because they have the same genre but the former has a higher rating than the latter.

As another example on the same relation, consider the following preference: "I prefer a movie from another one if the former lasts for less than or equal to 150 minutes while the latter does not". The corresponding preference relation $\succ_{C_2}$ can be defined by the following ipf formula C_2:

$$(i, n, d, g, m, r) \succ_{C_2} (i', n', d', g', m', r') \equiv (m \leq 150) \wedge (m' > 150)$$

According to the preference relation $\succ_{C_2}$, I prefer the "`Forrest Gump`" from "`The Godfather`", despite the fact that the latter has a higher rating. □

2.1. The Winnow Operator

In order to select the "best" tuples from a given relation r based on a preference formula C, SQL is enriched with a new operator called *winnow*, whose formal definition is the following [2]:

Definition 2. Let r be a relation and let C be a preference formula defining a preference relation $\succ_C$. The *winnow* operator is defined as

$$w_C(r) = \{t \in r : \neg\exists t' \in r \text{ such that } t' \succ_C t\}$$

In [2] a *preference query* is defined as an SQL query that contains at least one occurrence of the winnow operator, as the following examples demonstrate.

Example 2. Continuing Example 1, the preference query $w_{C_1}(\texttt{movie})$ returns the following tuples:

ID	Name	Director	Genre	Runtime (min)	Rating
m1	The Godfather	Coppola	drama	175	9.2
m5	Forrest Gump	Zemeckis	comedy	142	8.8
m6	Inception	Nolan	sci-fi	148	8.8

Moreover, the preference query $w_{C_2}(\texttt{movie})$ returns the following tuples:

ID	Name	Director	Genre	Runtime (min)	Rating
m3	Goodfellas	Scorsese	drama	146	8.7
m4	The Big Lebowski	Coen	comedy	117	8.2
m5	Forrest Gump	Zemeckis	comedy	142	8.8
m6	Inception	Nolan	sci-fi	148	8.8

In our framework we will be able to define alternative operators beyond winnow, and therefore our queries can be more general. □

2.2. Composition of Preference Relations

Preference relations can be composed in order to form more complex ones. In the framework of [2], in order to compose two preference relations one must compose the corresponding formulas, creating in this way more complicated ones. Two examples of such composition operations are the following:

- The *Boolean composition* of two preference relations (such as union, intersection and difference) can be captured by Boolean operations on the corresponding preference formulas. For example, the preference relation $\succ_{C_1 \wedge C_2} = \succ_{C_1} \cap \succ_{C_2}$ is captured by the formula:

$$t_1 \succ_{C_1 \wedge C_2} t_2 \equiv (t_1 \succ_{C_1} t_2) \wedge (t_1 \succ_{C_2} t_2)$$

- The *prioritized composition* $\succ_{C_1 \triangleright C_2}$ of two preference relations C_1 and C_2 has the intuitive meaning of "prefer according to C_2 unless C_1 is applicable", and can be defined as follows:

$$t_1 \succ_{C_1 \triangleright C_2} t_2 \equiv (t_1 \succ_{C_1} t_2) \vee ((t_1 \sim_{C_1} t_2) \wedge (t_1 \succ_{C_2} t_2))$$

  where $t_1 \sim_{C_1} t_2$ is the indifference relation defined by the formula $\neg(t_1 \succ_{C_1} t_2) \wedge \neg(t_2 \succ_{C_1} t_1)$.

The following example illustrates the use of such operators.

Example 3. Consider the `movie` relation of Example 1. If my preference of movies is the prioritized composition $C_2 \triangleright C_1$ of C_2 and C_1, then this means that I have a primary preference for the movies that have a duration that is less than or equal to 150 minutes, and I have a secondary preference for the high rating (among the ones that have the same genre). Thus, the preference query $w_{C_2 \triangleright C_1}(\texttt{movie})$ returns the following tuples:

ID	Name	Director	Genre	Runtime (min)	Rating
m3	Goodfellas	Scorsese	drama	146	8.7
m5	Forrest Gump	Zemeckis	comedy	142	8.8
m6	Inception	Nolan	sci-fi	148	8.8

Notice that my primary preference for the duration of the movies has excluded some highly rated ones. □

Through higher-order logic programming we can directly define generic composition operators that can be applied to compose arbitrary preference relations (without the need to manipulate preference formulas).

2.3. Preferences over Sets

There exist many applications that require expressing preferences over sets of tuples. This case, which is more demanding and general than the case of preferences over tuples, was recently considered in [4]. Notice that the sets involved in the work of [4] are assumed to be of fixed cardinality, an assumption which we also adopt in our implementation presented in Section 6. The main ideas behind the proposal of [4] can be illustrated by a simple example.

Example 4. Assume that I want to watch three movies, and I want to select them, based on some given preferences, out of our usual movie table. My preferences are the following:

- C_1: I prefer that the sum of the ratings of the movies is maximized.
- C_2: I prefer to watch at least one comedy.
- C_3: I prioritize C_2 over C_1.

The above preferences are defined over three-element sets of movies and therefore a new method seems to be required for their representation. □

The authors of [4] define a specialized approach for treating preferences over subsets. First they remark that for each preference there exists one or more "quantities of interest", which they call *features*. For example, for C_1 the relevant feature is the sum of the ratings of the movies, for C_2 the relevant feature is the number of comedies, and for C_3 both previous features are relevant. Given a set, we can construct a vector of all features based on the user preferences, ie., the *profile* of the set. The "best sets" are those that have the "best profiles".

Preference representation in [4] is performed as follows. For each feature $\mathcal{F}_i$, a function is defined that returns the result of an SQL query. Let `$S` be a variable denoting any three-element subset of the `movie` relation. Then, the following functions compute the total rating and the number of comedies in `$S`:

- $\mathcal{F}_1$(`$S`): `SELECT sum(rating) FROM $S`
- $\mathcal{F}_2$(`$S`): `SELECT count(genre) FROM $S WHERE genre='comedy'`

Preferences over sets are defined by formulas over the above functions:

$$s_1 \succ_{C_1} s_2 \equiv \mathcal{F}_1(s_1) > \mathcal{F}_1(s_2)$$
$$s_1 \succ_{C_2} s_2 \equiv (\mathcal{F}_2(s_1) \geq 1) \wedge (\mathcal{F}_2(s_2) = 0)$$
$$s_1 \succ_{C_3} s_2 \equiv s_1 \succ_{C_2 \triangleright C_1} s_2$$

The winnow operator is then used in order to select the "best" among all three-element subsets of the `movie` relation according to the above preference relations. In order for this to be done, the most direct approach proposed in [4] is to enumerate all fixed-size subsets and check them one by one. Notice that dealing with subsets of relations is in general an inherently inefficient task. For this reason, certain optimizations are discussed in [4] that can potentially alleviate the inefficiency problem in specific cases.

The problem with the proposal of [4] is that the simple first-order logic language for expressing tuple preferences has to adapt significantly in order to accommodate set preferences, and the resulting formalism is more complicated. As a general comment on the above ideas, we could say that the *features* of [4] are second-order predicates in disguise, and for this reason they can be represented directly and very elegantly in our framework.

2.4. Discussion

The framework for expressing qualitative preferences in relational databases developed in [2] and extended in [4], appears to have many advantages and it is relatively simple. However, it has certain shortcomings which, if resolved, could

result in a more expressive formalism. In this subsection we describe in detail the main shortcomings of the techniques in [2, 4]. In subsequent sections we demonstrate how these issues are resolved in our framework.

First of all, the framework of [2] uses two different languages: the preference relations are expressed using first-order logic while preference queries are expressed using SQL extended with the winnow operator. Moreover, in the case of preferences over sets, yet another formalism (namely that of *features* and *profiles*) is introduced. As we argue in the coming sections, preference relations and queries can be expressed in a single language, simplifying in this way the representation of preferences. The differences in the conciseness of the representation are even more apparent in the case of set preferences, where we avoid the concepts of *features* and *profiles* introduced in [4], and our representation is a simple extension of the one used for tuple preferences.

A second characteristic of [2] which we would like to avoid, is the restriction to *intrinsic* preference formulas. There exist many natural preference formulas that are *extrinsic* (see also the discussion in [2][Section 7.3]). Some extrinsic preferences can be simulated using intrinsic formulas, but this is not always possible or convenient. Moreover, as we are going to see in Section 5, the use of extrinsic preferences is very important in the case of set preferences.

A final issue of [2] is the restriction of having a single preference manipulation operator, namely winnow. To be fair, a few more operators similar to winnow are presented in [2]. However, all these operators are defined in a language that is different from both first-order logic and SQL (additional set-theoretic and relational notations are used). In other words, in the framework of [2], new operators can only be "custom tailored" and they can not be defined nor implemented with the available linguistic resources. In our framework, new operators can easily and naturally be defined as higher-order predicates.

3. Higher-Order Logic Programming

Higher-order logic programming [5, 3, 7] extends traditional first-order logic programming with higher-order constructs. In this section we define some common higher-order logic programming concepts and some basic predicates that will be used throughout the paper. A basic familiarity with some form of higher-order programming (either logic or functional) is assumed.

The most popular higher-order extension is to allow the programmer to define predicates that accept other predicates as arguments, and variables to occur in places where predicates typically occur. For example, the following program defines the transitive closure of a given relation `R`:

```
tc(R,X,Y) :- R(X,Y).
tc(R,X,Y) :- R(X,Z),tc(R,Z,Y).
```

An advantage of predicates such as `tc` is that they can be used to achieve a *generic* style of programming: to compute the transitive closures of different relations, we invoke `tc` with different parameters (avoiding to write a transitive closure predicate for each different relation). As an example, the query:

```
?- tc(parent,john,Y).
```

will return all the ancestors of `john` while the query:

```
?- tc(graph,v1,V).
```

will return all vertices that are reachable from vertex `v1` of a given binary relation `graph`.

One interesting feature of higher-order languages, is the use of partial application, ie., the ability to invoke a higher-order predicate with only some of its arguments. In order for this to be achieved, the single-tuple notation of classical logic programming is extended so as that a predicate can now have a sequence of tuples as successive parameters[1]. Then, we can invoke a predicate with only some of the tuples in the sequence. In this case we say that the predicate is *partially applied*. The partial application of a predicate yields a new predicate that expects the rest of the arguments and behaves exactly as a regular predicate. For example, we can write the `tc` predicate using a slightly different syntax:

```
tc(R)(X,Y) :- R(X,Y).
tc(R)(X,Y) :- R(X,Z),tc(R)(Z,Y).
```

Now, `tc(parent)` is an expression representing the transitive closure of the relation `parent` and can be used as an autonomous expression in the program and in queries. The idea of "partial application" can be further illustrated with the definition of two higher-order predicates that we will use in subsequent sections, namely the *conjunction* and *union* predicates over binary relations:

```
conj(R,Q)(X,Y) :- R(X,Y),Q(X,Y).
union(R,Q)(X,Y) :- R(X,Y).
union(R,Q)(X,Y) :- Q(X,Y).
```

Now the expression `union(tc(parent), tc(mother))` denotes the union of the two transitive closure relations while the expression `conj(tc(graph1), tc(graph2))` is the set of common edges that belong to the transitive closures of `graph1` and `graph2`.

The rest of this section contains some easy definitions of other higher-order predicates that we will use. Actually, some of them use a powerful feature of logic programming, namely negation-as-failure. A predicate that checks if a relation is empty can be implemented as follows:

```
nonempty(R) :- R(X).
empty(R) :- not nonempty(R).
```

The `nonempty` predicate succeeds if there exists an object that satisfies its argument. On the other hand, the `empty` predicate succeeds if `nonempty` fails for the same parameter. Other useful operators are the following:

```
minus(R,X)(Y) :- R(Y), not (X=Y).
diff(R,Q)(X) :- R(X), not Q(X).
```

[1] In the world of functional programming this idea is called *currying*.

The `minus` operator takes as a parameter a relation `R` and an object `X` and returns a relation that contains all the objects in `R` except for `X`. In this sense `minus` removes `X` from relation `R`. Similar to the `minus` operator is the `diff` operator that creates the set difference of its given two relations. Given the above definitions we can easily define the cardinality of a finite relation as:

```
size(R,0) :- empty(R).
size(R,N) :- R(X), size(minus(R,X),M), N is M+1.
```

In higher-order logic programming languages, queries of the following form are usually allowed:

```
?- tc(R)(john,mary).
```

The expected answer of such a query is not immediately obvious and this is actually a thorny subject among the higher-order logic programming approaches. The extensional approach [7] assumes that predicates denote sets and therefore two predicates that hold for the same elements are considered equal. An expected answer in the extensional approach is the substitution `R` = {`(john,mary)`}, namely the simplest set whose transitive closure contains the pair `(john,mary)`. On the other hand, the intensional approach [3] assumes that each predicate is represented by its name and therefore two predicates are considered equal only if their names are the same. Under the intensional approach, the answer to the aforementioned query depends on whether there exists a predicate defined in the program that satisfies the goal. In the case that there is no such predicate in the program, the goal fails. As we are going to see, we will need to evaluate queries involving uninstantiated predicate variables (such as the above) in the case of preferences over sets. The way that we handle such queries in our implementation will be described in Section 6.

4. Representing Preferences over Tuples in Higher-Order Logic Programming

In this section we demonstrate how preference relations over tuples as-well-as operators over preference relations, can be defined in higher-order logic programming. The language that we will be using does not exploit the full power of higher-order logic programming, because most of our examples will be written in "higher-order Datalog", namely higher-order logic programming without function symbols. There are only two exceptions to this issue:

- Our language supports tuples (in a restricted form) because preference relations are defined over tuples. In other words, we do not need arbitrary function symbols but we need our language to be able to handle tuples.
- In order to be able to define some operators over preference relations and some aggregate operations, we will use recursion and the usual arithmetic operations over the natural numbers. However, all the essential ideas from [2] can be implemented in our framework without ever using natural numbers or any function symbols.

In conclusion, our framework is essentially a higher-order version of Datalog that supports tuples. In this respect, one can also view our proposal as a *higher-order deductive database framework for representing preferences.*

4.1. Representing Database Relations

We follow the standard approach in deductive databases in which relations are represented by logic programming facts. In the rest of the paper we adopt the following notational convention: we assume that every predicate that represents a database relation, does not have many different attributes but instead a single attribute that is a tuple. For example, in order to represent the `movie` relation, we use a predicate that takes one argument that is a tuple, namely `movie((ID,Name,Director,Genre,Runtime,Rating))`. This approach allows us to write more generic operators on preference relations. The above convention has been adopted because the higher-order logic programming language that we will be using, namely Hilog, does not treat multiple arguments of a predicate as a unique entity that is a tuple.

4.2. Representing Preference Relations

We can represent a preference relation $\succ_C$ using a binary predicate `c_pref` with two arguments, each one of them being an n-ary tuple. For example, the atom `c_pref((X1,X2),(Y1,Y2))` corresponds to the formula $(\texttt{X1}, \texttt{X2}) \succ_C (\texttt{Y1}, \texttt{Y2})$. The preference formula C is encoded by the body of the rule defining `c_pref`. Notice that for the representation of intrinsic preference relations, we do not use the higher-order characteristics of our source language.

Example 5. Consider the `movie(ID,Name,Director,Genre,Runtime,Rating)` relation. We can represent the preference relations of Subsection 2.3 using predicates over tuples. Assume we have the preference relation C_1 of Example 1, namely: "I prefer one `movie` tuple over another iff their genres are the same and the rating of the first is higher". This can be represented as:

```
c1_pref((I1,N1,D1,G,M1,R1),(I2,N2,D2,G,M2,R2)) :-
        movie((I1,N1,D1,G,M1,R1)),
        movie((I2,N2,D2,G,M2,R2)),
        R1 > R2.
```

Consider now the preference relation C_2 from Example 1, namely "I prefer a movie from another one if the former lasts for less than or equal to 150 minutes while the latter does not". This can be represented by:

```
c2_pref((I1,N1,D1,G1,M1,R1),(I2,N2,D2,G2,M2,R2)) :-
        movie((I1,N1,D1,G1,M1,R1)),
        movie((I2,N2,D2,G2,M2,R2)),
        M1 =< 150, M2 > 150.
```

The use of the logic variables allows expressing other, more demanding, preference relations concisely. Assume we "prefer comedies from all other types of movies". This is represented as follows:

```
c3_pref((I1,N1,D1,comedy,M1,R1),(I2,N2,D2,G2,M2,R2)) :-
        movie((I1,N1,D1,comedy,M1,R1)),
        movie((I2,N2,D2,G2,M2,R2)),
        not (G2=comedy).
```

In the above examples we explicitly check that each tuple belongs to the `movie` relation, something that can not be expressed in the preference formulas of [2] (which only check properties of individual elements of tuples). This fact gives an advantage to our technique, because, as we are going to see below, this allows us to express extrinsic preferences. □

Example 6. Assume we prefer any tuple from a relation r over any tuple from another relation s. In the framework of [2] such a requirement can only be simulated, somewhat artificially, by adding an extra argument to each tuple that denotes the relation name to which the tuple belongs. In our case this can simply be written as:

```
c_pref(T1,T2) :- r(T1),s(T2).
```

Notice that our assumption that database facts only take a single attribute that is a tuple, allows us to use above only the variables `T1` and `T2` and to avoid listing all the attributes of relations `r` and `s`. □

Example 7. Assume that we prefer those movies that have the most popular directors (namely directors who have filmed the maximum number of movies). This is an extrinsic preference because we cannot compare two movie tuples based only on the information contained in the tuples. In order to solve this problem, a specialized approach is followed in [2] which is based on the construction of separate views through the use of SQL queries involving aggregate operators. In our case, this preference relation can be expressed as follows:

```
director_pref((I1,N1,D1,G1,M1,R1),(I2,N2,D2,G2,M2,R2)) :-
        movie((I1,N1,D1,G1,M1,R1)),
        movie((I2,N2,D2,G2,M2,R2)),
        director(D1,K1),
        director(D2,K2),
        K1 > K2.

movies_of_director(D)(I) :- movie((I,_,D,_,_,_)).

director(D,K) :- movie((_,_,D,_,_,_)),
                 size(movies_of_director(D),K).
```

Notice that in our case we also use an aggregate operator, namely the predicate `size` which was defined in Section 3. Moreover, we use for the first time a higher-order characteristic of our language: the partially applied expression `movies_of_director(D)` is a relation that contains all the different movie IDs that a director has filmed. A difference from [2] is that a unique language is used in order to express our preference relations. □

Example 8. We give another example of a natural extrinsic preference relation which can easily be encoded in our framework. Assume we "prefer movies that have an above-average rating over those that have a below-average rating". This preference requires the calculation of the average rating of all movies, and therefore it is an extrinsic one. It can be expressed in our setting as follows:

```
average_pref((I1,N1,D1,G1,M1,R1),(I2,N2,D2,G2,M2,R2)) :-
         movie((I1,N1,D1,G1,M1,R1)),
         movie((I2,N2,D2,G2,M2,R2)),
         size(movie,K),
         rating_sum(movie,S),
         R1 >= S/K, R2 < S/K.

rating_sum(Rel,0) :- empty(Rel).
rating_sum(Rel,S) :- Rel((I,N,D,G,M,R)),
                     rating_sum(minus(Rel,(I,N,D,G,M,R)),S1),
                     S is S1+R.
```

The `rating_sum` predicate calculates the sum of ratings of all tuples in the `movie` relation; dividing this sum by the size of the `movie` relation gives us the average rating of all movies. □

Another characteristic of our approach is that we can define preference relations over database relations that are defined recursively (known as IDBs in the deductive database literature); this is not generally possible in the approach of [2] (see the discussion in [2][Section 4.3, pages 439-440]).

Example 9. Consider the relation `edge((X,Y,C))`, which defines a weighted directed graph, where `C` denotes the cost of edge `(X,Y)`. We can formulate the shortest path problem, in a somewhat naive way, as follows:

```
naive_shortest(X,Y,C) :- winnow(path_pref,path)((X,Y,C)).

path_pref((X,Y,C1),(X,Y,C2)) :- path((X,Y,C1)),
                                path((X,Y,C2)), C1 < C2.

path((X,Y,C)) :- edge((X,Y,C)).
path((X,Y,C)) :- edge((X,Z,A)), path((Z,Y,B)), C is A+B.
```

In other words, the most preferred path is the one that has the smallest cost.□

The above program enumerates all paths from `X` to `Y`, and then uses `winnow` to select the most preferable one(s). We can write a more efficient version by embedding `winnow` inside the recursive definition of the `path` predicate. We believe that this is a nice consequence of a single language for representing preferences and operators on them, and gives another interesting application beyond the system described in [2]. This enhanced path idea was motivated by the "optimal subproblem property" for shortest distance, discussed in [10][page 94].

Example 10. The following program finds the shortest path from `X` to `Y` by first finding the *optimal* paths from every neighbor `Z` of `X` to `Y`.

```
enhanced_shortest(X,Y,C) :- winnow(path_pref,opt_path)((X,Y,C)).

path_pref((X,Y,C1),(X,Y,C2)) :- opt_path((X,Y,C1)),
                                opt_path((X,Y,C2)), C1 < C2.

opt_path((X,Y,C)) :- edge((X,Y,C)).
opt_path((X,Y,C)) :- edge((X,Z,C1)),
                     winnow(path_pref,opt_path)((Z,Y,C2)),
                     C is C1+C2.
```

Notice the use of `winnow` inside the recursive definition of `opt_path`. □

Both the naive and the optimized `path` predicates work correctly when the graph is acyclic. In a graph that contains cycles, there may exist an infinite number of paths that go from `X` to `Y`, because some paths can go around a cycle for an arbitrary number of times. One can easily extend the above program to take an extra parameter that restricts the length of the desired path to be less than or equal to the number of vertices in the graph. This modified path predicate can be found in `https://bitbucket.org/antru/holppref`.

4.3. Representing Composition Operators

A straightforward way to compose preference relations in our framework is to create new clauses that use in their bodies the predicates of the initial relations. For example in order to obtain the logical conjunction of two preference relations `c1_pref` and `c2_pref`, we can simply define:

```
c_pref(T1,T2) :- c1_pref(T1,T2),c2_pref(T1,T2).
```

However, in higher-order logic programming we can do better than this since we can have a generic `conj` operator, which is defined as follows:

```
conj(R,Q)(T1,T2) :- R(T1,T2),Q(T1,T2).
```

The advantage of the above approach is that in order to compose various preference relations we do not need to create new clauses (or new formulas as in the framework of [2]), but we can specify the compositions in combinatory form. For example `conj(c1_pref, union(c2_pref,c3_pref))` represents the conjunction of `c1_pref` with the union of `c2_pref` and `c3_pref`.

The prioritized composition of preference relations can be easily defined in an analogous way using a generic operator:

```
prioritized(C1,C2)(T1,T2) :- C1(T1,T2).
prioritized(C1,C2)(T1,T2) :- indifferent(C1)(T1,T2), C2(T1,T2).

indifferent(C)(T1,T2) :- not C(T1,T2), not C(T2,T1).
```

It is worth noting that there exist even more advanced composition operators for preference relations, such as the so-called *pareto* and *lexicographic* compositions [11, 2]. One can easily verify that these operators also have a direct and elegant encoding in higher-order logic programming.

Another characteristic of our approach is that it allows to define the transitive closure on preference relations by using the `tc` predicate given in Section 3. The following example motivates the need for this operator.

Example 11. Assume we have the following database of available items together with their color:

```
item((a1,black)).
item((a4,green)).
```

Moreover, assume we prefer black over red, red over blue, blue over yellow and yellow over green items. This can be expressed using the following facts:

```
color_pref((_,black), (_,red)).
color_pref((_,red), (_,blue)).
color_pref((_,blue), (_,yellow)).
color_pref((_,yellow), (_,green)).
```

Assume now that our `color_pref` relation is transitive. Instead of adding extra facts to the `color_pref` relation that would express its transitivity, we can instead use in our queries the relation `tc(color_pref)`. □

It should be noted that in the framework of [2] the transitive closure of a relation can not be directly specified (due to the inability of first-order logic to define transitive closure on finite structures). However, as shown in [2][Theorem 4.10], if a preference relation is defined through an ipf, then its transitive closure can also be defined through an ipf, which can be effectively constructed. In our case no special construction is required apart from the application of the `tc` predicate to the given preference relation. Moreover, in our case the transitive closure can also be applied to extrinsic preference relations.

4.4. Representing Operators on Preference Relations

An important characteristic of the proposed approach is its ability to define directly new operators on preference relations. These operators are in fact higher-order predicates that operate on database and preference relations. We start by recalling the *winnow* operator:

$$w_C(r) = \{t \in r : \neg\exists t' \in r \text{ s.t. } t' \succ_C t\}$$

The above definition can be directly transcribed in higher-order logic programming as follows:

```
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(T1), C(T1,T).
```

In the above definition observe that our assumption that database facts take a single attribute that is a tuple, allows us to use only the variables `T` and `T1` without caring about how many arguments the relation `R` has. If our facts did not use the single-tuple notation we would need to write a different *winnow* operator for each different tuple size. We can now use this operator to formulate queries such as:

```
?- winnow(c1_pref,movie)(T).
```

that will succeed for all tuples `T` that belong to the relation `movie` and are most preferred with respect to the relation `c1_pref`. Moreover, one can use the combinators defined in Section 4.3 to create more complex queries:

```
?- winnow(prioritized(c1_pref,c2_pref),movie)(T).
```

In a similar manner we can define other interesting operators. A simple variation of winnow is the *"iterated-winnow"* which is defined in [2][Section 8] as follows:

$$w_C^1(r) = w_C(r)$$
$$w_C^{n+1}(r) = w_C(r - \bigcup_{i=1}^{n} w_C^i(r))$$

The w_C^n operator selects the "n"-best tuples. For example, $w_C^2(r)$ returns the second-best tuples of r with respect to the preference relation C. In our framework w_C^n can be easily defined as a higher-order predicate:

```
w(C,R)(1)(T) :- winnow(C,R)(T).
w(C,R)(N)(T) :- N>1,M is N-1,
                winnow(C,diff(R,gen_union(M)(w(C,R))))(T).
```

where the `gen_union` operation is a generalized union operator (over an indexed family of sets) which can be defined as follows:

```
gen_union(1)(R)(X) :- R(1)(X).
gen_union(N)(R)(X) :- N>1,M is N-1,
                      union(gen_union(M)(R), R(N))(X).
```

Notice that the definition of `w` has a strong resemblance to the mathematical definition given previously. Given the definition of `w` one can retrieve the second-best tuples of a relation by posing the query:

```
?- w(c1_pref, movie)(2)(T).
```

Assume we want to return all the tuples up to a desired level. We can define the operator $wt_C^n(r) = \bigcup_{i=1}^{n} w_C^i(r)$ that uses w_C^n as follows:

```
wt(C,R)(N)(T) :- gen_union(N)(w(C,R))(T).
```

Finally, we can easily define the *ranking* operator [2], which ranks the elements of a relation r with respect to a preference C:

$$\eta_C(r) = \{(t, i) : t \in w_C^i(r)\}$$

The operator η can naturally be defined as follows:

```
eta(C,R)(T,I) :- size(R,N), between(1,N,I), w(C,R)(I)(T).
```

where `between` can be easily defined to return in its third argument all the integers between `1` and `N`.

5. Representing Preferences over Sets in Higher-Order Logic Programming

The approach we follow in order to express set preferences is a generalization of the techniques we have used in order to represent tuple preferences. More specifically, we now have to define *preference relations over sets*. This does not require any changes in our notation: in higher-order logic programming, predicates denote sets and therefore sets can be the parameters of other predicates.

Example 12. Consider the `movie` relation that we have been using as our running example. In the following we demonstrate how the example preference relations discussed in Subsection 2.3 can be represented in our framework.

1. I prefer the sum of the ratings of the movies to be the highest possible. This means that a set `S1` is preferable over a set `S2` if the sum of the ratings of the elements of `S1` is greater than that of the set `S2`. The `rating_sum` predicate, defined in Example 8, recursively calculates the cumulative rating of a given set by selecting an element from the set, calculating the sum of the ratings of the remaining elements and then adding the rating of the selected element.

```
rating_pref(S1,S2) :- rating_sum(S1,R1),
                      rating_sum(S2,R2),
                      R1 > R2.
```

2. I prefer to watch at least one comedy. In other words, a set `S1` is more preferred than another set `S2` if the former contains at least one comedy, while the latter does not.

```
comedy_pref(S1,S2) :- S1((_,_,_,comedy,_,_)),
                      not S2((_,_,_,comedy,_,_)).
```

3. Prioritize (1) to (2). In other words, choose the set that has the least cumulative rating, but if two sets have the same total rating, prefer the one that contains at least one comedy movie.

```
ratingcomedy_pref(S1,S2) :-
          prioritized(rating_pref,comedy_pref)(S1,S2).
```

Notice that we can directly use the `prioritized` operator we defined in Subsection 4.3 for preference relations over tuples. More generally, defining preference relations over sets in higher-order logic programming is an analogous task to that of defining preference relations over tuples. □

The use of the *winnow* operator is very similar as in the tuple case. For example, in order to find the "best" 3-element subsets of the `movie` relation with respect to the preference relation `rating_pref`, we simply need to ask the query:

```
?- winnow(rating_pref, subsetof(movie,3))(S).
```

In the above query the predicate `subsetof` takes two parameters, a relation `r` and a natural number `k` and returns all the subsets of `r` that have size `k`. The `subsetof` predicate can be implemented in various ways depending on the higher-order language adopted and on the applications aimed at. We will discuss in detail our implementation in the next section. The execution of the above query returns as bindings for the variable `S` all the "best' 3-element subsets with respect to `rating_pref`. The variable `S` in the above query is actually an uninstantiated predicate variable, ie., it represents a set. As we have already mentioned, the treatment of such queries depends on the higher-order language under consideration. The approach we have followed in order to treat such variables in a satisfactory way under Hilog, will be described in Section 6.

An advantage of the higher-order logic programming approach to preference representation is the ability to express non-intrinsic preferences over sets, as the following example demonstrates:

Example 13. Consider a (simplified) `movie` relation defined by the predicate `movie((ID, Name, PID))` where `ID` uniquely identifies the movie and `PID` refers to the id of the movie that this movie is a sequel of. We also have a `prequelOf` predicate that identifies a movie tuple that is a prequel of another one:

```
movie((m01, the_Godfather, null)).
movie((m02, the_Godfather_Part_II, m01)).
movie((m03, the_Godfather_Part_III, m02)).
movie((m04, kill_Bill_Vol_1, null)).
movie((m05, kill_Bill_Vol_2, m04)).
movie((m06, the_Sting, null)).

prequelOf((Z,_,_),(_,_,Z)).
```

Suppose that we prefer collections of movies that are complete, ie., collections that contain all parts of a movie. This can be expressed by the following preference relation over sets of movies:

```
movie_pref(S1,S2):- not seriesMissing(S1), seriesMissing(S2).

seriesMissing(S) :- S(X), movie(Y), inseries(X,Y), not S(Y).

inseries(X,Y) :- prequelOf(X,Y) ; prequelOf(Y,X).

preferred_sets(N)(S) :- winnow(movie_pref,subsetof(movie,N))(S).
```

Consider the following five queries:

```
?- preferred_sets(1)(S),S(T).
?- preferred_sets(2)(S),S(T).
?- preferred_sets(3)(S),S(T).
?- preferred_sets(4)(S),S(T).
?- preferred_sets(5)(S),S(T).
```

The first one will return a singleton set `S` that contains as the only tuple `T` the one that corresponds to "The Sting" movie. The second one will return the set `S` that contains the two tuples of the "Kill Bill" series. The third one will return two solutions: one that contains the three tuples concerning the "Godfather" series, and one that contains the "Kill Bill" series together with "The Sting". The fourth query will return a set containing the "Godfather" series together with "The Sting". Finally, the fifth query will return a set containing the "Godfather" series together with the "Kill Bill" series.

Notice also that the relation `movie_pref` is extrinsic since it generally depends on other tuples that are outside the subsets we are comparing. □

6. Implementation in Hilog and Optimizations

In order to assess the potential of higher-order logic programming for expressing qualitative preferences, we undertook an implementation of the proposed ideas in Hilog [3], a stable and well-known higher-order logic programming language. Our implementation has been realized in the XSB system [2], a mature, goal-oriented, top-down Prolog system that supports Hilog natively. In this section we describe the basic principles behind our implementation and the optimizations we have adopted in order to enhance its performance.

6.1. A Basic Unoptimized Implementation

For the case of preferences over tuples, it is relatively straightforward to perform a basic, unoptimized, implementation in Hilog. More specifically, the programs given in Section 4 as-well-as many other ones that we have experimented with, are compiled and executed by XSB directly[3] and run relatively efficiently as we are going to see in Section 7.

The case of preference queries over sets is more demanding than the queries over tuples, and required some nontrivial interventions from our side. As we have already seen in Section 5, in the case of preferences over sets the queries may involve uninstantiated variables that denote sets. In Hilog such queries are allowed but their interpretation is inconvenient for our purposes. More specifically, when a Hilog query contains a variable denoting a set, the implementation searches to find whether there exists *a predicate defined in the program* that could satisfy the given query. If there exists such a predicate, then its name is returned; otherwise, the query fails.

Example 14. Consider the following Hilog program:

```
p(Q) :- Q(a),Q(b).
q(a).
q(b).
```

[2] cf. `http://xsb.sourceforge.net/`

[3] The corresponding code and other similar examples can be retrieved from `https://bitbucket.org/antru/holppref`.

The query:

```
?- p(Q).
```

will return the answer `Q=q`. However, if there is no predicate `q` defined in the program, the above query will fail in Hilog. □

The reasons why the above handling of queries with uninstantiated predicate variables is inconvenient, is clarified by the following example.

Example 15. Recall Example 13 and consider the following query:

```
?- preferred_sets(3)(S).
```

which amounts to the equivalent query:

```
?- winnow(movie_pref,subsetof(movie,3))(S).
```

In order for Hilog to answer properly the above query, there must exist explicitly defined in the program predicates that denote three-element subsets of the `movie` relation (because otherwise the query will fail). □

In order for a Hilog implementation to work properly, the subsets of a given relation must be generated and asserted in the program as named predicates. Notice that there have been proposals for higher-order logic programming in which the implementation of queries involving uninstantiated predicate variables, directly produces all the potential subsets. Such a system is described in [7], but the corresponding implementation is not as mature as that of Hilog.

In our unoptimized implementation, the subset generation is performed by a predicate `subsetof(R,N)(S)`, which, given a relation `R` and a natural number `N` returns in `S` the *name* of a relation that is a subset of `R`. We assume that the relation `R` exists in the program as a set of ground facts (ie., it is essentially a table). For instance, assume we have defined a predicate `p` with the facts:

```
p(a).
p(b).
p(c).
```

and we have the goal clause:

```
?- subsetof(p,2)(S).
```

Then, the implementation proceeds as follows:

- Three new predicate names `s1`, `s2` and `s3` are introduced and corresponding facts are asserted in the program. more specifically, predicate `s1` consists of the facts `s1(a)` and `s1(b)`, predicate `s2` consists of the facts `s2(a)` and `s2(c)` and predicate `s3` of the facts `s3(b)` and `s3(c)`.
- The `S` variable gets the new predicate names `s1`, `s2` and `s3` as bindings.

In order to achieve this behavior, unique symbol names for the dynamically generated predicates are first produced. Then, a systematic procedure is followed in order to enumerate the elements of each subset, create the appropriate

facts and assert them into the knowledge-base. The procedure is implemented by a `generate_relations` predicate which produces a list of tuples and asserts each tuple as a fact under the newly generated predicate name. The `generate_relations` has been defined in an extensible way that allows the incorporation of specialized pruning strategies and more sophisticated optimizations, some of which will be discussed in the following subsections.

6.2. Transformations and Optimizations

There are various transformations and optimizations that one could use in order to enhance the performance of query evaluation of the simple implementation introduced in the previous subsection. First of all, since our source language is an extension of classical logic programming, one can extend corresponding transformations that have been proposed for first-order logic programs, such as the classical fold/unfold techniques (see for example [12]). Moreover, one can use standard optimization techniques that have been used for speeding-up logic programs (see for example [13]). An extended discussion of such techniques and optimizations is outside the scope of this paper. However, we will focus on two such approaches that have proved especially beneficial for our implementation, namely *removing higher-order parameters* and *memoizing* predicate calls. The former is less well-known and will be explained in more detail; the latter is well-known and directly supported by XSB (in most cases).

In the functional programming domain there have been proposed certain techniques that transform higher-order programs to semantically equivalent first-order ones (such as the *firstification* technique of [14] and the *higher-order removal* method of [15]). Since first-order programs are simpler and more easily implementable, the application of the above techniques can lead to significant performance benefits. However, we should note that not all higher-order programs have been shown to be amenable to this type of transformations, and therefore firstification and higher-order removal can be considered as transformation/optimization techniques and not as general implementation approaches for functional programming. To our knowledge, such techniques have never before been applied to the logic programming domain.

The majority of Hilog programs that we have considered in this paper have a relatively simple structure: they consist of a preference relation definition part (which usually is a first-order program) and a query using `winnow` (which is a simple non-recursive higher-order program). It is therefore not hard to make the query first-order by specializing the parameters of `winnow` for the relation and preference relation under consideration. First-order queries and programs can be executed more efficiently by the XSB system, for reasons that we will explain shortly. The following example demonstrates this transformation.

Example 16. Consider again our first movie preference relation (namely `c1_pref` of Example 5), namely:

```
c1_pref((I1,N1,D1,G,M1,R1),(I2,N2,D2,G,M2,R2)) :-
        movie((I1,N1,D1,G,M1,R1)),
        movie((I2,N2,D2,G,M2,R2)),
        R1 > R2.
```

together with the definition of the `winnow` operator:

```
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(T1), C(T1,T).
```

and the query:

```
?- winnow(c1_pref,movie)(T).
```

We can specialize `winnow` with the `movie` relation and the `c1_pref` preference relation, getting the following first-order predicate:

```
winnow_c1(T) :- movie(T), not bypassed_c1(T).
bypassed_c1(T) :- movie(T1), c1_pref(T1,T).
```

The query can now be stated as:

```
?- winnow_c1(T).
```

Notice that `winnow_c1` is a specialized version of the `winnow` operator that can be used to answer only the given query. □

We have been able to transform all programs in the paper that involve tuple preferences, to equivalent first-order ones. Our transformation follows similar steps to those we used in the above example, and can be easily automated and embedded as a standard preprocessing step regarding most tuple-preference queries. However, when we considered the operators that generalize `winnow`, such as w_C^n, the transformation to first-order proved to be more demanding. This is due to the fact that these operators combine higher-order features together with recursion, a fact that renders their translation non-trivial. We should note here that the difficulty lies in obtaining a first-order program that implements w_C^n for *every* n; if we just wanted to obtain an implementation of w_C^2 or of w_C^3, this would be a straightforward task (the translation would be a simple unfolding of the corresponding definitions, similar to that given in the above example).

In the following we present the full transformation to first-order for the *iterated-winnow* w_C^n operator of Subsection 4.4. The operator wt_C^n can be treated in an analogous way.

Consider the Hilog definition of iterated-winnow, together with all required predicates:

```
w(C,R)(1)(T) :- winnow(C,R)(T).
w(C,R)(N)(T) :- N>1,M is N-1,
                winnow(C,diff(R,gen_union(M)(w(C,R))))(T).

winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(Z), C(Z,T).

gen_union(1)(R)(X) :- R(1)(X).
gen_union(N)(R)(X) :- N>1, M is N-1,
                      union(gen_union(M)(R), R(N))(X).
```

```
union(R,Q)(X) :- R(X).
union(R,Q)(X) :- Q(X).
diff(P,Q)(X) :- P(X), not Q(X).
```

We see now that for every value of N, the definition of w(C,R)(N)(T) uses a different relation as the second parameter of winnow in its body. This gives the idea to parameterize diff with the level N and to also create a simpler union relation that is also parameterized by N, expressing in this way the different relations that show-up in the body of the clause for w(C,R)(N)(T). It is not hard to verify that the following program performs the same task as the above one, using a modified union and diff operator. Roughly speaking, the resulting program is constructed by unfolding the union predicate in the definition of gen_union, then by specializing gen_union with the w(C,R) predicate, and then by specializing diff with the gen_union predicate.

```
w(C,R)(1)(T) :- winnow(C,R)(T).
w(C,R)(N)(T) :- N>1, M is N-1, winnow(C,diff(C,R)(M))(T).

winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(Z), C(Z,T).

gen_union(1)(C,R)(X) :- w(C,R)(1)(X).
gen_union(N)(C,R)(X) :- N>1, w(C,R)(N)(X).
gen_union(N)(C,R)(X) :- N>1, M is N-1, gen_union(M)(C,R)(X).

diff(C,R)(N)(X) :- R(X), not gen_union(N)(C,R)(X).
```

In order to get a first-order program we need to remove the parameters C and R from all predicates (replacing them with specific relations) and also to parameterize winnow and bypassed with a variable N (which will represent each different relation of the form diff(C,R)(N) with which winnow is called in the above program). So, after unfolding the definition of winnow in w, by creating a version of bypassed which is specialized with diff and finally, by putting movie for R and c1_pref for C, we get the following first-order program:

```
w(1,T) :- movie(T), not bypassed(T).
w(N,T) :- N>1, M is N-1, diff(M,T), not bypassed(N,T).

bypassed(T) :- movie(Z), c1_pref(Z,T).
bypassed(N,T) :- N>1, M is N-1, diff(M,Z), c1_pref(Z,T).

gen_union(1,X) :- w(1)(X).
gen_union(N,X) :- N>1, w(N,X).
gen_union(N,X) :- N>1, M is N-1, gen_union(M,X).

diff(N,X) :- movie(X), not gen_union(N,X).
```

Transforming programs and queries to first-order ones has an important benefit: one can use more effectively the advanced memoization mechanisms of

XSB. Memoization [16] is a well-known optimization for declarative programming languages, which is based on the idea that we can store already computed results of calls, which can later be retrieved and used directly when the same calls occur again. In logic programming, memoization is sometimes referred as *tabling*. The XSB system is known for its elaborate and efficient implementation of tabling for first-order logic programs. For higher-order Hilog programs however, XSB's tabling mechanism is not as effective as it is for first-order ones. The reason is that in order to table any Hilog predicate one has to table all Hilog code: in XSB a Hilog program is transformed into a first-order program[4] using a special `apply` predicate and this alters the structure of the initial program making the tabling process less effective. As we are going to see, memoization has offered significant benefits in the case of programs that use alternative preferential operators beyond winnow and in programs that use extrinsic preference relations. These issues will be discussed in detail in Section 7.

6.3. Optimization and Pruning Strategies for Set Preferences

As discussed in Subsection 6.1, Hilog's approach for treating uninstantiated predicate variables in queries, is to instantiate them with predicate names that are defined in the program. By trying all possible instantiations for such variables we have a guarantee that if there exists a predicate that satisfies the query, it will be eventually found by the system. Therefore, it is evident that the performance of query execution in the case of preferences over sets, heavily depends on the number of the predicate names that must be substituted and checked, or equivalently the number of the sets that are generated by the `subsetof` predicate (which was described in Subsection 6.1). This process is inherently slow in the worst case, because there exist applications where all the subsets of a specific size must be tested. However, as remarked in [4], there also exist many applications where optimizations may help to get a performance that is acceptable in practice. Therefore, it would be beneficial to devise techniques that will reduce the number of the subsets generated and at the same time will not compromise the soundness of the implementation.

There are many possible optimizations that one could devise, that would enhance the performance of queries over set preferences. In our Hilog implementation we have experimented with the two main optimization techniques that have been proposed in [4] for speeding-up the subset generation process during query processing, namely *superpreference* and *M-relation*. Intuitively, superpreference removes tuples that will not contribute to the production of any best k-subset, while the M-relation "groups together" tuples that are exchangeable with respect to a given set preference. It should be noted that as observed in [4], these two optimizations can be applied only on preferences that are *additive* (see [4] for details), a class that is quite common in practice. Due to space restrictions, we give below a relatively high-level description of the two

[4]Notice that this transformation is different in philosophy from firstification and higher-order removal, and the resulting program is much more complicated than the initial one.

optimizations and of their implementation in our framework. For more details, the interested reader should consult the detailed exposition in [4].

The key idea behind the superpreference optimization is that in some cases, instead of examining a preference relation between sets, we can examine a (so-called) superpreference relation *over tuples*. More specifically, in some cases there exists a superpreference relation $\succ_+$ between tuples, such that $t_1 \succ_+ t_2$ iff for every $(k-1)$-subset s, $\{t_1\} \cup s$ is preferred to $\{t_2\} \cup s$. Consider again the preference relation C_1 from Example 4 which is defined over 3-element sets. It is not hard to see that this preference relation is closely connected to the following superpreference relation among tuples which holds if the rating of a tuple is higher than the rating of another one:

```
rating_superpref((I1,N1,D1,G1,M1,R1), (I2,N2,D2,G2,M2,R2)) :-
  movie((I1,N1,D1,G1,M1,R1)), movie((I2,N2,D2,G2,M2,R2)), R1 > R2.
```

Given a superpreference relation between tuples, the superpreference optimization applies two pruning conditions, which we outline below using our running `movie` example. First, notice that the movie m_4 has at least 3 movies that are "superpreferred" from it, and can be filtered-out before the start of the subset generation process, because it is certain that it will not contribute to a best set (1st pruning condition). Secondly, notice that any subset that contains the movie m_5 and does not contain the movie m_1 (i.e. it does not contain all movies that are "super-preferred" from it with respect to `rating_superpref`) is not a best subset and can be pruned at the subset generation step (2nd pruning condition). In our implementation we provide an optimized winnow operator, called `winnowsuper`, which is enhanced according to the above two pruning conditions of the superpreference optimization. In order to get the most preferred 3-element movie sets according to the C_1 preference, one can issue the following query:

```
?- winnowsuper(rating_pref,rating_superpref,movie,3)(S).
```

Omitting the details, this call begins by pruning the base relation `movie` according to the 1st pruning condition, using the superpreference `rating_superpref`. Then, it generates all 3-subsets of the pruned `movie` relation, but it keeps only the appropriate sets according to the 2nd pruning condition, again using `rating_superpref`. Finally, it applies a winnow operator over this pruned set of 3-element sets using the preference relation `rating_pref` between sets (which was defined in Example 12) in order to get the most preferred sets.

The other optimization technique that is proposed in [4] is the M-relation optimization. Again, we will outline the basic idea of this optimization using the C_1 preference from Example 4. The movies m_5 and m_6 have the same rating, so the sets $\{m_1, m_2, m_5\}$ and $\{m_1, m_2, m_6\}$ have the same sum of ratings and are equally preferred. So, if we could "group" the tuples m_5 and m_6 into one meta-tuple $m_{5,6}$, we would be able to "group" these two sets into the meta-set $\{m_1, m_2, m_{5,6}\}$, thus reducing the number of candidate sets that are of equal importance, and avoiding unnecessary repetitions of sets that are equally preferred. At the end, the most preferred sets are the sets that correspond to

the most preferred meta-set. The set of meta-tuples (called M-tuples) is constructed from the original tuples, by keeping only the fields that are needed for the computation of the best subsets, leaving out the fields that do not contribute in the computation of the preference relation. For instance, in the C_1 preference of our example, all M-tuples (i.e. the tuples of the M-relation) contain only one field, which is the rating of the specific movie. The transformation of a movie tuple into the corresponding single-field M-tuple can be encoded using this `rating_mrel` predicate:

```
rating_mrel((I,N,D,G,M,Rating), (Rating)) :-
        movie((I,N,D,G,M,Rating)).
```

The k-subsets are generated from the M-tuples and not from the original tuples (because the number of M-tuples is smaller than the number of the original tuples). However, the set generation process here differs from its unoptimized counterpart. Since one M-tuple t corresponds to $n \geq 1$ original tuples, we must allow up to n duplicates of t in the subset generation step. This is needed because if we did not allow duplicates of the M-tuples, the best set of Example 4 $\{m_1, m_5, m_6\}$ (which is an instance of the multiset $\{m_1, m_{5,6}, m_{5,6}\}$) could not be produced. On the other hand, if we allowed any arbitrary number of duplicates of M-tuples, the multiset $\{m_{5,6}, m_{5,6}, m_{5,6}\}$ would also be produced, which does not correspond to a valid solution of the problem. In our implementation we provide an optimized winnow operator, called `winnowmrelation`, which is enhanced with this optimization. In order to get the most preferred 3-element movie sets according to the C_1 preference, we issue the following query:

```
?- winnowmrelation(rating_pref,rating_mrel,movie,3)(S).
```

Omitting the details, this call begins by converting the original tuples into M-tuples using the `rating_mrel` predicate and tags every M-tuple with the number of the corresponding original tuples. Then, it generates all 3-multisubsets of the M-tuples, by allowing up to the correct number of duplicates, and after the generation step, a winnow operator is applied on these multisets, using the preference relation `rating_pref`. Notice though that since this winnow operator operates on multisets of M-tuples, the `rating_pref` predicate should be redefined in order to operate over multisets with M-tuples rather than sets of original tuples. Finally, since the resulting sets are sets of M-tuples, it concludes with a final procedure that transforms the 3-multisets with M-tuples back to 3-sets with original ones, again using the `rating_mrel` predicate.

The above optimizations can be expressed quite concisely and implemented efficiently in higher-order logic programming (and in Hilog in particular). As it has been noted many times throughout the present paper, the important advantage of the higher-order logic programming approach is that the definition of preference relations, the operators for processing preference relations, the queries, and even the optimizations, can be written in the same language. We believe that this provides a simple and unifying framework for dealing with preferences of various kinds.

7. Experiments and Evaluation

In this section we present experimental results we have obtained through our implementation of the ideas proposed in this paper. The purpose of the experimental evaluation is to identify the strengths and the weaknesses of the logic programming approach to preference representation. Such an evaluation can help us identify the potential next steps in further developing our approach. Since (to our knowledge) there do not exist any other available systems or implementations for representing preferences in a logic programming setting, we can not provide a comparative assessment of our approach. For this reason we focus on providing experimental results regarding the performance of our approach based on increasing base-relation sizes. In particular, we have built a test suite to measure the query running time for several preference queries, and we carried out the following four different types of experiments:

1. Queries on tuple preferences.
2. Queries of other preferential operators (besides winnow).
3. Queries on recursively defined preference relations.
4. Queries on set preferences.

All experiments were performed on a Linux Desktop PC (Ubuntu 14.04 LTS) with Intel(R) Core(TM) i7-4790 CPU, 8 GB RAM. The experiments were executed in XSB, version 3.7. All data has been artificially generated.

7.1. Experiments on Tuple Preferences

In the first type of experiments we evaluate the performance of the `winnow` operator over a randomly generated `movie` relation, using the preference relations we have defined in Section 4. We generate a relation that contains n facts of the form `movie(ID,Name,Director,Genre,Runtime,Rating)`, where the first two fields are unique, the `Director` field is randomly selected from a set that contains $n/5$ director ids, the `Genre` field is randomly selected from a set that contains 22 genres, and the remaining fields are random integers, with `Runtime` ranging in $[100, 260]$ and `Rating` ranging in $[5, 95]$. Then, we issue in XSB the following queries:

```
?- winnow(pref,movie)(X), fail.
?- winnow_pref(X), fail.
```

which select all best tuples from `movie` with respect to the `pref` relation. The former goal corresponds to the standard higher-order `winnow` query, while the latter corresponds to the optimized version, as described in Subsection 6.2. In our experiments we use the intrinsic preference relations `c1_pref`, `c2_pref`, `c3_pref`, the conjunction composition `conj(c1_pref,c2_pref)`, the prioritized composition `prioritized(c2_pref,c1_pref)` and the extrinsic preference relation `director_pref`. We evaluate the system for $n = 100$, 500, 1000, 2000, 4000, 8000 and 10000 facts. The results for this experiment are shown in Table 1.

In the first column we illustrate the size of the `movie` relation, while in the remaining columns we illustrate the `winnow` query execution times in seconds.

	winnow execution time (sec)					
	c1_pref		c2_pref		c3_pref	
#facts	naive	optimized	naive	optimized	naive	optimized
100	0.07	0.08	0.04	0.05	0.02	0.05
500	0.15	0.13	0.12	0.10	0.06	0.04
1,000	0.32	0.30	0.37	0.31	0.08	0.07
2,000	1.00	0.82	1.66	1.36	0.46	0.38
4,000	3.21	2.74	10.51	8.48	2.72	2.20
8,000	13.18	10.94	58.80	45.70	18.86	15.35
10,000	22.51	21.02	96.44	78.09	49.23	39.63
	conj		prioritized		director_pref	
	naive	optimized	naive	optimized	naive	optimized
100	0.05	0.05	0.05	0.05	0.16	0.07
500	0.21	0.16	0.15	0.10	4.72	0.58
1,000	0.91	0.68	0.40	0.31	12.75	1.36
2,000	5.39	4.10	1.57	1.27	99.89	10.84
4,000	30.48	24.27	6.96	5.50	547.88	61.13
8,000	154.49	123.18	29.22	23.37	923.22	94.40
10,000	254.49	204.69	44.23	35.68	1,805.95	165.85

Table 1: `winnow` execution times (in sec) for movie tuple preferences.

	winnow execution time (sec) / # results						
						director_pref	
#facts	c1_pref	c2_pref	c3_pref	conj	prioritized	naive	optimized
100	0.004	0.001	0.004	0.001	0.002	0.015	0.006
500	0.006	0.001	0.002	0.001	0.005	0.236	0.029
1,000	0.011	0.004	0.002	0.002	0.015	1.063	0.113
2,000	0.025	0.010	0.005	0.011	0.065	4.162	0.452
4,000	0.056	0.035	0.015	0.046	0.260	11.414	1.273
8,000	0.140	0.097	0.053	0.164	1.052	65.944	6.743
10,000	0.198	0.124	0.096	0.224	1.480	120.397	11.057

Table 2: `winnow` execution times per result (in sec) for movie tuple preferences.

For each preference, we show the query execution time for both the naive and the optimized version of the query. Regarding the first five intrinsic preferences, both versions of the `winnow` operator have quite similar execution times, with the optimized version being a little faster than the unoptimized one, probably due to the runtime overhead that is introduced by XSB in order to execute Hilog code. However, a more extreme time difference occurs in the execution times for the extrinsic preference `director_pref`. This behavior is expected because in order to compare two movie tuples in this case, one has to compute the exact number of movies that each of the two directors has directed. The transformation to first-order and the memoization of this preference relation results to a significant speedup, and as a result the query execution times differ by an order of magnitude.

The query execution time obviously increases as the size of the relation increases, but for similar base-relation sizes, the execution time clearly depends on the preference relation being evaluated. Since each of these queries return result sets of varying size, in Table 2 we divide the query execution time of the `winnow` query for each preference relation with the number of the returned results. This measurement shows how time consuming is to obtain a single result and this is clearly analogous to the difficulty of each preference computation. For the extrinsic preference `director_pref` we show the different measurements for each of the optimized and unoptimized case, while in the intrinsic ones, we display a mean value because the execution times are similar. For the simple intrinsic preferences `c1_pref`, `c2_pref`, `c3_pref` the execution times per result are quite similar and relatively low. The use of compositions of preferences though, lead to slower execution times per result, with the prioritized composition being the slowest of the two. This is due to the fact that prioritized composition of two preference relations requires a more difficult computation than a simple conjunction. Finally, as we saw in the previous paragraph, the most difficult preference computation is the extrinsic preference relation `director_pref`. This fact is also highlighted here.

7.2. Experiments on Preferential Operators

In the second class of experiments we evaluate the performance of the operators w_C^n and wt_C^n introduced in Subsection 4.4. For each of the two operators we have also implemented an optimized version, as described in Subsection 6.2. The datasets we use are generated randomly with the same manner as in the previous class of experiments. For the evaluation we use the queries:

```
?- w(c1_pref,movie)(n)(X), fail.
?- wt(c1_pref,movie)(n)(X), fail.
?- wo(n,X), fail.
?- wto(n,X), fail.
```

where `w` and `wt` correspond to w_C^n and wt_C^n respectively, and `wo` and `wto` are their optimized versions. We have used only the `c1_pref` preference over the `movie` relation, and `n` is the desired level n at which each operator is evaluated. The results of this experiment are shown in Table 3. The empty entries in the table

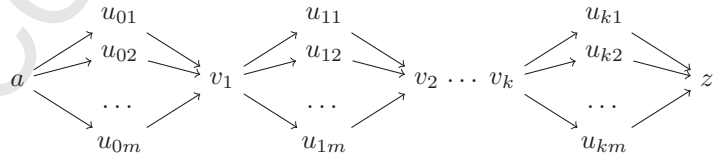
#facts	level	w_C^n time		wt_C^n time	
		naive	optimized	naive	optimized
100	$n=1$	0.08	0.06	0.03	0.03
	$n=2$	0.08	0.03	0.09	0.03
	$n=3$	1.69	0.04	1.74	0.04
	$n=4$	42.90	0.04	44.54	0.04
500	$n=1$	0.13	0.16	0.05	0.08
	$n=2$	2.62	0.24	2.71	0.13
	$n=3$	219.96	0.18	217.63	0.18
	$n=4$	–	0.20	–	0.23
1,000	$n=1$	0.33	0.36	0.16	0.22
	$n=2$	12.50	0.39	12.60	0.41
	$n=3$	1,367.27	0.54	1,373.70	0.54
	$n=4$	–	0.64	–	0.63
2,000	$n=1$	0.95	1.02	0.54	0.77
	$n=2$	73.62	1.52	74.13	1.50
	$n=3$	–	1.93	–	1.78
	$n=4$	–	2.32	–	2.29

Table 3: Query execution times for alternative preferential operators.

correspond to query execution times over 2 hours (7200 seconds). Obviously, the unoptimized implementations of the w_C^n and wt_C^n operators are completely impractical. Using the optimized versions of these operators makes the execution times much more reasonable. Observe that while in the unoptimized versions the execution times increase dramatically as n increases, the corresponding execution times for the optimized versions exhibit a graceful degradation.

7.3. Experiments on Path Preferences

In the third class of experiments we evaluate the performance of our approach for the path preferences as defined in Subsection 4.2. For each experiment we generate a directed acyclic graph of the following form:



We generate several graphs with varying k and m parameters, and then we measure the execution times of the following queries:

```
?- naive_shortest(a,z,C),!.
?- enhanced_shortest(a,z,C),!.
```

# edges	graph parameters		winnow execution time (sec)			
			unoptimized		optimized	
			naive	enhanced	naive	enhanced
10	$k = 1$	$m = 5$	0.06	0.03	0.07	0.04
20	$k = 1$	$m = 10$	0.05	0.03	0.03	0.03
40	$k = 1$	$m = 20$	0.05	0.05	0.04	0.04
20	$k = 2$	$m = 5$	0.04	0.07	0.04	0.04
40	$k = 2$	$m = 10$	0.27	1.84	0.06	0.03
30	$k = 3$	$m = 5$	0.14	22.54	0.03	0.03
80	$k = 2$	$m = 20$	1.95	32.84	0.06	0.04
40	$k = 4$	$m = 5$	10.47	3,242.15	0.08	0.05
60	$k = 3$	$m = 10$	13.22	2,264.04	0.12	0.03
50	$k = 5$	$m = 5$	179.74	–	0.45	0.04
80	$k = 4$	$m = 10$	–	–	1.08	0.03
100	$k = 5$	$m = 10$	–	–	1.83	0.02
200	$k = 10$	$m = 10$	–	–	20.64	0.07
400	$k = 10$	$m = 20$	–	–	98.86	0.05

Table 4: `winnow` execution times (in sec) for path preferences.

```
?- naive_shortest_opt(a,z,C),!.
?- enhanced_shortest_opt(a,z,C),!.
```

As in the previous examples, the first two calls correspond exactly to the programs shown in Subsection 4.2, while the remaining two calls correspond to the optimized versions of these predicates, which are derived using the techniques of Subsection 6.2. The results of this experiment are displayed in Table 4. As in the previous experiment, the empty entries correspond to query execution times over 2 hours (7200 seconds).

The results in Table 4 lead to the following two observations. First, as in the previous experiments, we notice that the non-optimized versions of the path predicates are becoming impractical for larger inputs, and that this situation can be remedied by transforming the corresponding queries to first-order and using memoization. Second, we notice that in the non-optimized case, the predicate that selects the best path by examining all possible paths, is faster than the predicate that uses `winnow` in its recursive invocation. In the optimized versions of these predicates though, we observe that the naive method becomes much slower; moreover, we have observed that the naive method uses a very large amount memory (this fact is not depicted in Table 4). As a result, we see that the optimization techniques that we have developed in Subsection 6.2 combined with the flexibility of embedding the `winnow` operator inside the base relations, as we have shown in Subsection 4.2, can result to a much better performance.

		winnow execution time (sec)			
#facts	set size	winnow	winnowopt	winnowsuper	winnowmrel
10	$k = 3$	0.10	0.10	0.08	0.06
25	$k = 3$	0.93	0.28	0.15	0.21
50	$k = 3$	83.27	11.41	1.16	5.14
75	$k = 3$	217.62	41.47	3.59	7.90
100	$k = 3$	2,089.67	350.43	33.02	34.53
25	$k = 2$	0.12	0.11	0.15	0.10
25	$k = 3$	0.93	0.28	0.15	0.21
25	$k = 4$	15.13	2.35	0.19	1.22
25	$k = 5$	179.18	32.51	0.22	9.55
25	$k = 6$	1,434.29	274.41	0.23	54.15

Table 5: `winnow` execution times (in sec) for set preferences.

7.4. Experiments on Set Preferences

In the final class of experiments we evaluate the performance of our approach for the `rating_pref` set preference of Section 5. For each experiment we first generate n facts of the `movie` relation, as described in the first two experiments. For each run of the experiments we measure the execution times of the following four queries:

```
?- winnow(rating_pref,subsetof(movie,k)(S), fail.
?- winnow(opt_rating_pref,subsetof(movie,k)(S), fail.
?- winnowsuper(rating_pref,rating_superpref,movie,k)(S), fail.
?- winnowmrelation(rating_pref,rating_mrel,movie,k)(S), fail.
```

The first two calls select the best k-sets of movies according to `rating_pref` in a naive way, the third one is optimized using the "superpreference" technique, and the fourth one is optimized using the M-relation approach. The difference between the first two calls, is that in the second one we use standard Prolog built-ins in order to construct an efficient version of the `rating_sum/2` predicate (which counts the sum of the ratings in a set). We evaluate the queries for fixed subset size $k = 3$ and varying base-relation sizes $n = 10, 25, 50, 75, 100$. Then, we perform again the evaluation for fixed base-relation size $n = 25$ and varying subset size $k = 2, 3, 4, 5, 6$. For each run of the experiments we also measure the total number of sets that are generated for each approach. The query execution times are displayed in Table 5 and the number of the generated sets for each query are displayed in Table 6.

As we can see from the results of Tables 5 and 6, the superpreference and M-relation optimizations are both performing very well in our case, due to the effective pruning of the total number of subsets that need to be generated. Considering the naive evaluation, all $\binom{n}{k}$ subsets have to be generated. However, since the second version of the naive evaluation (`winnowopt`) uses a more efficient implementation of the preference relation predicate we have a significant

		# sets			
#facts	set size	winnow	winnowopt	winnowsuper	winnowmrel
10	$k = 3$	120	120	1	92
25	$k = 3$	2,300	2,300	1	1,603
50	$k = 3$	19,600	19,600	1	8,808
75	$k = 3$	67,525	67,525	1	18,054
100	$k = 3$	161,700	161,700	3	37,558
25	$k = 2$	300	300	1	234
25	$k = 3$	2,300	2,300	1	1,603
25	$k = 4$	12,650	12,650	1	7,948
25	$k = 5$	53,130	53,130	1	30,384
25	$k = 6$	177,100	177,100	1	93,139

Table 6: Number of generated sets for set preferences.

drop in the execution times, that is roughly one order of magnitude. Despite this decrease, the other two optimizations that focus on the reduction of the number of the generated sets are performing even better. Considering the superpreference optimization, notice that in our case only the movies that have the highest ratings contribute to the best sets. Therefore, at the first step of the superpreference optimization all movie tuples from the base relation are removed, except for the ones whose ratings belong to the k highest ones. This results to a radical pruning of the set of the generated subsets, which results to a significant decrease of the query execution time. Considering the M-relation optimization, notice that we do not have as much pruning here as in the superpreference optimization. However, the M-tuples contain only the required information for the preference comparison, namely the rating of the movie, and since these tuples are more compact than the original movie tuples, the generation of all subsets is much faster, and this results to a better performance.

As a general comment we could say that the efficient handling of set preferences possibly represents the most challenging area for future work, both at the theoretical and also at the experimental level. First of all, it appears that there is a lot of space for defining new and powerful optimization techniques for set preferences that would further alleviate the cost of the subset generation procedure. Moreover, since the handling of sets can be intractable in the general case, many of these techniques would have to be heuristics and would need to be experimentally evaluated in practice in order to assess their potential for real life applications.

8. Related Work

Preferences play an important role in knowledge representation and have many applications in diverse domains. In this section, we discuss related work regarding preference representation formalisms that have been proposed in the

areas of databases and logic programming. As a general comment, we could say that our approach is not very directly related to most of the existing ones, since, to our knowledge, it is the first time that the use of a higher-order language is proposed as a framework for expressing preferences. However, in certain cases there exist some underlying connections to certain of the proposed techniques, which we highlight below. Moreover, it is interesting to observe that most of the techniques we review below, introduce new language constructs and semantics, an indication that a stronger formalism than that of traditional relational calculus or classical logic programming is needed in order to express preferences.

As already mentioned, formalisms for representing preferences can be divided [1] into the *qualitative* and the *quantitative* ones. We start with a discussion about quantitative systems (which are the most distant from our approach) and continue with a discussion about qualitative approaches in databases and logic programming.

8.1. Formalisms for Quantitative Preferences

In the quantitative approach, preferences are expressed by assigning numerical values on tuples, such that one tuple is preferred over another if its preference score is higher. In the database domain, relevant quantitative approaches are developed in [17], where the score can be obtained by a function which takes a tuple and returns a numerical value, and in [18], where the score is obtained by specifying appropriate selection conditions. Quantitative approaches in logic programming can be found in certain many-valued or probabilistic extensions of traditional logic programming, such as for example [19, 20, 21, 22, 23]. In probabilistic extensions the programmer is usually required to rank rules (or facts) with a certainty factor or to use a special form of implication in rules that has attached a numerical attenuation factor. These extensions of logic programming are usually not designed in order to express preferences but have a flavor of prioritized logic programming, since, for example, facts that have different numerical factors can be considered to be of different preferences. Another relevant work in this line of research is [24] which proposes a technique for representing preferences using a logic with an infinite set of truth values; in [24] the numerical values are not explicit but are created using appropriate preferential operators.

It has been argued that the quantitative approaches are less general than the qualitative ones: there exists a preference relation that can not be expressed using a scoring function (see Example 1.2 in [2][page 428] and the discussion in Subsection 7.1.2 of the same paper). We believe that many quantitative systems are actually quite sophisticated and they may prove to have interesting applications. However, when it comes to expressing preferences at a high declarative level, we feel that the qualitative approach has an advantage. As remarked in [25], "*humans are rarely willing to express their preferences directly in terms of a value function. [...] Instead of rating complete alternatives immediately, it is normally much easier and arguably more natural to provide information about preferences in separate pieces, preferably in a qualitative way*".

8.2. Formalisms for Qualitative Preferences

In this subsection we discuss the most well-known qualitative approaches for expressing preferences in databases and logic programming. We start with the work of [2, 4], on which the current paper is based, and continue with other well-known approaches for the same problem.

8.2.1. The Preference-Formulas Approach for Databases

The framework of [2, 4] is one of the most mature ones in the corresponding database literature. Since (to our knowledge) there does not exist any widely available implementation of the ideas in [2, 4], it is not possible to directly compare this approach with our own in terms of efficiency. We can however present the novelties of our approach with respect to the work of [2, 4], which, as we believe, make it worthwhile for further development.

- Our approach uses a uniform language to represent preferences over tuples, preferences over sets, queries, and new preferential operators. We believe that uniformity is an important characteristic for preference representation formalisms.

- It can express both intrinsic and extrinsic preference relations. Also, it can express recursively defined preference relations which appear to be quite useful (see for example the path predicates at the end of Subsection 4.2).

- It can be easily used in order to experiment with various ideas regarding preferences, because it can run, almost directly, on existing higher-order logic programming systems. This is especially true for tuple-preferences (the case of preferences over sets requires certain interventions for producing the candidate sets). To our knowledge, there does not exist any other widely available implementation for representing preferences in a logical way. Moreover, our approach offers a new, interesting application area for higher-order logic programming.

- Our approach establishes the close ties between higher-order logic and preferences, and suggests that we can have a purely deductive database framework for representing and experimenting with preferences. This opens up the possibility of devising specialized transformations, optimizations and proof procedures for preference applications. Promising such optimizations are given in Subsection 6.2 and their efficiency is verified in Section 7.

It would however be unfair not to mention the advantages of the work in [2, 4]. First of all, it is a simple framework, and possibly the most appealing one in the database literature regarding preference representation. We believe that it is not very hard to implement the ideas in [2, 4] and to amalgamate them with existing relational database systems. Since SQL is undoubtedly more useful than logic programming, an implementation of the system in [2, 4] will probably be more efficient and can have a broader impact and real-world applications than our logic programming approach.

In general, we feel that the relationship between the work of [2, 4] and our own, is similar to the relationship between SQL and deductive databases: both approaches are needed and each one has its own advantages. For this reason we believe that it is important for both research directions to further develop, and this will benefit the area of preference representation as a whole.

8.2.2. Other Approaches to Qualitative Preferences in Databases

One of the earliest works in qualitative preference queries in databases is [26] where the authors propose an extension of relational calculus in which preferences for tuples satisfying given logical conditions can be expressed. For example, one could say: pick the tuples of R satisfying $Q \wedge P_1 \wedge P_2$; if the result is empty, pick the tuples of R satisfying $Q \wedge P_1 \wedge \neg P_2$; if the result is again empty, pick the tuples of R satisfying $Q \wedge \neg P_1 \wedge P_2$. As discussed in [2], this approach can be simulated by the techniques in [2], and therefore also by our framework.

Another influential work is that of Kießling [11]. Contrary to the logical approach advocated by Chomicki, Kießling takes an algebraic one by using a language which offers preference constructors such as `POS` and `NEG`. For example, a `POS(genre,{comedy})` preference states that a comedy movie is preferred, while a `NEG(genre,{drama,sci-fi})` preference states that a movie is not preferred if it is either a `drama` or a science fiction one. These preference constructors can be further combined in order to express more complex preferences. Several preference combinators are supported, such as pareto, lexicographic, and so on. In addition, two versions of this approach (Preference XPATH [27] and Preference SQL [28]) are provided. Kießling's framework has some common characteristics with that of [2] (see [2][Section 10.1] for a detailed comparison) and many arguments we have used throughout the paper in order to compare our work with that of [2], can also be used in order to compare it against the work of [11]. For example, the work in [11] does not allow having arbitrary constraints in preference formulas and it allows only restricted use of extrinsic preference relations and of the transitive closure operation.

8.2.3. Qualitative Preferences in Logic Programming

A number of different approaches have been proposed in the logic programming domain with the purpose of supporting qualitative preferences. These approaches can be categorized in two main streams:

- those that use preferences in order to select the best solutions to a given problem that has been expressed as a logic program.
- those that use preferences in order to resolve conflicts that appear in non-monotonic extensions of logic programming (such as multiple minimal models).

Closer to the approach presented in this paper are these that fall in the first category, such as [29, 10, 30, 31, 32, 33]. In order to support preferences, it is common to use syntactically-extended logic programs. For example, the approach in [29, 10, 30, 31] studies logic programs that consist of two basic parts. The first part is a set of first-order definite logic clauses while the second

part is a set of optimization definitions expressed as constraints on the atoms of the first part. The proposed semantics selects a preference model of the program among the possible models, such that it optimizes the second part of the program. The formalism presented in [29] does not support negation-as-failure in programs. A later work reported in [32] identifies this lack of negation in [29] and proposes a different technique in order to support logic programs with negation-as-failure. The proposed approach in [32] is to transform a logic program with preferences into an equivalent logic program with negation. The authors of [32] suggest that the well-founded model of the transformed program is the intended model of the program with preferences and they prove that this model coincides with that of [29] (for the class of programs supported by [29]). The program transformation encodes a behavior that is similar to that of the `winnow` operator. In [33], which is also closely related to [29], the proposed semantics uses two meta-operators over the minimum model of the non-optimization part of the program in order to remove the atoms that are not the most desired ones with respect to the preference definition. These operators, again, resemble to the `winnow` operator that we are using, but in this case the `winnow` operator is not hard-coded in the program but in the evaluation of the preference model. As a result, in the approaches [29, 32, 33] the preference relations and the `winnow` operator can not appear as building blocks of queries (not can one define alternative operators beyond `winnow`). The work reported in [29, 10, 30, 31, 32, 33] has been demonstrated to have applications in resolving ambiguity in programming language and natural language grammars, in scheduling and optimization, as-well-as in database querying. The main difference between this line of research and our own, is that the former uses a specialized formalism that requires the development of novel model-theoretic techniques in order to express its semantics and also of specialized techniques in order to implement it. On the other hand, our approach does not use any specialized machinery outside the realm of higher-order logic programming, and can be implemented using standard higher-order logic programming languages. However, as an overall comment, the work reported in [29, 10, 30, 31, 32, 33] is based on quite interesting concepts, and the idea of the "optimal subproblem property" discussed in [10] has motivated our optimized path program given in Example 10.

The second category of approaches is applicable to logic programming languages that are extended with features such as disjunctions in the head, default and explicit negation, and so on. In short, non-monotonicity plays a vital role in the approaches of this category. Some research works that fall into this category are reported in [34, 35, 36, 37] (see also the excellent review given in the related-work section of [34]). The key idea behind all these approaches can be described as follows. A logic program that is extended with non-monotonic characteristics, usually has many minimal models. In order to choose the most appropriate models, we add to the program preference information. Usually this preference information is either given as an ordering of atoms, literals, or even rules of the program. We concentrate on [34] (and similar arguments can be given for the rest of the cited works). In [34] a priority relation, which is reflexive and

transitive, is given over the set of literals of the program. Then, the *preferred answer-set* semantics of the program is defined, an approach that generalizes the classical stable model semantics of Gelfond and Lifschitz. There are many important differences between our work and that of [34]. First, our work can be applied both to positive programs as-well-as to programs that use negation. On the other hand, the work of [34] is only meaningful when non-monotonicity is present in the program (as noted in [34], if a program is positive then it always has the same model independently of the preference relation given). Therefore it is not obvious how the technique of [34] (and all other related approaches) can be applied to express preferences over simple database relations. The task of defining preferences over sets of tuples appears to be even more difficult.

Another important difference of the proposed approach with respect to many other existing approaches in the logic programming domain, is that it is not very hard to implement and it can be used to directly run some interesting and non-trivial applications.

9. Conclusions and Future Work

In this paper we have proposed the use of higher-order logic programming for expressing and manipulating preferences. The proposed framework can express both intrinsic and extrinsic preference relations, it can be used to define a variety of interesting alternative operators beyond `winnow` and it can also represent set preferences in a natural way. All the programs and techniques we have presented have been implemented and run in Hilog. There exist however certain interesting open questions that if answered would help firmly establish the very close connections between higher-order logic programming and preferences:

- The present paper suggests that a fragment of higher-order logic programming can be used as a purely logical framework for expressing preferences. This fragment is essentially a higher-order extension of Datalog that supports negation and tuples in a restricted way. We have used natural numbers only in order to define specialized operators over preference relations (see Subsection 4.4) and aggregate operators in the case of extrinsic and set-based preference relations (see Example 12). We believe it would be very interesting to study the properties of this Higher-Order Datalog with tuples and negation, a language that generalizes classical first-order Datalog. For example, it would be interesting to investigate bottom-up proof procedures, optimizations (such as a higher-order extension of magic-sets [38]), alternative semantics of negation, and so on.

- For implementing the concepts presented in the paper, we have used the XSB system which provides a mature and stable implementation of Hilog. Despite the fact that we faced no problems in implementing preferences over tuples, the implementation of queries over set preferences was not straightforward. The reason for the difficulties we faced was the way that Hilog treats uninstantiated predicate variables: it searches the program to find appropriate predicates that can be substituted; if no predicates are

found, the query fails. This state of affairs is due to a well-known distinction between *extensional* and *intensional* semantics for higher-order logic programming (see for example the corresponding discussion in the first pages of [7]). We believe that an extensional higher-order logic programming language would offer advantages in the implementation of set preferences because the interpreter of the language would automatically produce all the possible sets that should be substituted for the uninstantiated predicate variables. We have implemented an extensional language called *Hopes*[5] and we are currently working on extensions of the implementation that would allow preferences to be expressed more effectively than in Hilog. In particular, we are working on producing a stable and efficient version of Hopes that will support constructive negation [8], a concept that is very useful in the processing of set preferences.

Overall, we believe that the representation of preferences in a higher-order setting is an exciting new application area for higher-order logic programming and also a chance for developing new and very expressive preference representation formalisms.

References

[1] K. Stefanidis, G. Koutrika, E. Pitoura, A survey on representation, composition and application of preferences in database systems, ACM Transactions on Database Systems 36 (3) (2011) 19.

[2] J. Chomicki, Preference formulas in relational queries, ACM Transactions on Database Systems 28 (4) (2003) 427–466.

[3] W. Chen, M. Kifer, D. S. Warren, HILOG: A Foundation for Higher-Order Logic Programming, Journal of Logic Programming 15 (3) (1993) 187–230.

[4] X. Zhang, J. Chomicki, Preference queries over sets, in: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, 1019–1030, 2011.

[5] D. Miller, G. Nadathur, Programming with Higher-Order Logic, Cambridge University Press, ISBN 978-0-521-87940-8, 2012.

[6] W. W. Wadge, Higher-Order Horn Logic Programming, in: V. A. Saraswat, K. Ueda (Eds.), Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991, MIT Press, ISBN 0-262-69147-7, 289–303, 1991.

[7] A. Charalambidis, K. Handjopoulos, P. Rondogiannis, W. W. Wadge, Extensional Higher-Order Logic Programming, ACM Transactions on Computational Logic 14 (3) (2013) 21.

[5] The current implementation of Hopes can be retrieved from `http://www.github.com/acharal/hopes`.

[8] A. Charalambidis, P. Rondogiannis, Constructive Negation in Extensional Higher-Order Logic Programming, in: Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014, 12–21, 2014.

[9] A. Charalambidis, P. Rondogiannis, A. Troumpoukis, Higher-order logic programming: an expressive language for representing qualitative preferences, in: J. Cheney, G. Vidal (Eds.), Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016, ACM, ISBN 978-1-4503-4148-6, 24–37, 2016.

[10] K. Govindarajan, B. Jayaraman, S. Mantha, Optimization and Relaxation in Constraint Logic Languages, in: H. Boehm, G. L. S. Jr. (Eds.), Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, ACM Press, ISBN 0-89791-769-3, 91–103, URL http://doi.acm.org/10.1145/237721.237735, 1996.

[11] W. Kießling, Foundations of Preferences in Database Systems, in: VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China, 311–322, 2002.

[12] H. Tamaki, T. Sato, Unfold/Fold Transformation of Logic Programs, in: S. Tärnlund (Ed.), Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984, Uppsala University, 127–138, 1984.

[13] R. A. O'Keefe, The craft of Prolog, vol. 283, MIT press Cambridge, 1990.

[14] G. C. Nelan Jr, Firstification, Ph.D. thesis, Arizona State University, 1992.

[15] W. Chin, J. Darlington, A Higher-Order Removal Method, Lisp and Symbolic Computation 9 (4) (1996) 287–322.

[16] A. Field, P. Harrison, Functional Programming, Addison-Wesley, 1988.

[17] R. Agrawal, E. L. Wimmers, A Framework for Expressing and Combining Preferences, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA., 297–306, 2000.

[18] G. Koutrika, Y. E. Ioannidis, Personalization of Queries in Database Systems, in: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA, 597–608, 2004.

[19] M. H. van Emden, Quantitative Deduction and its Fixpoint Theory, Journal of Logic Programming 3 (1) (1986) 37–53.

[20] E. Dantsin, Probabilistic Logic Programs and their Semantics, in: Logic Programming, First Russian Conference on Logic Programming, Irkutsk, Russia, September 14-18, 1990 - Second Russian Conference on Logic Programming, St. Petersburg, Russia, September 11-16, 1991, Proceedings, 152–164, 1991.

[21] S. Muggleton, Stochastic Logic Programs, in: L. D. Raedt (Ed.), Advances in Inductive Logic Programming, IOS Press, 254–264, 1996.

[22] K. Kersting, L. D. Raedt, Bayesian Logic Programs, in: Inductive Logic Programming, 10th International Conference, ILP 2000, Work-in-progress reports, London, UK, July 2000, Proceedings, 2000.

[23] L. D. Raedt, A. Kimmig, H. Toivonen, ProbLog: A Probabilistic Prolog and Its Application in Link Discovery, in: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, 2462–2467, 2007.

[24] P. Rondogiannis, A. Troumpoukis, Expressing preferences in logic programming using an infinite-valued logic, in: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015, 208–219, 2015.

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

[26] M. Lacroix, P. Lavency, Preferences; Putting More Knowledge into Queries, in: VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England, 217–225, 1987.

[27] W. Kießling, B. Hafenrichter, S. Fischer, S. Holland, Preference XPATH: A Query Language for E-Commerce, in: Information Age Economy: 5. Internationale Tagung Wirtschaftsinformatik 2001, Augsburg, Germany, 32, 2001.

[28] W. Kießling, G. Köstler, Preference SQL - Design, Implementation, Experiences, in: VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China, 990–1001, 2002.

[29] K. Govindarajan, B. Jayaraman, S. Mantha, Preference Logic Programming, in: Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995, 731–745, 1995.

[30] B. Jayaraman, K. Govindarajan, S. Mantha, Preference Logic Grammars, Comput. Lang. 24 (3) (1998) 179–196.

[31] K. Govindarajan, B. Jayaraman, S. Mantha, Preference Queries in Deductive Databases, New Generation Comput. 19 (1) (2000) 57–86.

[32] B. Cui, T. Swift, Preference Logic Grammars: Fixed point semantics and application to data standardization, Artificial Intelligence 138 (1-2) (2002) 117–147.

[33] H. Guo, B. Jayaraman, Logic programming with solution preferences, Journal of Logic and Algebraic Programming 78 (1) (2008) 1–21.

[34] C. Sakama, K. Inoue, Prioritized logic programming and its application to commonsense reasoning, Artificial Intelligence 123 (1-2) (2000) 185–222.

[35] G. Brewka, Well-Founded Semantics for Extended Logic Programs with Dynamic Preferences, Journal of Artificial Intelligence Research (JAIR) 4 (1996) 19–36.

[36] G. Brewka, T. Eiter, Preferred Answer Sets for Extended Logic Programs, Artificial Intelligence 109 (1-2) (1999) 297–356.

[37] Y. Zhang, N. Y. Foo, Answer Sets for Prioritized Logic Programs, in: Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997, 69–83, 1997.

[38] F. Bancilhon, D. Maier, Y. Sagiv, J. D. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, in: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA, 1–15, 1986.