

Practical Preference Relations for Large Data Sets

Kenneth Ross
Columbia University
kar@cs.columbia.edu

Peter J. Stuckey
NICTA Victoria Laboratory
University of Melbourne
pjs@cs.mu.oz.au

Amélie Marian
Rutgers University
amelie@cs.rutgers.edu

Columbia University Technical Report CUCS-028-06

Abstract

User-defined preferences allow personalized ranking of query results. A user provides a declarative specification of his/her preferences, and the system is expected to use that specification to give more prominence to preferred answers. We study constraint formalisms for expressing user preferences as base facts in a partial order. We consider a language that allows comparison and a limited form of arithmetic, and show that the transitive closure computation required to complete the partial order terminates. We consider various ways of composing partial orders from smaller pieces, and provide results on the size of the resulting transitive closures. We introduce the notion of “covering composition,” which solves some semantic problems apparent in previous notions of composition. Finally, we show how preference queries within our language can be supported by suitable index structures for efficient evaluation over large data sets. Our results provide guidance about when complex preferences can be efficiently evaluated, and when they cannot.

1 Introduction

A variety of applications demand functionality that allows users to specify which among a large set of potential answers to a query is most relevant to them. Based on this specification, the most relevant answers

are given more prominence; for example, they may be displayed first.

Current search engines such as Google provide a scalable implementation of ranking functionality. However, the ranking is done according to a single ranking function (e.g., “pagerank” [14]) that is not adjustable by users. Ideally, different users should be able to specify different relevance measures for the same data.

Past work on preferences has followed one of two approaches, termed the *quantitative* and *qualitative* approaches [4]. According to the quantitative approach [1], one starts by defining a measure on the underlying data set. For example, a simple measure for an automobile sales application might be price, with lower prices having a higher rank. It is then possible to write queries that order the results according to this measure. A common approach is to define a scoring measure that assigns weights based on the importance to the user of each attribute in the data set [8]. In some cases, one can limit the number of answers using a “top- k ” query primitive on the underlying data values or ranks, and achieve savings in query processing cost [7, 6]. Most work on top- k query use a quantitative approach by aggregating individual measures using a monotonic pre-defined scoring function. Quantitative approaches based on scoring functions provide an easy mechanism to produce a complete ordering of the underlying data set. However, quantitative preferences lack expressive power [4].

More general approaches allow a vector of measures, such as the pair (price, safety-rating) for automobiles. Queries then solicit the “skyline” of the resulting combined measure, returning the Pareto-optimal records (i.e., those not dominated by another record) among those satisfying the query [2]. A complete ordering of the underlying data set may not be possible as some records may not be comparable. For instance, it is not possible to order a very safe but expensive car and a cheap but less safe car. While more expressive than quantitative preference approaches, approaches based on (vectors of) measures are not sufficiently expressive

for some applications [4]:

Example 1.1: Suppose that for sports cars, a red car is preferred to a blue car, while for economy cars, a blue car is preferred to a red car. This function cannot be expressed as a monotonic composition of rankings on car-type and color, because red is sometimes preferred to blue, and sometimes blue is preferred to red. A single measure on (car-type,color) pairs cannot capture the requirement that sports cars and economy cars are incomparable with each other. \square

Example 1.2: Suppose that the user cares about price, but does not care about small differences in price. For example, the user might wish to state “For any given class of car, car A is preferred to car B if the price of A is less than 80% of the price of B .” Cars that differ by 20% or less in price are incomparable.¹ Again, this is not specifiable using a single measure on price. Among other things, this kind of rule prevents a car with a price of \$1999 “obscuring from view” another car with a price of \$2000, when only Pareto-optimal records are displayed. \square

According to the qualitative approach, one defines a binary preference relation between entities [12]. We take the qualitative approach in this paper, following Kießling and Chomicki in using strict partial orders to represent the preference relation. One writes $x \succ y$ to describe a preference for x over y . In order to return the Pareto-optimal set of answers, one can use the definition of the partial order to test whether a dominating record appears in the database [12, 13, 4].

Since partial orders are transitive, a user-specified set of preference facts needs an application of transitive closure to generate the complete partial order. In general, the transitive closure cannot be avoided, because a preference for r_1 over r_2 may be a consequence of a chain $r_1 \succ s_1 \succ s_2 \succ \dots \succ s_n \succ r_2$, with none of the s_i actually being present in the database. (In special cases, such as when the preference is specified by a numeric total order, an explicit transitive closure step is not required.)

We focus on what Chomicki calls “intrinsic” preferences, i.e., preference relations that can be specified based solely on values in the database records being compared [4]. Some seemingly non-intrinsic preference relations can be represented intrinsically by creating a view in the database that adds new columns to the records being compared [4]. For efficiency, these views could be materialized and indexed. Sometimes the views can be computed efficiently on the fly, so that additional columns are available to the user to specify preferences.

¹Because we are building a strict partial order, one cannot say something like “cars that differ by 20% or less in price are equivalent.” However, we will achieve a similar effect when combining partial orders in Section 3.1.

Example 1.3: Consider a travel agency website that allows users to list which of the airlines’ frequent flier programs they belong to. On any flight query, the system could add an extra column to the output of each flight leg, indicating whether or not the airline belongs to the user’s frequent-flier list. This is easily performed using a join (or outer join) on the fly, since the list of airlines is likely to be small. \square

The choice of intrinsic preferences decouples the complexity of preprocessing the partial order from the size of the database, and makes the order insensitive to database updates.

We formalize the specification of the partial order using a Datalog-like syntax. A user supplies a set of base rules defining a (strict) partial order \succ , and an additional recursive rule is used to transitively close the \succ relation. In some cases, user specified rules may involve constraints, such as

$$r(C_1, P_1) \succ r(C_2, P_2) \text{ :- } C_1 = C_2, P_1 < 0.8 * P_2$$

which states that a record r_1 is preferred to a record r_2 if the first column of the two records is the same, and r_1 ’s second column is less than 0.8 times the corresponding value in r_2 . This rule expresses Example 1.2.

We preprocess the set of rules by generating a least fixpoint in the sense of [11], working within an appropriate domain of constraints. A minimal requirement is that the constraint domain together with the class of allowed rules guarantee that a least fixpoint is reached after a finite number of iterations. Chomicki showed that such a fixpoint exists for preferences defined in terms of equality and $<$ (but without arithmetic) on rational numbers [4], using a result of [11].

Our first contribution is to show that a fixpoint exists for a more general class of constraints that is useful for applications employing preferences, including arithmetic constraints like those in Example 1.2.

It would also be desirable to guarantee that preprocessing the rules has low complexity. The complexity of deriving all consequences of a set of rules on a relation with c columns can be exponential in c , even without arithmetic.

Example 1.4: Consider the following set of c rules on relations with c columns:

$$\begin{aligned} r(X_1, \dots, X_{c-1}, 1) &\succ r(X_1, \dots, X_{c-1}, 0) \\ r(X_1, \dots, X_{c-2}, 1, X_c) &\succ r(X_1, \dots, X_{c-2}, 0, X_c) \\ &\dots \\ r(1, X_2, \dots, X_c) &\succ r(0, X_2, \dots, X_c) \end{aligned}$$

The transitive closure of these rules contains terms of the form

$$r(Y_1, \dots, Y_c) \succ r(Z_1, \dots, Z_c)$$

where for each $i = 1, \dots, c$, either $Y_i = 1$ and $Z_i = 0$, or $Y_i = Z_i$ represent a common variable. Each combination of these possibilities is generated, except for

the case where $Y_i = Z_i$ for all i . There are $2^c - 1$ such terms in the transitive closure, none of which is subsumed by another. \square

In typical applications, c may be large because each record may have many descriptive attributes that are pertinent to the definition of relevance.

Our second contribution is to describe how to define and compose preferences in a way that limits the size of the transitive closure. We also introduce a new notion of “covering composition” that eliminates some semantic anomalies present in previous notions of composition.

Finally, we consider the relationship between preferences and indexing. Once a preference set has been preprocessed, one can predict the kinds of database lookups necessary to find records preferable to a given record. By using appropriate index structures, one can achieve good data complexity bounds.

Proofs are omitted from the main text, and can be found in Appendix A.

2 Preference Classes

Motivated by preferences like those of Example 1.2, we define a class of preferences that allows a particular form of arithmetic comparisons in the constraints. We define two languages within which constraints may be expressed. All constraint languages allow the Boolean values *true*, denoted \top (corresponding to the empty constraint) and *false*, denoted \perp (corresponding to an inconsistent constraint).

Definition 2.1: The constraint language \mathcal{L}_S is defined over a finite set of distinct constants S , and contains

- constants from S and variables as basic expressions,
- the standard equality predicate $=$ on expressions,
- conjunctions of predicates. \square

Definition 2.2: Let $C(X_1, \dots, X_n, Y_1, \dots, Y_n)$ be a constraint in \mathcal{L}_S over the variables $X_1, \dots, X_n, Y_1, \dots, Y_n$. We say C is *=-allowed*, if (a) Every equation c in C has the form $X_i = Y_j$ for some i and j , or $X_i = a$ for some i and some constant a in S , or $Y_j = a$ for some j and some constant a in S ; and (b) No two equations of the form $X_i = Y_j$ in C share the same Y_j variable. Note that the empty constraint, i.e., the Boolean value \top , is *=-allowed* (vacuously). The false constraint \perp is also *=-allowed*. \square

Definition 2.3: The constraint language $\mathcal{L}_{\mathcal{R}}$ is defined over the nonnegative real numbers (denoted here by \mathcal{R}), and contains

- variables and constants from \mathcal{R} as basic expressions,
- composition of subexpressions using multiplication and addition,
- the standard ordering predicate $<$ on expressions,
- the standard equality predicate $=$ on expressions,
- conjunctions of predicates. \square

Definition 2.4: Let $C(X_1, \dots, X_n, Y_1, \dots, Y_n)$ be a constraint in $\mathcal{L}_{\mathcal{R}}$ over the variables $X_1, \dots, X_n, Y_1, \dots, Y_n$. We say C is *<-allowed*, if it has the following syntactic form: (a) Every inequality c in C has the form

$$X_i + b < aY_j$$

for some i and j , where b is a nonnegative constant, and a is a constant in $(0, 1]$. (b) Every equality e in C has the form $X_i = Y_j$ for some i and j , and no two equalities in C share the same Y_j . (c) Each X_i may appear in at most one equality or inequality in C . Note that the empty constraint, i.e., the Boolean value \top , is *<-allowed*, as is the false constraint \perp . \square

For notational convenience, we may sometimes write an *<-allowed* inequality as

$$X_i < aY_j - b$$

even though subtraction is not strictly part of $\mathcal{L}_{\mathcal{R}}$.

One can define an analogous class of *>-allowed* constraints of the form

$$X_i > a_1Y_1 + a_2Y_2 + \dots + a_nY_n + b$$

by requiring $b \geq 0$ and each a_j to be either zero or at least 1. This class is slightly more general than *<-allowed* constraints in that addition of multiple Y_i terms is permitted in constraints. Nevertheless, the main ideas are analogous to those described below for *<-allowed* rules, and are omitted due to space limitations.

The syntax for *=-allowed* and *<-allowed* constraints is not symmetric, in that it treats the variables X_1, \dots, X_n differently from the variables Y_1, \dots, Y_n . The reasons for this choice will become apparent when we discuss indexing in Section 4.2.

Definition 2.5: An *=-allowed* rule is said to be *rigid* if for every occurrence of an equation $X_i = Y_j$, $i = j$. An *<-allowed* rule is said to be *rigid* if for every occurrence of an equation $X_i = Y_j$, $i = j$, and for every occurrence of an inequality $X_i < aY_j - b$, $i = j$. \square

Rigid rules require that column variables in one record are compared (via $=$ or $<$) only with the same column variables in another record. Rigid rules have certain nice composition properties, described in Section 3.

When we have two preference facts of the form $r_1 \succ r_2$ and $r_2 \succ r_3$, we will apply transitivity to infer $r_1 \succ r_3$. The following two lemmas show that this transitivity step can be done within the corresponding constraint language while eliminating variables from r_2 .

Lemma 2.1: Let $C_1(X_1, \dots, X_n, Y_1, \dots, Y_n)$ and $C_2(Y_1, \dots, Y_n, Z_1, \dots, Z_n)$ be $=$ -allowed constraints in \mathcal{L}_S . Then $\exists Y_1, \dots, Y_n : C_1 \wedge C_2$ can be expressed as an $=$ -allowed constraint $C_3(X_1, \dots, X_n, Z_1, \dots, Z_n)$ in \mathcal{L}_S . \square

Lemma 2.2: Let $C_1(X_1, \dots, X_n, Y_1, \dots, Y_n)$ and $C_2(Y_1, \dots, Y_n, Z_1, \dots, Z_n)$ be $<$ -allowed constraints in \mathcal{L}_R . Then $\exists Y_1, \dots, Y_n : C_1 \wedge C_2$ can be expressed as an $<$ -allowed constraint $C_3(X_1, \dots, X_n, Z_1, \dots, Z_n)$ in \mathcal{L}_R . \square

We will consider collections of rules in which preferences are specified using $<$ -allowed constraints from \mathcal{L}_R , and $=$ -allowed constraints from \mathcal{L}_S for an appropriate set \mathcal{S} . We will partition the variables into two groups: those that are of “ \mathcal{L}_S type” and those that are of “ \mathcal{L}_R type.” By convention, we shall write the \mathcal{L}_S variables first, and assume that $0 \leq q \leq n$ of the n variable pairs are of \mathcal{L}_S type. Because the variables of each type are disjoint, we can apply Lemmas 2.2 and 2.1 together when applying a transitivity rule to two preferences.

Example 2.1: Consider a car database in which cars have a color and a price. The color column has type \mathcal{L}_S where \mathcal{S} is a set of colors. The price column has type \mathcal{L}_R . The rule “I prefer a red car to a blue car if the price of the red car is more than \$100 below the price of the blue car” is expressed as

$$r_1 : r(C_1, P_1) \succ r(C_2, P_2) :- \\ C_1 = \text{red}, C_2 = \text{blue}, P_1 < P_2 - 100.$$

The rule “Among two cars of the same color, I prefer one if its price is less than 0.8 times the price of the other” from Example 1.2 is expressed as

$$r_2 : r(C_1, P_1) \succ r(C_2, P_2) :- \\ C_1 = C_2, P_1 < 0.8 * P_2.$$

As a shorthand, we may sometimes repeat variables or put constants in the head of a rule, as in Example 1.4. \square

Example 2.2: Consider the rules of Example 2.1, for which we will apply one round of transitivity. There

are four possible rule compositions: r_1 with itself, r_2 with itself, r_1 with r_2 and r_2 with r_1 . Using the constructions of Lemmas 2.1 and 2.2, the rule bodies of the four compositions are:

$$r_{11}: C_1 = \text{red}, C_2 = \text{blue}, \text{red} = \text{blue}, P_1 < P_2 - 200 \\ r_{22}: C_1 = C_2, P_1 < 0.64 * P_2 \\ r_{12}: C_1 = \text{red}, C_2 = \text{blue}, P_1 < 0.8 * P_2 - 100 \\ r_{21}: C_1 = \text{red}, C_2 = \text{blue}, P_1 < 0.8 * P_2 - 80$$

The body of rule r_{11} is inconsistent because of the $\text{red} = \text{blue}$ predicate, and so r_{11} can be dropped since it generates no answers. The body of r_{22} is consistent. However, it is subsumed by the body of rule r_2 , since $P_1 < 0.64 * P_2$ is a more restrictive constraint than $P_1 < 0.8 * P_2$ on the nonnegative real numbers. Thus r_{22} can be dropped. Similarly, the body of r_{12} is subsumed by the body of r_1 , and can be dropped. The body of r_{21} is not subsumed by either r_1 or r_2 .

A second transitivity step can be applied to r_{21} , to get r_{211} and r_{212} . r_{211} is inconsistent for the same reasons as r_{11} . r_{212} is subsumed by r_{21} . Thus, the complete transitive closure of $\{r_1, r_2\}$ is $\{r_1, r_2, r_{21}\}$. \square

2.1 Termination

The process of computing the transitive closure in Example 2.2 terminated. Is this a general property of the class of rules we are considering? That rules over \mathcal{L}_S terminate under transitive closure is relatively easy to see, because the rules can be written as Datalog rules with nonground facts. The least fixpoint computation on such rules terminates when duplicate elimination is based on subsumption, since there are only finitely many possible fact variants that can be generated.

However, in \mathcal{L}_R , there is no such finiteness property. In fact, there are infinite collections of constraints for which no single constraint is subsumed by the others. (Imagine a collection of half-planes whose boundaries are tangent to the unit circle in a single quadrant.)

The main result of this section is that the transitive closure computation of a finite set of $<$ -allowed rules over \mathcal{L}_R always terminates. A simple subsumption check that is sound but not complete is sufficient for guaranteeing termination.

Definition 2.6: Let $X_i < aY_j - b$ and $X_i < a'Y_j - b'$ be constraints, denoted by C and C' respectively. We say C *dominates* C' if $b' \geq b$ and $a' \leq a$. An equality constraint is said to dominate itself. A conjunction C_1 of constraints dominates a conjunction C_2 of constraints if every conjunct of C_1 dominates some conjunct of C_2 . \square

It is straightforward to show that on the nonnegative reals \mathcal{R} , if C dominates C' then C is implied by C' .

Theorem 2.3: Transitive closure of $<$ -allowed rules in $\mathcal{L}_{\mathcal{R}}$ terminates under an evaluation that checks whether a newly generated constraint is dominated by any single previously generated constraint. \square

We cannot extend the definition of $<$ -allowed constraints to include addition, as in the discussion of $>$ -allowedness above.

Example 2.3: Consider the (non $<$ -allowed) constraint

$$(X_1 < Y_1) \wedge (X_2 < 0.5 * Y_1 + 0.5 * Y_2).$$

Combining this rule transitively with itself n times leads to the constraint

$$(X_1 < Y_1) \wedge (X_2 < (1 - 0.5^n)Y_1 + (0.5)^n Y_2).$$

Each such constraint is not subsumed by the set of previously generated constraints. \square

The termination in each of $\mathcal{L}_{\mathcal{S}}$ and $\mathcal{L}_{\mathcal{R}}$ implies termination in the combined language, since the columns of each type are disjoint.

There are additional constraint languages that have previously been considered, and could be used equally well in the following sections. For example, a language allowing equalities and inequalities (but no arithmetic) on constants and variables [4] would be useful for expressing some kinds of constraint. When the transitive closure can be computed in finite time, the same general techniques can be used.

2.2 Consistency

It is possible that a user may specify a set of constraints that violate the requirements of a partial order, even if they satisfy the syntactic conditions defined above. This violation can be detected during the transitive closure computation. If (at any stage of the transitive closure computation) the constraint in a rule is consistent with $X_1 = Y_1 \wedge \dots \wedge X_n = Y_n$, then we have a cycle that violates the partial order. Users can be told which rules participated in the cycle, and therefore need to be modified.

Example 2.4: Consider the rules

$$\begin{aligned} r(C_1, P_1) \succ r(C_2, P_2) & :- C_2 = blue, P_1 < 0.8 * P_2 \\ r(C_1, P_1) \succ r(C_2, P_2) & :- C_1 = blue, C_2 = red. \end{aligned}$$

then the rules are individually consistent but the transitive closure contains the rule

$$r(C_1, P_1) \succ r(C_2, P_2) :- C_1 = blue, C_2 = blue.$$

which violates the partial order requirement. \square

3 Composing Preferences

Suppose we have two preference orders \succ_1 and \succ_2 on relations r_1 and r_2 respectively. Composition is defined on a relation r whose domain is the cross product of the domains of r_1 and r_2 . We write $r(\vec{x}_1, \vec{x}_2)$ to distinguish the attributes of r that come from each of r_1 and r_2 respectively.

The prioritized composition [12], $\succ = \succ_1 \& \succ_2$, is defined as: $r(\vec{x}_1, \vec{x}_2) \succ r(\vec{y}_1, \vec{y}_2)$ iff

$$r_1(\vec{x}_1) \succ_1 r_1(\vec{y}_1) \text{ or } (\vec{x}_1 = \vec{y}_1 \text{ and } r_2(\vec{x}_2) \succ_2 r_2(\vec{y}_2))$$

Prioritized composition gives priority to the first preference order, and uses the second order only to break ties in the first order.

The Pareto composition [12], $\succ = \succ_1 \otimes \succ_2$, is defined as: $r(\vec{x}_1, \vec{x}_2) \succ r(\vec{y}_1, \vec{y}_2)$ iff

$$\begin{aligned} & r_1(\vec{x}_1) \succ_1 r_1(\vec{y}_1) \text{ and } r_2(\vec{x}_2) \succ_2 r_2(\vec{y}_2), \text{ or} \\ & r_1(\vec{x}_1) \succ_1 r_1(\vec{y}_1) \text{ and } \vec{x}_2 = \vec{y}_2, \text{ or} \\ & \vec{x}_1 = \vec{y}_1 \text{ and } r_2(\vec{x}_2) \succ_2 r_2(\vec{y}_2). \end{aligned}$$

Pareto composition treats the component orders symmetrically. A record must be strictly better according to at least one of the component orders than a comparison record, and either better or equal according to the other component order.

The strict composition $\succ = \succ_1 \times \succ_2$, is defined as: $r(\vec{x}_1, \vec{x}_2) \succ r(\vec{y}_1, \vec{y}_2)$ iff

$$r_1(\vec{x}_1) \succ_1 r_1(\vec{y}_1) \text{ and } r_2(\vec{x}_2) \succ_2 r_2(\vec{y}_2)$$

Strict composition is also symmetric, and requires that a record must be strictly better according to all of the component partial orders.

Both prioritized composition and Pareto composition define partial orders [12], as does strict composition.

3.1 Covering Preferences

For $<$ -allowed rules, Pareto-composition and prioritized composition may give unintuitive results, as shown in the following example.

Example 3.1: Consider a preference \succ_1 on prices defined as in some of the examples above via a rule with body like $P_1 < 0.8 * P_2$. Let \succ_2 represent a preference for red cars over blue cars. According to $\succ_1 \otimes \succ_2$ (and also $\succ_1 \& \succ_2$), (75,red) and (100,red) are both preferred to (100,blue), while (90,red) is not. This problem reflects a conflict between the philosophy behind $<$ -allowed rules and equality. An $<$ -allowed rule tries to express the requirement that small differences do *not* lead to a preference relationship. The equality in Pareto-composition and prioritized composition expresses the opposite: a preference is possible when the column values match exactly. \square

To resolve this problem, we define the notion of covering constraints. Covering constraints relax constraints just enough to bridge the gap between the partial order and an equality constraint.

Definition 3.1: Let R be a set of $<$ -allowed rules. Define the *cover* of R , denoted by \widehat{R} , to be the rules formed from R by replacing all multiplicative constants in the body of the rules by 1, and all additive constants in the bodies of rules by 0. If \succ is the partial order defined by R , then $\widehat{\succ}$ is the partial order defined by \widehat{R} . \square

\widehat{R} is $<$ -allowed, and defines a partial order. Further, by the construction of \widehat{R} , all preferences implied by R are also implied by \widehat{R} , i.e., $\succ \subseteq \widehat{\succ}$. We can now define covering versions of prioritized and Pareto composition.

Definition 3.2: Given partial orders \succ_1 and \succ_2 , where \succ_1 is defined by $<$ -allowed rules, define

$$\begin{aligned}\succ_1 \widehat{\&} \succ_2 &= \succ_1 \& \succ_2 \cup \widehat{\succ_1} \times \succ_2 \\ \succ_1 \widehat{\otimes} \succ_2 &= \succ_1 \otimes \succ_2 \cup \widehat{\succ_1} \times \succ_2\end{aligned}\quad \square$$

Definition 3.2 defines “one-sided” versions of composition using covered preferences. It is also possible to define a “two-sided” version of Pareto composition, but the details are omitted here.

In general, $\succ_1 \widehat{\&} \succ_2$ and $\succ_1 \widehat{\otimes} \succ_2$ are not partial orders, because they may violate transitivity. They do define partial orders for an important subclass of rules.

Lemma 3.1: $\succ_1 \widehat{\&} \succ_2$ and $\succ_1 \widehat{\otimes} \succ_2$ define partial orders if \succ_1 is defined using a single rigid rule. \square

Example 3.2: Revisiting Example 3.1, suppose we instead used the covering versions $\succ_1 \widehat{\otimes} \succ_2$ (or $\succ_1 \widehat{\&} \succ_2$) to define the preferences. Then even though $90 \not\succeq_1 100$, we do have $90 \widehat{\succ}_1 100$, and so (90,red) is now preferred to (100,blue). Note that (90,red) is still not preferred to (100,red). \square

A potential alternative to these notions of composition, and the idea of covering, is defined by Chomicki [4]. He defines a different version of prioritized composition that we will call “triangle composition” to distinguish it from the previously defined notion of prioritized composition.

Definition 3.3: The *indifference relation* \sim for a binary relation \succ is defined as

$$x \sim y \text{ iff } x \not\succeq y \text{ and } y \not\succeq x.$$

Given two binary relations \succ_1 and \succ_2 , the *triangle composition* $\succ = \succ_1 \triangleright \succ_2$ is defined as

$$(x_1, x_2) \succ (y_1, y_2) \text{ iff } \\ x_1 \succ_1 y_1 \text{ or } (x_1 \sim_1 y_1 \text{ and } x_2 \succ_2 y_2)$$

where \sim_1 is the indifference relation for \succ_1 . \square

The notion of indifference is intended to capture the absence of a preference between elements. The triangle composition of two strict partial orders is not necessarily transitive [4]. Triangle composition can give problematic preference relations, as illustrated in the following example.

Example 3.3: Consider again the rules from Example 3.1, extending \succ_2 so that red cars are preferred to blue cars, which are in turn preferred to green cars. Let \succ denote $\succ_1 \triangleright \succ_2$. Then according to the definition of triangle composition, we have

$$(100, \text{red}) \succ (90, \text{blue}) \succ (75, \text{green}) \succ (100, \text{red}).$$

Thus, there is a cycle of preferences according to triangle composition. \square

Cycles of preferences present several obvious problems. The transitive closure of such a relation is not an order, and would be considered inconsistent (see Section 2.2). An evaluation method that discards a tuple in favor of a preferred tuple may get stuck in an infinite loop, even in a finite domain.

In contrast, it is relatively easy to see that cycles do not occur in the covering versions of composition (Definition 3.2). In any chain of tuples, each preferred to the next, the first argument will always be decreasing relative to $\widehat{\succ}_1$.

Example 3.4: The cycle identified in Example 3.3 does not occur with covering constraints. $(100, \text{red}) \not\succeq (90, \text{blue})$, even though the value 100 is within 20% of the value 90. This difference highlights the importance of relaxing constraints slightly, to bridge the gap between inequalities and equality, without relaxing the constraints so much that inconsistency results. \square

3.2 Complexity

We aim to investigate the complexity of prioritized, Pareto, and strict composition. (The covering versions of these compositions can be expressed in terms of the three basic compositions.) At this level, we are interested in the size of the computed transitive closure of a composed partial order as a function of the sizes of the transitive closures of the component partial orders. Given a partial order \succ specified by a set of rules, we let $S(\succ)$ denote the cardinality of the transitive closure.

Definition 3.4: Let R_1 and R_2 be rules defining the partial orders \succ_1 and \succ_2 on domains (\vec{X}_1, \vec{Y}_1) and (\vec{X}_2, \vec{Y}_2) respectively.

Define $R_1 \& R_2$ to be the rule set obtained by extending the domain of both rule sets to $(\vec{X}_1 \cup \vec{X}_2, \vec{Y}_1 \cup \vec{Y}_2)$, and adding the constraint $\vec{X}_1 = \vec{Y}_1$ to the body of rules in R_2 .

Define $R_1 \otimes R_2$ to be the rule set obtained by extending the domain of both rule sets to $(\vec{X}_1 \cup \vec{X}_2, \vec{Y}_1 \cup \vec{Y}_2)$, adding the constraint $\vec{X}_1 = \vec{Y}_1$ to the body of rules in R_2 , and adding the constraint $\vec{X}_2 = \vec{Y}_2$ to the body of rules in R_1 .

Define $R_1 \times R_2$ to be the rule set obtained as follows. Define a rule r over $(\vec{X}_1 \cup \vec{X}_2, \vec{Y}_1 \cup \vec{Y}_2)$ in which the body of r applies r_1 to (\vec{X}_1, \vec{Y}_1) and r_2 to (\vec{X}_2, \vec{Y}_2) .

Rules for covering compositions can be derived as unions of rules defined above, according to Definition 3.2. \square

Note that if R_1 and R_2 are $<$ -allowed then so are $R_1 \& R_2$ and $R_1 \otimes R_2$, and similarly for $=$ -allowed rules. While $R_1 \times R_2$ is not necessarily allowed, we will identify classes of rules below for which $R_1 \times R_2$ can be written as an allowed set of rules.

Lemma 3.2: The $\&$, \otimes , and \times operators on rules faithfully implement the corresponding operations on the underlying partial orders. \square

Lemma 3.3: $S(\succ_1 \& \succ_2) = S(\succ_1) + S(\succ_2)$. \square

Example 3.5: We illustrate Lemma 3.3 by considering \succ_1 on (X_1, Y_1) defined by $X_1 < Y_1$, and \succ_2 on (X_2, Y_2) defined by $X_2 = 1, Y_2 = 0$. The rules in the composition are

$$\begin{aligned} c_1 \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 < Y_1 \\ c_2 \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 = Y_1, X_2 = 1, Y_2 = 0 \end{aligned}$$

c_1 composed with c_2 yields

$$\begin{aligned} c_{12} \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 < Y_1, Y_2 = 0 \\ c_{21} \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 < Y_1, X_2 = 1 \end{aligned}$$

both of which are subsumed by c_1 . \square

Lemma 3.4: $S(\succ_1 \otimes \succ_2) = (S(\succ_1) + 1)(S(\succ_2) + 1) - 1$. \square

Example 3.6: We illustrate the construction in Lemma 3.4 by considering \succ_1 on (X_1, Y_1) defined by $X_1 < Y_1$, and \succ_2 on (X_2, Y_2) defined by $X_2 < Y_2$. The rules in the Pareto composition are

$$\begin{aligned} c_1 \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 < Y_1, X_2 = Y_2 \\ c_2 \quad r(X_1, X_2, Y_1, Y_2) &:- X_1 = Y_1, X_2 < Y_2 \end{aligned}$$

c_1 and c_2 are closed under self-composition. c_1 composed with c_2 (in either order) yields

$$c_3 \quad r(X_1, X_2, Y_1, Y_2) :- X_1 < Y_1, X_2 < Y_2$$

The transitive closure contains three rules. \square

When considering strict composition, we would like to identify circumstances when $\succ_1 \times \succ_2$ can be expressed as an $=$ -allowed or $<$ -allowed set of rules. Under such circumstances, we will be able to show that strict composition has good scaling properties, and we will not suffer the exponential blowup inherent in Lemma 3.4.

Lemma 3.5: A rule set containing a single rigid $=$ -allowed rule is transitively closed in \mathcal{L}_S . A rule set containing a single rigid $<$ -allowed rule is transitively closed in \mathcal{L}_R . \square

Since single rigid rules are transitively closed, we can rewrite $\succ_1 \times \succ_2$ if one of the partial orders (say \succ_2) is defined by a single rigid rule. The rules for r_1 and r_2 can be unfolded into the combining rule in $\succ_1 \times \succ_2$, resulting in one rule for $\succ_1 \times \succ_2$ for each rule in \succ_1 . Further, the resulting unfolded rules are allowed if the rules for \succ_1 are allowed.

Lemma 3.6: If \succ_2 is defined by a single rigid allowed rule, then $S(\succ_1 \times \succ_2) = S(\succ_1)$. \square

Lemmas 3.3, 3.4, and 3.6 have important implications for how one might build complex preferences out of simpler components while keeping the overall complexity under control. Lemma 3.3 indicates that a prioritized composition scales additively, meaning that the number of probes to the database required to determine preference is likely to be manageable.

On the other hand, Lemma 3.4 indicates that a Pareto-composition scales multiplicatively, and so the number of probes may be exponential in the number of component partial orders. This exponential behavior holds even when the transitive closure of a component partial order has just one rule, since the multiplicative term is $S(\succ) + 1$. Example 1.4 is an instance of this observation.

In contrast with Example 1.4, the following similar set of rules is better behaved, because it is generated using prioritized composition rather than Pareto composition.

Example 3.7: Consider the following set of c rules on relations with c columns:

$$\begin{aligned} r(X_1, \dots, X_{c-1}, 1) &\succ r(X_1, \dots, X_{c-1}, 0) \\ r(X_1, \dots, X_{c-2}, 1, _) &\succ r(X_1, \dots, X_{c-2}, 0, _) \\ &\dots \\ r(1, _ , \dots, _) &\succ r(0, _ , \dots, _) \end{aligned}$$

In these rules, an underscore denotes an unconstrained variable; different instances of “ $_$ ” represent different variables. These rules are transitively closed: composing two rules gives an instance of an existing rule. \square

Lemma 3.6 shows that this exponential behavior can be broken if we use strict composition, and if one of the component orders is defined by a single strict allowed rule.

Suppose that we wish to limit the size of the transitive closure to be polynomial in c where c is the number of columns. We can achieve this effect as follows:

- Each basic partial order is allowed to mention at most K columns, for some constant K . This decouples the complexity of basic orders from the number of columns.

- Prioritized composition can be used as needed.
- Strict composition with an order defined by a single strict allowed rule can be used as needed.
- Pareto composition can be used a bounded number of times.

Example 3.8: Consider an example for finding “good” flight plans from a travel agency database. The flight records include columns: S, T, F, P that respectively give the number of stops S in the flight plan, the total time of the flight plan, whether the flight is with an airline for which you are a member of the frequent flier program ($F = yes$) or not ($F = no$), and the price P of the flight plan. The three preference relations of a flier might be: (a) Flights with fewer stops are preferable.

$$f_1 \quad r_1(S_1) \succ_1 r_1(S_2) :- \quad S_1 < S_2$$

(b) Flights shorter by at least an hour are preferable

$$f_2 \quad r_2(T_1) \succ_2 r_2(T_2) :- \quad T_1 < T_2 - 1$$

and (c) Cheaper flights are preferred but a non-frequent flier flight must be 20% cheaper to be preferred over a frequent flier flight.

$$\begin{aligned} f_3 \quad r_3(F_1, P_1) \succ_3 r_3(F_2, P_2) & :- \\ & \quad F_1 = F_2, P_1 < P_2 \\ f_4 \quad r_3(F_1, P_1) \succ_3 r_3(F_2, P_2) & :- \\ & \quad F_1 = yes, F_2 = no, P_1 < P_2 \\ f_5 \quad r_3(F_1, P_1) \succ_3 r_3(F_2, P_2) & :- \\ & \quad F_1 = no, F_2 = yes, P_1 < 0.8 * P_2 \end{aligned}$$

The final ordering relation might be $\succ_1 \ \& (\succ_2 \ \& \widehat{\succ}_3)$. That is better flights have fewer stops, and if the number of stops is equal, then better flights are either shorter by more than an hour, or both no longer and better with respect to (qualified) price.

The ordering can be defined by a collection of rules with head

$$r(S_1, T_1, F_1, P_1) \succ r(S_2, T_2, F_2, P_2) :-$$

and bodies given below:

$$\begin{aligned} S_1 < S_2 \\ S_1 = S_2, T_1 < T_2 - 1 \\ S_1 = S_2, T_1 = T_2, F_1 = F_2, P_1 < P_2 \\ S_1 = S_2, T_1 = T_2, F_1 = yes, F_2 = no, P_1 < P_2 \\ S_1 = S_2, T_1 = T_2, F_1 = no, F_2 = yes, P_1 < 0.8 * P_2 \\ S_1 = S_2, T_1 < T_2, F_1 = F_2, P_1 < P_2 \\ S_1 = S_2, T_1 < T_2, F_1 = yes, F_2 = no, P_1 < P_2 \\ S_1 = S_2, T_1 < T_2, F_1 = no, F_2 = yes, P_1 < 0.8 * P_2 \quad \square \end{aligned}$$

```
Pareto() { /* All entries start unmarked */
  while (there are unmarked records) {
    X := some unmarked record;
    while (X is unmarked & there is
      some Y such that Y > X)
      { mark X; X := Y; }
    if (X is unmarked)
      { output X; mark X; }
  } }
```

Figure 1: Outputting the Pareto-optimal elements

3.3 Ceteris Paribus vs. Global Rules

By using Datalog-like rules, we have the flexibility to define preferences that span the range between global preferences and ceteris-paribus preferences. A global preference is something like “ C_1 is better than C_2 if the price of C_1 is less than the price of C_2 ,” where all attributes other than the price are irrelevant to the preference function. A ceteris-paribus preference [5, 3] is something like “ C_1 is better than C_2 if the price of C_1 is less than the price of C_2 , *all else being equal*.” The “all else being equal” clause implies that C_1 and C_2 will be comparable according to this rule only if all attributes besides price are the same in C_1 and C_2 .

There has been some discussion in the preference literature about the relative merits of each of these approaches [5]. Compared to global preferences, ceteris-paribus permits a finer granularity of expressiveness by providing some context to the preference. However, this added expressiveness comes at a cost in the size of the transitive closure. Our discussion of the complexity of composition adds to this discussion by quantifying the cost of a pure ceteris paribus approach. Example 1.4 is a classic ceteris paribus formulation, and the transitive closure has size exponential in the number of columns. Example 3.7 has some ceteris paribus features. The “all else being equal” is applied partially, according to a lexicographic ordering of attributes. By doing so, the size of the transitive closure has become linear in the number of columns.

4 Preferences and Indexing

4.1 The Pareto-Optimal Set

Based on the preference relation, one can generate as output the Pareto-optimal set of records, i.e., those records R for which there is no R' in the database with $R' \succ R$. Pseudo-code is given in Figure 1. We assume that the database records have all been inserted into an appropriate index structure. (The choice of index data structure will be discussed in Section 4.2.).

An index structure may be useful to efficiently find a single record Y better than a given record X . If $O(f(n))$ is the complexity of this lookup step, it is relatively easy to see that the overall complexity of the Pareto algorithm is $O(nf(n))$. Note that the mark-related operations of the algorithms above can be performed

in constant time and linear space using a hash table.

Algorithm **Pareto** shares some similarities with the *TA* algorithm [7]. **Pareto** outputs one of the top records when it knows that the record is undominated. *TA* outputs the top- k objects, when no other object in the data set can have a higher grade than the current k best objects. However, their query models are different: *TA* needs to aggregate information from several indexes to get complete object information, and unlike **Pareto** does not need to consider all objects but can stop as soon as it has reached some guarantees on the scores of the k best objects.

4.2 Index Structures

Given a set of rules, one applies the transitive closure operation to get a set of rules of the form

$$r(X_1, \dots, X_n) \succ r(Y_1, \dots, Y_n) :- \\ C(X_1, \dots, X_n, Y_1, \dots, Y_n)$$

where C is a constraint (from the constraint language) on the variables $X_1, \dots, X_n, Y_1, \dots, Y_n$. For the Pareto algorithm, we need to determine records that are “better” than a given record $r(a_1, \dots, a_n)$, where the a_i are known constants. We must therefore search the database for records $r(X_1, \dots, X_n)$ satisfying $C(X_1, \dots, X_n, a_1, \dots, a_n)$.

Example 4.1: Consider the rules r_1, r_2, r_{21} from Examples 2.1 and 2.2. Suppose we have a database record $d = r(\text{blue}, 1000)$, and want to determine records that are better than d . Substituting the constants from d as values for C_2 and P_2 in the three rules yields the queries $(C_1 = \text{red}, P_1 < 900)$, $(C_1 = \text{blue}, P_1 < 800)$, and $(C_1 = \text{red}, P_1 < 720)$. (If the system is sufficiently smart, it might notice that the third query is redundant, being a special case of the first query.) For this example, a tree index on (C, P) would allow such records to be found efficiently. \square

Example 4.1 highlights the advantages of using =-allowed and <-allowed rules. The constraints that are generated when values for the Y_j variables are fixed are guaranteed to be of the form $X_i = a$ or $X_i < b$, where a and b are constants. Further, at most one such constraint is needed for each variable — other constraints on that variable will be redundant. These will be easier to support directly using standard indexes than non-allowed rules. For example, the non=--allowed rule

$$r(X_1, X_2) \succ r(Y_1, Y_2) :- X_1 = X_2, Y_1 = a, Y_2 = b$$

yields a query of the form $r(X, X)$, which is not efficiently supported in standard index structures.

For allowed rules, a query template will be a collection of equality and inequality predicates on distinct variables. Let us call the equality variables E_1, \dots, E_p ,

and the inequality variables I_1, \dots, I_q . If $q = 0$, then a hash table or tree-index on (E_1, \dots, E_p) are good candidates. If $q = 1$, one could also employ a hash table on (E_1, \dots, E_p) where the elements of the hash table are tree indexes on I_1 on the corresponding subsets of the data. A simple tree index on (E_1, \dots, E_p, I_1) is another candidate. If $q > 1$, we could employ a hash table of kd trees or a hash table of range-trees, where the trees index the I_1, \dots, I_q columns. The various choices and their complexity are summarized in Table 1. Note that the probe time refers to a probe that returns a single match. (In a database context, one would use I/O efficient versions of these tree structures, such as Bkd-trees [15] or KDB-trees [16].)

In general, the transitive closure rules may yield multiple lookup templates, each of which may be best handled by a different kind of index. It is likely to be worthwhile to build multiple indexes to facilitate such lookups. The lookup templates are database independent, and so building multiple indexes does not add to the data complexity of the approach. In practice, the number of indexes needed is likely to be small. In some cases, such as when one template is a prefix of another, a single index can support multiple templates.

If we do not have control over which indexes are built, then we should simply choose the best available index for each template. For example, if we need to probe on (E_1, \dots, E_p, I_1) , and an index exists on (E_1, \dots, E_{p-1}) then we can use the index to narrow down the search, and explicitly check the conditions on E_p and I_1 on each returned record until a match is found.

4.3 Alternative Presentations

There are other ways to give prominence to preferred records besides outputting only the Pareto-optimal records. One could output all records in strata, with the first stratum being the Pareto-optimal records, the second stratum being those that are Pareto-optimal among the remaining records, and so on [4]. One could alternatively generate a total order of the records that is consistent with the preference partial order, and output records according to that total order. A third alternative would be to output records r in an ascending order with respect to the number of records in the data set that dominate r .

Each of these alternatives could potentially be achieved by using the **Pareto** algorithm as a subroutine, although it is possible that faster direct methods exist for some of these alternatives. We outline here one way to present records according to the third alternative above; a full discussion of such alternatives is left to future work.

Consider a B-tree index structure augmented with counts at each node, representing the number of ele-

Structure	Applicability	Space	Build-time	Probe-time	Pareto
Hash table	$q = 0$	$O(n)$	$O(n)$	$O(1)$	$O(Tn)$
B-tree	$q \leq 1$	$O(n)$	$O(n \log n)$	$O(\log n)$	$O(Tn \log n)$
Hash table of kd trees	$q = k > 1$	$O(n)$	$O(n \log n)$	$O(n^{1-1/k})$	$O(Tn^{2-1/k})$
Hash table of k dimensional range trees	$q = k > 1$	$O(n \log^{k-1} n)$	$O(n \log^{k-1} n)$	$O(\log^k n)$	$O(Tn \log^k n)$

Table 1: Space and time complexity as a function of the number of records n , and the size T of the transitive closure of the rules, for various index structures.

ments in the subtree rooted at that node. Such counts can be maintained without increasing the asymptotic complexity of insert/delete/probe operations. Using these counts, it is possible to return the number of records preferable to a given record according to a single preference tuple in time logarithmic in the number of records in the index.

There are many preference tuples in the transitive closure, and so we need to combine the counts for each such preference tuple. However, naively adding the counts may overcount the number of dominating records, because a single record may be preferable according to more than one rule in the transitive closure. One can deal with this issue in three possible ways.

1. One could look for sets of rules for which overcounting is guaranteed not to happen because two rules will never relate the same pair of records. See Example 4.2 below. In such a case, adding the counts will correctly generate the total number of dominating records.
2. One could try to adjust for overcounting by subtracting counts for records that are identified by more than one rule. This solution requires extra work, and one has to be careful not to “oversubtract” for records that are generated by multiple rules.
3. One could accept that the counts are only an approximation of the number of dominating records, an approximation that is useful enough for determining the order of result presentation. Even with this approximation, the Pareto-optimal records (those with a count of 0) are correctly identified.

Once counts are generated, the records would be sorted into an increasing count order and output. The net complexity would be $O(Tn \log n)$ where T is the size of the transitive closure, and n is the number of records.

Example 4.2: Consider the rules of Example 3.7. No rule instance in the transitive closure generates the same preference between the same pair of records. To see this, let us number the rules so that rule i has the form

$$r(X_1, \dots, X_{i-1}, 1, \neg, \dots, \neg) \succ r(X_1, \dots, X_{i-1}, 0, \neg, \dots, \neg).$$

Suppose that rules j and k (where $j < k$) both generate a fact of the form

$$r(a_1, \dots, a_n) \succ r(b_1, \dots, b_n).$$

According to rule k , $b_k = 0$, $a_k = 1$, and $a_j = b_j$. According to rule j , $b_j = 0$ and $a_j = 1$, contradicting $a_j = b_j$. Thus two rules never generate a common fact. Therefore, for this set of rules, one can compute the number of dominating records as the sum of the number of dominating records for each rule.

Suppose there was a B-tree index on r over (X_1, \dots, X_n) , augmented with counts as described above. Consider a record x that is 0 in positions j and k ($j < k$) and 1 elsewhere. Using rule j , we would use the B-tree to count the number of records with a prefix of j 1’s. Using rule k we would use the B-tree to count the number of records with a prefix of $1 \dots 101 \dots 1$, where there are k elements in the prefix and the zero element is at position j . These two numbers would be added to get the total number of records that dominate record x . \square

5 Conclusions

We have defined an expressive and useful constraint language using equalities, inequalities, and arithmetic. We have shown that the transitive closure of partial order constraints expressed in our language can be effectively computed. We have investigated the complexity of composing preferences into larger preference relations, and have described how to eliminate certain semantic anomalies present in previous notions of composition. We have described the selection of index structures to support the selection of Pareto-optimal records.

We believe that the constraints introduced here are practical, and can form the basis of a large-scale preference management system.

References

- [1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD Conference*, 2000.
- [2] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *IEEE Conf. on Data Engineering*, 2001.

- [3] C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21, 2004.
- [4] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4), 2003.
- [5] J. Doyle and M. P. Wellman. Representing preferences as ceteris paribus comparatives. In *Decision-Theoretic Planning: Papers from the 1994 Spring AAAI Symposium*, 1994.
- [6] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, 2003.
- [7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [8] R. Fagin and E. L. Wimmers. A formula for incorporating weights into scoring rules. *Theor. Comput. Sci.*, 239(2):309–338, 2000.
- [9] J.-B. Fourier. reported in: Analyse des travaux de l’academie royale des sciences, pendant l’annee 1824, partie mathematique. In *Histoire de l’Academie Royale des Sciences de l’Institut de France 7*, volume 7, pages xlvii–lv. 1827. (Partial English translation in: D.A. Kohler, Translation of a Report by Fourier on his work on Linear Inequalities, *Opsearch* 10 (1973) 38–42.).
- [10] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP(\mathcal{R}) constraints. *New Generation Computing*, 11:449–469, 1993.
- [11] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1), 1995.
- [12] W. Kießling. Foundations of preferences in database systems. In *VLDB*, 2002.
- [13] W. Kießling and G. Köstler. Preference SQL - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.
- [14] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.
- [15] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *SSTD*, 2003.
- [16] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD Conference*, pages 10–18, New York, NY, USA, 1981. ACM Press.

A Proofs

Lemma 2.1. Let $C_1(X_1, \dots, X_n, Y_1, \dots, Y_n)$ and $C_2(Y_1, \dots, Y_n, Z_1, \dots, Z_n)$ be =-allowed constraints in \mathcal{L}_S . Then $\exists Y_1, \dots, Y_n : C_1 \wedge C_2$ can be expressed as an =-allowed constraint $C_3(X_1, \dots, X_n, Z_1, \dots, Z_n)$ in \mathcal{L}_S .

Proof. Assume that $C_1 \wedge C_2$ is satisfiable, otherwise the result C_3 can be expressed as \perp . There is at most one equation of the form $X_i = Y_j$ in C_1 for each Y_j . We can replace each other occurrence of Y_j in equations of the form $Y_j = Z_k$ or $Y_j = a$ by X_i . The system C_3 resulting from all such replacements has at most one occurrence of Z_k (the original occurrence after replacement), and hence is =-allowed. \square

Lemma 2.2. Let $C_1(X_1, \dots, X_n, Y_1, \dots, Y_n)$ and $C_2(Y_1, \dots, Y_n, Z_1, \dots, Z_n)$ be <-allowed constraints in \mathcal{L}_R . Then $\exists Y_1, \dots, Y_n : C_1 \wedge C_2$ can be expressed as an <-allowed constraint $C_3(X_1, \dots, X_n, Z_1, \dots, Z_n)$ in \mathcal{L}_R .

Proof. We construct C_3 as the conjunction of predicates formed by combining the conjuncts within C_1 and C_2 as follows. We aim to eliminate the variables Y_j while preserving equivalence, using (extended) Fourier elimination [9, 10]

Consider the constraint c in C_1 containing X_i . It is either of the form $X_i = Y_j$ or

$$X_i < aY_j - b$$

If Y_j does not appear on the left hand side of any equation or inequality in C_2 , then no constraint on X_i in C_3 results from c . Otherwise, there is exactly one equation or inequality $c' \in C_2$ with Y_j on the left hand in C_2 , because C_2 is <-allowed. The equation or inequality resulting from the Fourier elimination is given below. If $c' \equiv Y_j = Z_k$ the resulting equation or inequality is $X_i = Z_k$ or $X_i < aZ_k - b$ respectively. If $c' \equiv Y_j < a'Z_k - b'$ the resulting inequality is $X_i < a'Z_k - b$ or $X_i < aa'Z_k - b - ab'$. Clearly the equation is <-allowed, and for each inequality the coefficient of Z_k is in $(0, 1]$ and the constant term is nonnegative, and hence it is <-allowed.

Any Z_k can appear in at most one resulting equation, since Y_j appears in at most one equation in C_1 . Each X_i appears in at most one equation or inequality since it only generates one equation or inequality.

C_3 is equivalent to $\exists Y_1, \dots, Y_n : C_1 \wedge C_2$, by the soundness and completeness of Fourier elimination [9]. \square

Theorem 2.3. Transitive closure of <-allowed rules in \mathcal{L}_R terminates under an evaluation that checks whether a newly generated constraint is dominated by any single previously generated constraint.

Proof. Suppose to the contrary that there was an infinite sequence C_1, C_2, \dots of <-allowed constraints,

where C_{m+1} is generated by the transitive closure rule using C_m and one of the original rules, and for which each element is not dominated by any of its predecessors.

No C_m can be empty, because then $C_{m+1} = C_m$ and we would have terminated. As a result, there must be at least one X_i that is constrained in every member of the sequence. Without loss of generality, let the X_i variables that are always constrained be X_1, \dots, X_p . Further, once an X_i is constrained in some C_m by an inequality constraint, all subsequent occurrences of X_i must also be in inequality constraints. (Of course, X_i may be constrained by an equality constraint in all members of the sequence.) For what follows below, we ignore the prefix C_1, \dots, C_M of the sequence that either mentions X variables not among X_1, \dots, X_p , or mentions an X_i in an equality constraint when X_i is later mentioned in an inequality constraint. We focus our attention on the remainder of the sequence.

For each X_i appearing in an inequality constraint, $i = 1, \dots, p$, let a_i^m denote the a coefficient for the constraint on X_i in C_m , and b_i^m the corresponding b term, where $m > M$. (Note that the constraints may mention different Y_j variables in each C_m .) As m increases, the b_i^m values are nondecreasing and the a_i^m values are nonincreasing, since the multipliers are positive numbers less than or equal to 1. Since there are only finitely many variables, the combination of p Y_j variables corresponding respectively to the p constraints on X_i must eventually repeat. When they do repeat, the second constraint is dominated by the first because the b value is no smaller, the a value is no larger, and the constraints mention the same X_i s, each in terms of the same Y_j s.

For each X_i appearing in an equality constraint, a similar argument applies, showing that since there are only finitely many Y_j variables that could be equated with X_i , eventually there is repetition. \square

Lemma 3.1 $\succ_1 \widehat{\&} \succ_2$ and $\succ_1 \widehat{\otimes} \succ_2$ define partial orders if \succ_1 is defined using a single rigid rule.

Proof. Since $\succ_1, \succ_2, \widehat{\succ}_1, \succ_1 \& \succ_2$, and $\succ_1 \otimes \succ_2$ are all partial orders, irreflexivity is immediate. Antisymmetry follows for $\succ_1 \& \succ_2$ and $\succ_1 \widehat{\otimes} \succ_2$ because they are subsets of the relations $\widehat{\succ}_1 \& \succ_2$ and $\widehat{\succ}_1 \otimes \succ_2$, respectively, which are themselves partial orders.

For transitivity, consider first $\succ_1 \widehat{\&} \succ_2$. Both $\succ_1 \& \succ_2$ and $\widehat{\succ}_1 \times \succ_2$ are transitively closed, being themselves partial orders. We therefore need to look only at compositions of an element from $\succ_1 \& \succ_2$ with an element from $\widehat{\succ}_1 \times \succ_2$. Suppose $(x, y) \in (\succ_1 \& \succ_2)$ and $(y, z) \in (\widehat{\succ}_1 \times \succ_2)$. We can write $\succ_1 \& \succ_2$ as the disjoint union $(\succ_1 \times D_2) \cup (=_1 \times \succ_2)$, where D_2 is the complete binary relation on the domain of \succ_2 and $=_1$ is the equality relation on the domain of \succ_1 . If $(x, y) \in (=_1 \times \succ_2)$, then $(x, z) \in \widehat{\succ}_1 \times \succ_2$. If $(x, y) \in (\succ_1 \times D_2)$, then $(x, z) \in (\succ_1 \circ \widehat{\succ}_1, D_2)$,

where \circ denotes the standard composition of binary relations. If \succ_1 is defined by a single rigid \leftarrow -allowed rule r , it is relatively easy to see that r composed with \widehat{r} gives exactly r . As a result, $(x, z) \in (\succ_1, D_2)$.

The opposite composition is similar, as is the transitivity proof for $\succ_1 \widehat{\otimes} \succ_2$. \square

Lemma 3.2. The $\&$, \otimes , and \times operators on rules faithfully implement the corresponding operations on the underlying partial orders.

Proof. We give the proof for $R_1 \otimes R_2$; the others are similar. Let \succ_1 and \succ_2 be the orders defined by R_1 and R_2 respectively. We first show $\succ_1 \otimes \succ_2 \subseteq R_1 \otimes R_2$. By the construction of $R_1 \otimes R_2$, it is clear that both $\succ_1 \times =_2$ and $=_1 \times \succ_2$ are contained in (the transitive closure of) $R_1 \otimes R_2$, where $=_i$ is the equality relation on domain i . Since any fact in $\succ_1 \times \succ_2$ can be expressed as the composition of a fact in $\succ_1 \times =_2$ and a fact in $=_1 \times \succ_2$, we are done. We now show $\succ_1 \otimes \succ_2 \supseteq R_1 \otimes R_2$. Any constraint generated by rules originally from R_1 yields a fact in $\succ_1 \times =_2$; any constraint generated by rules originally from R_2 yields a fact in $=_1 \times \succ_2$. A constraint that mixes rules from both original sources must generate a fact in $\succ_1 \times \succ_2$. \square

Lemma 3.3. $S(\succ_1 \& \succ_2) = S(\succ_1) + S(\succ_2)$.

Proof. A rule r_1 of \succ_1 when (left or right) composed with a rule r_2 of \succ_2 yields an instance of r_1 , due to the equalities on the variables in the domain of \succ_1 and the “free variables” in the domain of \succ_2 . \square

Lemma 3.4. $S(\succ_1 \otimes \succ_2) = (S(\succ_1) + 1)(S(\succ_2) + 1) - 1$.

Proof. The size of the transitive closure is equal to $S(\succ_2)$ for the rules defining \succ_2 , plus $S(\succ_1)$ for the rules defining \succ_1 , plus $S(\succ_1)S(\succ_2)$ for the preferences that can be generated by composing a preference from each transitive closure. \square

Lemma 3.5. A rule set containing a single rigid $=$ -allowed rule is transitively closed in \mathcal{L}_S . A rule set containing a single rigid \leftarrow -allowed rule is transitively closed in \mathcal{L}_R .

Proof. When a rigid rule is composed with itself, the resulting rule is, by the definition of allowedness, subsumed by the original rule. \square

Lemma 3.6. If \succ_2 is defined by a single rigid allowed rule, then $S(\succ_1 \times \succ_2) = S(\succ_1)$.

Proof. By the unfolding construction described in the main text. \square