A Framework for Expressing and Combining Preferences

Rakesh Agrawal *Edward L. Wimmers*

IBM Almaden Research Center
San Jose, CA 95120

Abstract

The advent of the World Wide Web has created an explosion in the available on-line information. As the range of potential choices expand, the time and effort required to sort through them also expands. We propose a formal framework for expressing and combining user preferences to address this problem. Preferences can be used to focus search queries and to order the search results. A preference is expressed by the user for an entity which is described by a set of named fields; each field can take on values from a certain type. The * symbol may be used to match any element of that type. A set of preferences can be combined using a generic combine operator which is instantiated with a value function, thus providing a great deal of flexibility. Same preferences can be combined in more than one way and a combination of preferences yields another preference thus providing the closure property. We demonstrate the power of our framework by illustrating how a currently popular personalization system and a real-life application can be realized as special cases of our framework. We also discuss implementation of the framework in a relational setting.

1 Introduction

The World Wide Web is suffering from abundance. The publicly indexable web contains an estimated 800 million pages [LG99]. The number of pages is anticipated to expand 1000% over the next few years [BP96]. The current on-line catalog of Amazon.com contains more than 3 million books, 225,000 CDs, 60,000 Videos, and other merchandise. The auctioning site eBay has on-line information on more than 3 million items on sale at any time. The emergence of industry-specific exchanges such as Sciquest, Chemdex, Chipcenter, etc. will cause the amount of on-line information about product and services to further explode.

As the range of potential choices expand, the time and effort required to sort through them also expands. These problems are difficult enough when a person is actively searching for a product to meet a specific need. The problem becomes even more severe when people are browsing. The effort required to browse through thousands, if not millions, of product variants within specific categories becomes like searching for the proverbial needle in a haystack [HS99]. The importance and potential commercial impact of managing this data so that users can quickly and flexibly state their preferences represents an important new potential direction for database technology.

We propose a framework for expressing and combining user preferences to address the above problem. Preferences can be used to focus search queries and to order the search results. While the Web applications motivated our work, the framework is more generally applicable.

The salient features of our framework are:

- A user expresses preference for an entity by providing a numeric score between 0 and 1[1], or vetoing it, or explicitly stating indifference. By default, indifference is assumed. Thus, a user states preference for only those entities that the user cares about.
- An entity is described by a set of named fields; each field can take on values from a certain type. The * symbol may be used to match any element of that type. For example, (painting, cubist, *) refers to any cubist painting, (painting, *, Picasso) refers to paintings of Picasso, and (*, *, Picasso) refers to any artwork of Picasso.
- Preferences can be combined. There is one generic combine operation for this purpose. This operator is instantiated by value functions to yield specific instances of the combine operation. Having a single

[1] Extension to the case where the scores are discretized and assigned symbolic labels is straightforward.

generic combine operation makes for a lean and easy to understand and implement system. Allowing value functions provides a great deal of flexibility.

- Specification of preferences is decoupled from how they are combined. The same preferences may be combined in different ways depending upon the application.
- Autonomy of various preferences is preserved. Preference for an entity can be changed without affecting any score of an unrelated entity.
- The combining operation has the closure property so that the result of combining two preferences may be further combined with another preference.

To illustrate the flexibility of our framework, we take a current popular personalization system and a real-life application and show how they can be modeled within our framework. We also sketch how to implement our framework in a relational setting.

Related Work The problem of expressing and combining preferences arises in several applications. Customization by selecting from a menu of choices (e.g. ticker symbols for tracking stocks, city names for weather forecast) can be thought of as a simple expression of preferences. Term expressions used for filtering documents (e.g. myexcite) can also be viewed as simple form of preferences. The recommendation systems based on collaborative filtering [RV97] ask users to rank items and combine preferences of similar users to make recommendations. The need for combining rankings of different models has arisen in meta-search problems [EHJ+96] [FISS98], multi-media systems [Fag98], and information retrieval [SM83]. Perhaps the most famous theorem related to combining preferences is the Arrow Impossibility Theorem in Economics [Arr50]. The theorem says that it is impossible to construct a "social preference function" (ranking the desirability of various social arrangements) out of individual preferences while retaining a particular set of features.

While related, the main thrust of our work is quite orthogonal to the above literature. Our main concern is to develop a flexible framework for expressing and combining preferences that has certain desirable properties. The specific function used in combining some preferences is a parameter in our framework; we only require that this function obey certain constraints. (Arrow's social preference function does not obey these constraints.)

Paper Organization The rest of the paper is structured as follows. In Section 2, we present our preference framework. We formally define preference functions and how they are combined. We introduce modular combining forms that have the desirable properties of efficiency and conservation of the autonomy of various preferences. Modular combining forms are closed under composition. We show that all the preference combing forms defined using our framework are indeed modular.

In Section 3, we model the Personalogic system (http://www.personalogic.com) using our framework. In this case study, we combine several preference functions from the same person that cover different aspects of a total picture. We also model a real-life design application in which a company's preferences are combined with an engineer's preferences into a single preference function. We then present a completeness result that explains the power of our framework.

In Section 4, we sketch how our preference system can be implemented on a relational database system. We conclude with a summary in Section 5.

In this paper, we assume that the user explicitly provides preferences. It is easy to extrapolate how such a system can be used in conjunction with a data mining system that observes a user's past interactions and offers suggestions for preferences.

2 The Framework

2.1 Preference Functions

In this section, we formalize the notion of a preference function.

We start with a set of (base) types which typically include ints, strings, floats, booleans, etc.

We introduce a data type called *score* that represents a user preference. Formally, this is $[0,1] \cup \{ \natural, \perp \}$. A score of 1 indicates the highest level of user preference. A score of 0 indicates the lowest level of user preference. The "$\natural$" score represents a veto. The "$\perp$" score represents that no user preference has been indicated.

We also make use of record types. Since it is an important building block of preference functions, it is worth briefly reviewing. A *record type* is a set of pairs { name_1:type_1, ..., name_n:type_n } in which all n names (a name is simply a non-empty string) are different (although the types are allowed to be the same). In this case, name_i is the name of a field in the record and type_i is the type of that field. A record is where each field takes on a value in the type of that field. More formally, a record is a function r whose domain is { name_1 , ..., name_n } such that r(name_i) is an element of type_i. Usually, r(name_i) is written as r.name_i.

A type is called *wild* iff it contains "*". The "*" symbol is used to indicate a wild card that "matches" any value.

Definition 2.1 *A* preference function *is a function that maps records of a given record type to a score. If p* *is a preference function, we use $dom(p)$ to refer to the given record type that is the domain of p.*

Since we sometimes wish to apply a preference function to a record with more fields than are present in the domain of the preference function, we introduce a projection operator to eliminate the extra fields. This is formalized in Definition 2.2.

Definition 2.2 *Let rt be the record type $\{n_1 : t_1, \ldots, n_k : t_k\}$. Let r be a record of type $\{n_1 : t_1, \ldots, n_k : t_k, n_{k+1} : t_{k+1}, \ldots n_l : t_l\}$ where $k \leq l$. Then $\pi_{rt}(r)$ is the record of type rt where $\pi_{rt}(r).n_i = r.n_i$ for $i \leq k$.*

While a preference function does not require that the types of fields in the record type that is the domain of a preference function be wild, most of the time these fields will be wild so as to allow the user a convenient method for specifying a whole class of preferences.

Definition 2.3 *Given two records r_1, r_2 of type $rt = \{n_1 : t_1, \ldots, n_k : t_k\}$, we say r_2* generalizes *r_1 (which is written as $r_2 \rhd r_1$) iff for all $i \leq k$, either $r_2.n_i = r_1.n_i$ or $r_2.n_i = *$.*

It is clear that the $\rhd$ relation is reflexive and transitive. Note that for any record r there are 2^j records that generalize r where j is the number of fields that have a wild type and for which the value of the field in r is not "*"

2.2 Combining Preference Functions

It is frequently desirable to combine preference functions to form a new preference function. We define a preference function meta-combining form called *combine* which takes a "value function" that says how to compute a new score based on the original scores and produces a preference function combining form (which takes a finite list of preference functions and produces the new preference function). Imagine two roommates, Alice (who never cooks but likes to decorate) and Betty (who does all the cooking) are purchasing a refrigerator. Alice has a preference function (called A_0) whose domain is $\{model : int, color : string \cup \{*\}\}$; Betty has a preference function (called B_0) whose domain is $\{model : int, quality : int\}$. The *model* field indicates the model number, the *color* field is a string describing the color, and *quality* is an integer between 1 and 4 indicating the quality of the refrigerator. Notice that *color* is the only field with a wild type. The two roommates agree that the combined preference function should be what Betty wants (since she does all the cooking) but that Alice should have veto power over any refrigerator they buy. In this subsection, we define a preference function combining form and show how this roommate example can be expressed using this preference function combining form.

We assume the existence of a special character "!" that is reserved for system use and is not allowed to appear in the name of any (user) record field.

Definition 2.4 *Given a record type, $rt = \{n_1 : t_1, \ldots, n_k : t_k\}$, define $ScoreBoard(rt) = \{n'_1! \ldots !n'_k : score | (t_i \text{ is wild} \wedge n'_i = \text{"star!"}) \vee n'_i = n_i\}$.*

Notice that we are using the special character "!" both as a separator character as well as at the end of the "star!" string to avoid conflict with the name of a user field (which is not allowed to contain the "!" character). Thus, $ScoreBoard(rt)$ has 2^j fields where j is the number of t_i types that are wild. The careful reader will note that the record type is a set which is unordered whereas the new field names have an order (namely n'_1 occurs before n'_2, etc.). This gap is easy to remedy by simply taking the names in the new record field to be listed in alphabetical order. In the roommate example, $ScoreBoard(dom(A_0)) = \{color!model : score, star!!model : score\}$ and $ScoreBoard(dom(B_0)) = \{model!quality : score\}$.

Definition 2.5 *Given a record type $rt = \{n_1 : t_1, \ldots, n_k : t_k\}$
and the name of a field $n'_1! \ldots !n'_k$ in record type $ScoreBoard(rt)$,
and a record r of type rt,
define $RecordOf_{rt}(r, n'_1! \ldots !n'_k)$ to be a record of type rt such that, for each field name n_j in rt,*

$$RecordOf_{rt}(r, n'_1! \ldots !n'_k).n_j = \begin{cases} r.n_j & \text{if } n'_j = n_j \\ * & \text{if } n'_j = star! \end{cases}$$

Notice that $RecordOf_{rt}$ completely specifies the record since a value is supplied for every field in the record of type rt. Also note that n'_j can be "star!" only if t_j is wild so that the n_j field in the record $RecordOf_{rt}$ is always a valid member of type t_j. For example, in the roommate example, $RecordOf_{dom(A_0)}(r, star!!model) = \{color = *, model = r.model\}$. (Note that we use the $=$ sign (rather than :) when giving a specific instance of a record.) It is clear that, when applied to a record of type rt, $RecordOf_{rt}$ produces a generalization of that record.

Definition 2.6 *Given a preference function p whose domain is the record type $rt = \{n_1 : t_1, \ldots, n_k : t_k\}$,
and a record r of type rt,
define $Scores(p, r)$ to be a record of type $ScoreBoard(rt)$ such that $Scores(p, r).n'_1! \ldots !n'_k =$
$p(RecordOf_{rt}(r, n'_1! \ldots !n'_k))$.*

The basic idea is that $Scores(p, r)$ provides the value of $p(r')$ for all the generalizations r' of r when the type of r is $dom(p)$. Clearly, $Scores(p, r) : ScoreBoard(dom(p))$ provided r is of type $dom(p)$. In the roommate example, $Scores(A_0, r) =$

$\{star!!model = A_0(\{color = *, model = r.model\}),$
$color!model = A_0(\{color = r.color, model = r.model\})\}.$
In the case that $n'_i = n_i$ for each i (i.e., none of the n'_i are "star!"), note that
$Scores(A_0, r).n_1!\ldots!n_k = A_0(r)$.

Definition 2.7 *A finite set of record types $\{rt_1, \ldots, rt_n\}$ is* compatible *iff whenever rt_i and rt_j share a field with the same name, then those two fields have the same type.*

If $\{rt_1, \ldots, rt_n\}$ is compatible, define $merge(rt_1, \ldots, rt_n)$ to be the record type that has a field $n : t$ iff at least one of the record types has the field $n : t$.

Note that, by compatibility, each field in the merged record type will have a uniquely determined type. In other words, $merge(rt_1, \ldots, rt_n)$ is the "smallest" record type that "contains" all of the record types $rt_1, \ldots, rt_n$. Also note that the order of the arguments to *merge* is irrelevant. In the roommate example, $dom(A_0)$ and $dom(B_0)$ are compatible types and $merge(dom(A_0), dom(B_0)) = \{model : int, color : string \cup \{*\}, quality : int\}$.

Definition 2.8 *For every set $\{rt_1, \ldots, rt_n\}$ of compatible record types, C is called a* preference combining form *based on $(rt_1, \ldots, rt_n)$ iff C maps n preference functions $p_1, \ldots, p_n$ with $dom(p_i) = rt_i$ into a new preference function with domain $merge(rt_1, \ldots, rt_n)$. This new preference function is denoted by $C(p_1, \ldots, p_n)$.*

Definition 2.9 *Let $\{rt_1, \ldots, rt_n\}$ be a set of compatible record types. A function f is called a* value function *based on $(rt_1, \ldots, rt_n)$ iff $f : ScoreBoard(rt_1) \times \ldots \times ScoreBoard(rt_n) \times merge(rt_1, \ldots, rt_n) \rightarrow score$.*

We are now ready to define the meta-combining form *combine* which is at the heart of the combining preference functions.

Definition 2.10 *Let $\{rt_1, \ldots, rt_n\}$ be a set of compatible types. Let f be a value function based on $(rt_1, \ldots, rt_n)$. Then $combine(f)$ is preference combining form based on $(rt_1, \ldots, rt_n)$ defined by*
$combine(f)(p_1, \ldots, p_n)(r)$
$= f(Scores(p_1, \pi_{rt_1}(r)), \ldots, Scores(p_n, \pi_{rt_n}(r)), r)$
for all records r of type $merge(rt_1, \ldots, rt_n)$ and all preference functions with $dom(p_i) = rt_i$ for all $i \leq n$.

The idea behind this notion of combining preference functions is that only the "relevant" scores are examined. The relevant scores are the scores associated with a record as well as any generalizations of that record. All the other values are irrelevant.

Notice that a value function is based on a list (rather than a set) of record types since the order of the arguments to a value function might make a difference.

In the roommate example, the computation that gives Alice veto power would be the function FirstVeto defined as follows:

```
FirstVeto(a : ScoreBoard(dom($A_0$)) ,
        b : ScoreBoard(dom$B_0$),
        c : merge(dom($A_0$),dom($B_0$))) returns score
{
        if a.color!model = ♮ then return ♮;
        else if a.star!!model = ♮ then return ♮;
        else return b.model!quality;
}
```

The preference function $combine(FirstVeto)(A_0, B_0)$ would be the desired combined preference function. If Alice can't stand a particular refrigerator, then she would veto it and the result would be a veto. If Alice chooses not to veto a particular refrigerator, then Betty's preference would be the one that is returned.

For example, assume Alice hated all the model 123 refrigerators and the green refrigerator in model 234; if she had no preference among the other refrigerators, then her preference function A_0 would be:

$A_0(\{color = *, model = 123\}) = \natural$
$A_0(\{color = green, model = 234\}) = \natural$
$A_0(r) = \perp$ for all other records r.

Let's look at a particular example in which Alice's veto prevails. Let $r_1 = \{color = purple, model = 123, quality = 2\}$.

$$\begin{aligned} & combine(FirstVeto)(A_0, B_0)(r_1) \\ = \; & FirstVeto(Scores(A_0, r_1), Scores(B_0, r_1), r_1) \\ = \; & FirstVeto(\{color!model = \perp, star!!model = \natural\}, \\ & \qquad \{model!quality = B_0(r_1)\}, r_1) \\ = \; & \natural \end{aligned}$$

Let's look at a particular example in which Betty's preference prevails. Let $r_2 = \{color = purple, model = 234, quality = 2\}$.

$$\begin{aligned} & combine(FirstVeto)(A_0, B_0)(r_2) \\ = \; & FirstVeto(Scores(A_0, r_2), Scores(B_0, r_2), r_2) \\ = \; & FirstVeto(\{color!model = \perp, star!!model = \perp\}, \\ & \qquad \{model!quality = B_0(r_2)\}, r_2) \\ = \; & B_0(r_2) \end{aligned}$$

In the roommate example, note that the final argument to *FirstVeto* is ignored. A variation on the example would be that Alice can not veto a refrigerator with a quality rating of at least 3. In this case, the definition of *FirstVeto* would change to:

```
FirstVetoSometimes(a : ScoreBoard(dom($A_0$)) ,
        b : ScoreBoard(dom$B_0$),
        c : merge(dom($A_0$),dom($B_0$))) returns score
{
        if c.quality $\geq$ 3 then return b.model!quality;
        else if a.color!model = ♮ then return ♮;
        else if a.star!!model = ♮ then return ♮;
        else return b.model!quality;
}
```

2.3 Modular Combining Forms

In this section, we formalize and study the notion of the modular combining forms (see Definition 2.12). Having modular combining forms has two desirable results. The first desirable result is that autonomy of various preferences is conserved. If a preference function is created using only modular combining forms, then a user may change a preference for a particular record (in one of the original preference functions) without affecting any preference (in the final preference function) for any unrelated records. For example, if Alice changes her perference towards yellow refrigerators in model 123, this will have no affect on the final preference for green refrigerators in model 123. The second desirable result is that an implementation need only provide first order value functions. The value functions do not need to take entire preference functions as arguments. Instead, they only require the finite amount of information that is contained in a scoreboard.

Next we say when two preference functions are equivalent with respect to a record in their domain. The idea is that they agree on all the information that is relevant to a record.

Definition 2.11 *Let p and p' be preference functions with the same domain. Let r be a record of type $dom(p)$. We say p and p' are* equivalent with respect to r *iff $p(\overline{r}) = p'(\overline{r})$ for all $\overline{r} \rhd r$*

We are now ready to define the modular combining forms.

Definition 2.12 *A combining form C based on $(rt_1, \ldots, rt_n)$ is* modular *iff $C(p_1, \ldots, p_n)(r) = C(p'_1, \ldots, p'_n)(r)$ provided that for all $i \leq n$, p_i and p'_i are equivalent with respect to $\pi_{rt_i}(r)$.*

It is clear this definition captures the desired notion of relevance. If a user changes their preference on a given record r (thereby changin their preference function from p to p'), it is clear that p and p' are equivalent with respect to any record which is not generalized by r. The result of the new combined preference function will agree with the old combined preference function on all the records that are not generalized by r.

Modular combining forms enjoy the property of being closed under composition. This is formalized in Proposition 2.13 which, to enhance readability, is stated only for binary combining forms.

Proposition 2.13
Let $\{rt_1, rt_2, rt_3, rt_4\}$ be a compatible set of record types. Let C_1, C_2, and C_3 be modular preference combining forms based on (rt_1, rt_2), (rt_3, rt_4), and $(merge(rt_1, rt_2), merge(rt_3, rt_4))$ respectively. Then the combining form C_0 based on (rt_1, rt_2, rt_3, rt_4) defined by $C_0(p_1, p_2, p_3, p_4) = C_3(C_1(p_1, p_2), C_2(p_3, p_4))$ is a modular combining form.

Proof: This is proven as Proposition 6.1 in Appendix A (Section 6). □

Now that we have seen that modular combining forms are desirable to have around, Theorem 2.14 is important because it guarentees that all the preference combining forms defined using *combine* are modular.

Theorem 2.14
If f is a value function based on $(rt_1, \ldots, rt_n)$, then $combine(f)$ is a modular combining form based on $(rt_1, \ldots, rt_n)$.

Proof: This is proven as Theorem 6.2 in Appendix A (Section 6). □

2.4 Preference System

Definition 2.15 *A* basic preference system *is a collection of record types, and for each record type, a collection of preference functions with that domain (called the* available *preference functions), and a collection of value functions (called the* available *value functions) each of which is based on a finite collection of those record types.*

A basic preference system is closed *iff $combine(f)(p_1, \ldots, p_n)$ is an available preference function provided $p_1, \ldots, p_n$ are available preference functions and f is an available value function that is based on $\{dom(p_1), \ldots, dom(p_n)\}$.*

It is important to note that a basic preference system need not make available all possible preference functions. We expect that it will be the case that most basic preference systems will be closed but to increase flexibility, we do not require this. For example, a system designer might put a sematic condition that a score for any record be within ten percent of the score of any of its generalizations. Since this might be a difficult condition to enforce within the value functions, all the possible value functions might be available even though the combined preference function might not be available. This system would, therefore, not be closed.

3 Flexibility

In this section, we discuss how flexible basic preference systems are. We first model a single person system in which several preference functions from the same person that cover different aspects of a total picture are combined. We use the popular Personalogic system for this purpose. We then consider a multiple person system in which preferences of two (or more) individuals are combined into a single preference function. We have already seen an instance of this in the roommate example; we now model a real-life design application. Finally, we present a completeness results that explains the power of our framework.

3.1 Modeling Personalogic

The Personalogic system[2] is a popular system for making selections and ordering results based on user provided preferences. We sketch below how the functionality provided by Personalogic can be realized as a special case of our framework. We will use Personalogic's decision guide for selecting a dog for illustration.

The dog decision guide allows users to express preferences for various attributes of different breeds of dogs through a series of questions. These attributes include size, indoor energy, exercise time, trainability, barking, history of inflicting injuries, dog group, and coat characteristics such as length, shedding, and hypoallerginicity. The user can also specify the importance of indoor energy, exercise time, trainability, and barking. The values not selected for some of the attributes (size, dog group, coat length) act as vetoes. For other attributes, the user may indicate preference or no opinion. For the history of inflicting injuries and hypoallerginicity attributes, the user may also specify must not have values. The system computes a combined score for each dog in the database based on the weightings of all the individual preferences, provided that all the predicates are satisfied, and returns results ordered by score.

The reader can immediately note that this system is easy to model in our basic preference system. The choices on individual attributes would be a preference function on that attribute. A predicate could be treated as a veto if the predicate is not satisfied. The user controlled weighting could be modeled as a value function.

The reader can also note that a system built on the basic preference system would provide more flexibility in allowing users to express preferences. For instance, the user does not have the option of specifying preference values for a combination of attributes in the Personalogic system. This can be important for a user who wants to veto a combination of some specific values for different attributes while admitting those values in other combinations.

3.2 Design Application

Design houses typically have component engineering departments that are responsible for approving and rating parts that are allowed to be used by design engineers in the company products. Within the guidelines provided by the component engineering, design engineers have considerable flexibility in exercising their preferences. We illustrate below how to model this common situation in our basic preference system so that searches over part databases become cognizant of individual preferences.

A design house deals with three major product categories: inductors, capacitors, and resistors; these are represented by a field called "Product". For each of these categories, there are further subcategories; these are represented by a field called "Subcatory". Manufacturers X, Y, and Z supply all the three categories; these are represented by a field call "Manufacturer". The component engineering has forbidden the use of all parts from Z. It has rated inductors from X as superior (score $= 0.8$) and capacitors as good (score $= 0.6$). On the other hand, the ratings for inductors and capacitors from Y are good and superior respectively. Component engineering has not yet rated resistors. To save writing, we present the records as a (Manufacturer,Product,Subcategory) list. The component engineering expresses these preferences as follows:

$C_0(Z,*,*) = \natural$
$C_0(X,\text{inductors},*) = 0.8$
$C_0(X,\text{capacitors},*) = 0.6$
$C_0(Y,\text{inductors},*) = 0.6$
$C_0(Y,\text{capacitors},*) = 0.8$

Engineer Elizabeth generally likes products from Y better than products from X, except that she really likes ceramic resistors built by X. She also thinks highly of resistors made by Z. She expresses her preferences as follows:

$E_0(Y,*,*) = 0.8$
$E_0(X,*,*) = 0.7$
$E_0(X,\text{capacitors},\text{ceramic}) = 1.0$
$E_0(Z,\text{resistors},*) = 0.9$

By providing different combining forms, it is possible to implement different policies that affect search results in different ways. Note that neither component engineering nor Elizabeth has to restate any of their preferences. Example of policies include:

- *Component engineering has priority.* Elizabeth's searches for inductors will resolve in favor of X, searches for capacitors will resolve in favor of Y, and searches for resistors will resolve in favor of Y. The interesting case is the resistor case. Elizabeth's preference for Z for resistor is vetoed because of blanket ban on Z. However, since component engineering has no preference between X and Y for resistors, Elizabeth's general preference for Y over X prevails.

- *Engineer's preferences have priority unless vetoed by component engineering.* All of Elizabeth's searches now resolve in favor of Y since she prefers Y over X, except for ceramic capacitors for which she has explicit higher preference for X. Her preference for Z for resistors has been vetoed by the component engineering's veto on Z.

[2]http://www.personalogic.com. Personalogic is now owned by America Online (AOL) and their customers include National Geographic, American Express, and E-Trade.

As the time goes by, Elizabeth was able to convince component engineering to loosen its ban on Z for resistors. However, component engineering still rates resistors from Z below than those from X. It can simply add the following preferences:

$C_0(X, \text{resistors}, *) = 0.8$
$C_0(Z, \text{resistors}, *) = 0.6$

No change is required in the combining forms. The reader can easily verify that these additions do not affect the results of searches for inductors or capacitors. Elizabeth's search for resistors now resolve in favor of X under the first policy and in favor of Z under the second.

3.3 Completeness of Combine Operator

We have seen the tremendous flexibility of a basic preference system. In fact, there is good reason for this. The *combine* meta-combining form (Definition 2.10) is complete in that all modular combining forms are definable using *combine*!!!

This is formalized in Theorem 3.1.

Theorem 3.1
Let C be a modular combing form based on $(rt_1, \ldots, rt_n)$. Then there is a value function f based on $(rt_1, \ldots, rt_n)$ such that $C = combine(f)$

Proof: This is proven as Theorem 6.3 in Appendix A (Section 6). □

Putting this result together with the fact that *combine*(f) is always modular gives us the following complete characterization of the modular combining forms.

Theorem 3.2 *C is a modular combining form based on $(rt_1, \ldots, rt_n)$ iff there exists a value function f based on $(rt_1, \ldots, rt_n)$ such that $C = combine(f)$.*

Proof: Follows from Theorems 2.14 and 3.1. □

4 Implementation on a Relational Database System

Let us consider the roommate example and see how it might be represented using a relational database system. The purpose of this example is to show how a relational database system could be used to implement a basic preference system. There are many other possible ways to implement a basic preference system and we think there is a good bit of interesting research to be done to take full advantage of a database system.

In one such implementation, Alice and Betty's preference functions can be stored in separate tables. Records with score of $\perp$ are not represented.

Alice's preference function:

Color	Model	Score
Red	123	0.4
*	123	♮
Green	234	♮
White	456	0.8
White	234	0.6

Betty's preference function:

Model	Quality	Score
123	3	0.7
123	4	0.9
234	4	0.5
345	3	0.3
345	4	0.5

Here is how a system would compute the combined preference function defined by *combine*(*FirstVeto*)(A_0, B_0). Given a record r, the system would perform the following steps:

1. Form the ScoreBoard called sb_a for A_0 as follows:
   (a) Issue the query: SELECT Score FROM Alice WHERE Color = r.color AND Model = r.model
   Store the unique answer in sb_a.color!model (and let this value be $\perp$ if the query returns an empty answer).
   (b) Issue the query: SELECT Score FROM Alice WHERE Color = * AND Model = r.model
   Store the unique answer in sb_a.star!!model (and let this value be $\perp$ if the query returns an empty answer).
2. Form the ScoreBoard called sb_b for B_0 as follows:
   (a) Issue the query: SELECT Score FROM Betty WHERE Model = r.model AND Quality = r.quality
   Store the unique answer in sb_b.model!quality (and let this value be $\perp$ if the query returns an empty answer).
3. Return the value obtained from the user-defined function *FirstVeto*(sb_a, sb_b, r)

The implementor might choose to materialize this new preference function for retrieval efficiency. In this example, we assume that there are four colors: Red, Green, White, and Purple; we assume four models: 123, 234, 345, and 456; and we assume that there are four quality levels: 1, 2, 3, and 4. Under these assumptions, the combined preference function would look like the following:

Color	Model	Quality	Score
Red	123	1	$\natural$
Green	123	1	$\natural$
Purple	123	1	$\natural$
White	123	1	$\natural$
*	123	1	$\natural$
Red	123	2	$\natural$
Green	123	2	$\natural$
Purple	123	2	$\natural$
White	123	2	$\natural$
*	123	2	$\natural$
Red	123	3	$\natural$
Green	123	3	$\natural$
Purple	123	3	$\natural$
White	123	3	$\natural$
*	123	3	$\natural$
Red	123	4	$\natural$
Green	123	4	$\natural$
Purple	123	4	$\natural$
White	123	4	$\natural$
*	123	4	$\natural$

Color	Model	Quality	Score
Red	234	4	0.5
Green	234	1	$\natural$
Green	234	2	$\natural$
Green	234	3	$\natural$
Green	234	4	$\natural$
Purple	234	4	0.5
White	234	4	0.5
*	234	4	0.5
Red	345	3	0.3
Green	345	3	0.3
Purple	345	3	0.3
White	345	3	0.3
*	345	3	0.3
Red	345	4	0.5
Green	345	4	0.5
Purple	345	4	0.5
White	345	4	0.5
*	345	4	0.5

Of course, there is a lot of redundancy in the above table. There are many interesting research questions that merit further investigation such as: when to materialize; how to have more compact representations of the preference functions; what restrictions to put on the available preference functions and available value functions to permit an efficient implementation; etc.

5 Summary

We have presented a framework for expressing and combining user preferences. The system is very lean in that it only has two basic notions:

1. A preference function (Definition 2.1) that specifies user preferences.
2. A single meta combining form *combine* (Definition 2.10) that is based on value functions (Definition 2.9).

Yet, in spite of its very lean nature, the framework is very powerful. The single combine meta-function is able (in conjunction with the value functions) to express all modular preference combining forms (Theorem 3.1).

In addition to being quite powerful, the basic preference system is quite flexible since it does not require the system to provide every possible preference function or every possible value function. Limits might be placed to facilitate user interaction, impose semantic conditions, or enable an efficient implementation. Furthermore, there is flexibility in that the system does not arbitrarily limit the possible value functions.

Future Work Since this paper presents a framework, there is a lot of work that can be done realizing this framework. There is considerable room for system implementors to address efficiency issues and experiment with user interfaces. In fact, a generic user interface could be built for a basic preference system that would work with any preference system. Different representations of preference functions are possible. Another important issue concerns value functions. We expect the system to have a library of canned value functions that should meet the needs of a large number of users. But should value functions be definable by end users and what would be a good interface?

References

[Arr50] K.J. Arrow. A difficulty in the concept of social welfare. *J. of Political Economy*, 58:328–346, 1950.

[BP96] John M. Berrie and David E. Presti. The word wide web as an instructional tool. *Science*, 274:371–372, 1996.

[EHJ+96] O. Etzioni, S. Hanks, T. Jiang, R.M. Karp, O. Madani, and O. Waarts. Efficient information gathering on the internet. In *37th Annual Symp. Foundations of Computer Science*, 1996.

[Fag98] Ronald Fagin. Fuzzy queries in multimedia database systems. In *17th ACM Symp. Principles of Database Systems*, June 1998.

[FISS98] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. In *Machine Learning: 15th Int. Conf*, 1998.

[HS99] John Hagel and Marc Singer. *Net Worth.* Harvard Business School Press, 1999.

[LG99] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[RV97] P. Resnick and H. Varian. Recommender systems. *Communications of the ACM*, 40(3), 1997.

[SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

6 Appendix A: Modular Combining Form Proofs

Proposition 6.1 (Restatement of Proposition 2.13) *Let $\{rt_1, rt_2, rt_3, rt_4\}$ be a compatible set of record types. Let C_1, C_2, and C_3 be modular preference combining forms based on (rt_1, rt_2), (rt_3, rt_4), and $(merge(rt_1, rt_2), merge(rt_3, rt_4))$ respectively. Then the combining form C_0 based on (rt_1, rt_2, rt_3, rt_4) defined by $C_0(p_1, p_2, p_3, p_4) = C_3(C_1(p_1, p_2), C_2(p_3, p_4))$ is a modular combining form.*

Proof: First note that it is clear that C_0 is based on (rt_1, rt_2, rt_3, rt_4). Let r be a record of type $merge(rt_1, rt_2, rt_3, rt_4)$. Assume that p_i and p'_i are equivalent with respect to $\pi_{rt_i}(r)$ for $i = 1, 2, 3, 4$ with the goal of showing that $C_0(p_1, p_2, p_3, p_4)(r) = C_0(p'_1, p'_2, p'_3, p'_4)(r)$.

First we show that $C_1(p_1, p_2)$ and $C_1(p'_1, p'_2)$ are equivalent with respect to $\pi_{merge(rt_1,rt_2)}(r)$. Let $r' \rhd \pi_{merge(rt_1,rt_2)}(r)$. Let $r'' \rhd r'$. Then $\pi_{rt_1}(r'') \rhd \pi_{rt_1}(r') \rhd \pi_{rt_1}(\pi_{merge(rt_1,rt_2)}(r)) = \pi_{rt_1}(r)$. Thus, $p_1(\pi_{rt_1}(r'')) = p'_1(\pi_{rt_1}(r''))$ since p_1 and p'_1 are equivalent with respect to $\pi_{rt_1}(r)$. Thus, p_1 and p'_1 are equivalent with respect to r'. Similarly, p_2 and p'_2 are equivalent with respect to r'. Since, C_1 is modular, it follows that $C_1(p_1, p_2)(r') = C_1(p'_1, p'_2)(r')$. This proves that $C_1(p_1, p_2)$ and $C_1(p'_1, p'_2)$ are equivalent with respect to $\pi_{merge(rt_1,rt_2)}(r)$. Similarly, $C_2(p_3, p_4)$ and $C_2(p'_3, p'_4)$ are equivalent with respect to $\pi_{merge(rt_3,rt_4)}(r)$. Since C_3 is a modular combining form based on $(merge(rt_1, rt_2), merge(rt_3, rt_4))$, it follows that $C_3(C_1(p_1, p_2), C_2(p_3, p_4))(r) = C_3(C_1(p'_1, p'_2), C_2(p'_3, p'_4))(r)$. This proves that $C_0(p_1, p_2, p_3, p_4)(r) = C_0(p'_1, p'_2, p'_3, p'_4)(r)$ as desired. □

Next we show that every combining form defined using the *combine* operator is modular. This is formalized in Theorem 2.14.

Theorem 6.2 (Restatement of Theorem 2.14) *If f is a value function based on $(rt_1, \ldots, rt_n)$, then $combine(f)$ is a modular combining form based on $(rt_1, \ldots, rt_n)$.*

Proof: Let $p_1, \ldots, p_k$ and $p'_1, \ldots, p'_n$ be preference functions such that $dom(p_i) = rt_i = dom(p'_i)$ for all $i \leq n$. Let r be a record of type $RT = merge(rt_1, \ldots, rt_n)$. Let f be a value function based on $(rt_1, \ldots, rt_n)$. Assume that p_i and p'_i are equivalent with respect to rt_i for all $i \leq n$. The goal is to show that $combine(f)(p_1, \ldots, p_k)(r) = combine(f)(p'_1, \ldots, p'_k)(r)$.

Let n_0 be an arbitrary name of a field in $ScoreBoard(rt_i)$.

$$\begin{aligned} & Scores(p_i, \pi_{rt_i}(r)).n_0 \\ = \; & p_i(RecordOf_{rt_i}(\pi_{rt_i}(r), n_0)) \\ = \; & p'_i(RecordOf_{rt_i}(\pi_{rt_i}(r), n_0)) \\ & \text{since } RecordOf_{rt_i}(\pi_{rt_i}(r), n_0)) \rhd \pi_{rt_i}(r) \\ = \; & Scores(p'_i, \pi_{rt_i}(r)).n_0 \end{aligned}$$

Since the choice of n_0 was an arbitrary name in $ScoreBoard(rt_i)$, it follows that $Scores(p_i, \pi_{rt_i}(r)) = Scores(p'_i, \pi_{rt_i}(r))$ for all $i \leq n$. We can compute as follows:

$$\begin{aligned} & combine(f)(p_1, \ldots, p_k)(r) \\ = \; & f(Scores(p_1, \pi_{rt_1}(r)), \ldots, Scores(p_k, \pi_{rt_k}(r)), r) \\ = \; & f(Scores(p'_1, \pi_{rt_1}(r)), \ldots, Scores(p'_k, \pi_{rt_k}(r)), r) \\ = \; & combine(f)(p'_1, \ldots, p'_k)(r) \end{aligned}$$

as desired. □

Now that we know that every combining form $combine(f)$ is modular, the next question to address is are there any other modular combining forms other than the ones definable by *combine*. It turns out the answer is no. This means that *every* modular combining form can be expressed using the *combine* operator. This is formalized in Theorem 3.1.

Theorem 6.3 (Restatement of Theorem 3.1) *Let C be a modular combing form based on $(rt_1, \ldots, rt_n)$. Then there is a value function f based on $(rt_1, \ldots, rt_n)$ such that $C = combine(f)$.*

Proof: It is helpful to have a function *NameOf* that takes a record of type rt for any type rt with field names $n_1, \ldots, n_k$ and produces a name of a field in $ScoreBoard(rt)$ as follows: $NameOf(r_0) = n'_1! \ldots !n'_k$ where $n'_i = n_i$ if $r_0.n_i \neq *$ and $n'_i = star!$ if $r_0.n_i = *$. It is clear that if $r_0 \rhd r$, then $RecordOf_{rt_i}(r, NameOf(r_0)) = r_0$.

First defined a set valued function

$$\begin{aligned} S : & \; ScoreBoard(rt_1) \times \ldots \times ScoreBoard(rt_n) \\ & \times merge(rt_1, \ldots, rt_n) \to \mathcal{P}(score). \end{aligned}$$

Recall that $\mathcal{P}$ represents the power set operation so that $\mathcal{P}(score)$ is the set of all subsets of *score*. The definition of S is as follows: Define $s_0 \in S(sb_1, \ldots, sb_n, r)$ iff there exist preference functions $p_1, \ldots, p_n$ such that $C(p_1, \ldots, p_n)(r) = s_0$ and $\forall i \leq n(dom(p_i) = rt_i \& Scores(p_i, \pi_{rt_i}(r)) = sb_i)$. It is clear that $C(p_1, \ldots, p_n)(r) \in S(Scores(p_1, \pi_{rt_1}(r)), \ldots, Scores(p_n, \pi_{rt_n}(r)), r)$.

Assume that s_0 and s_0' are elements of $S(sb_1, \ldots, sb_n, r)$. Hence, it follows that $s_0 = C(p_1, \ldots, p_n)(r)$, $s_0' = C(p_1', \ldots, p_n')(r)$, and $\forall i \leq n(Scores(p_i, \pi_{rt_i}(r)) = sb_i = Scores(p_i, \pi_{rt_i}(r))$. Let $r_i \rhd \pi_{rt_i}(r)$.

$$\begin{aligned}
& p_i(r_i) \\
= \; & p_i(RecordOf_{rt_i}(\pi_{rt_i}(r), NameOf(r_i)) \\
= \; & Scores(p_i, \pi_{rt_i}(r)).NameOf(r_i) \\
= \; & Scores(p_i', \pi_{rt_i}(r)).NameOf(r_i) \\
& \text{since } Scores(p_i, \pi_{rt_i}(r)) = \\
& \qquad Scores(p_i', \pi_{rt_i}(r)) \\
= \; & p_i'(RecordOf_{rt_i}(\pi_{rt_i}(r), NameOf(r_i)) \\
= \; & p_i'(r_i)
\end{aligned}$$

Hence, it follows that p_i and p_i' are equivalent with respect to $\pi_{rt_i}(r)$ for all $i \leq n$. Since C is a modular combining form, it follows that
$s_0 = C(p_1, \ldots, p_n)(r) = C(p_1', \ldots, p_n')(r) = s_0'$.
This proves that $S(sb_1, \ldots, sb_n, r)$ has at most one element.

If $S(sb_1, \ldots, sb_n, r)$ is empty, then define
$f(sb_1, \ldots, sb_n, r) = \bot$;
otherwise, define $f(sb_1, \ldots, sb_n, r)$ to be the unique member of $S(sb_1, \ldots, sb_n, r)$.
Since $C(p_1, \ldots, p_n)(r) \in$
$S(Scores(p_1, \pi_{rt_i}(r)), \ldots, Scores(p_n, \pi_{rt_i}(r)), r)$,
it follows that $C(p_1, \ldots, p_n)(r)$
$= f(Scores(p_1, \pi_{rt_i}(r)), \ldots, Scores(p_n, \pi_{rt_i}(r)), r)$.
By Definition 2.8, it follows that
$C(p_1, \ldots, p_n)(r) = combine(f)(p_1, \ldots, p_n)(r)$
as desired. □