MADMAX: A DSL for Explanatory Decision Making*

Martin Erwig
Oregon State University
Corvallis, Oregon, USA
erwig@oregonstate.edu

Prashant Kumar
Oregon State University
Corvallis, Oregon, USA
kumarpra@oregonstate.edu

Abstract

MADMAX is a Haskell-embedded DSL for multi-attribute, multi-layered decision making. An important feature of this DSL is the ability to generate explanations of why a computed optimal solution is better than its alternatives.

The functional approach and Haskell's type system support a high-level formulation of decision-making problems, which facilitates a number of innovations, including the gradual evolution and adaptation of problem representations, a more user-friendly form of sensitivity analysis based on problem domain data, and fine-grained control over explanations.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: AHP, decision making, explanations

ACM Reference Format:
Martin Erwig and Prashant Kumar. 2021. MADMAX: A DSL for Explanatory Decision Making. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3486609.3487206

1 Introduction

Multi-attribute decision making (MADM) is an important part of modern decision sciences [22], and the *Analytic Hierarchy Process* (AHP) [16] is a popular MADM method that supports the hierarchical decomposition of decision making.

The theory and methods of MADM have been extensively applied in many areas, ranging from engineering projects, economics, public administration, to management and military projects. For example, in 1986 the Institute of Strategic Studies in Pretoria, a government-backed organization, used AHP to analyze the conflict in South Africa and recommended actions ranging from the release of Nelson Mandela to the removal of apartheid and the granting of full citizenship and equal rights to the black majority [18]. All the recommendations were implemented within a short time. Another high-profile example is the use of AHP in the 1995 US/China conflict over Chinese illegal copying of music, video, and software [19]. An AHP analysis involving four hierarchies for benefits, costs, opportunities, and risks showed, surprisingly, that it was much better for the United States *not* to sanction China. The result of the study predicted what happened. Shortly after the study was complete, the United States awarded China the most-favored nation status and didn't sanction it. In the domain of business, the Xerox Corporation has used AHP to allocate almost a billion dollars to its research projects [17], and IBM used AHP in 1991 in designing its successful mid-range AS 400 computer [20].

Alas, the programming language support for AHP is not commensurate with the important role it plays in decision making in today's world. Most programming languages either provide minimal or no support for AHP. Even the small number of AHP libraries offered by various programming languages require the AHP problems to be encoded in special formats, such as JSON, making them inconvenient to use.

In this paper we address this shortcoming by introducing an embedded Haskell DSL called MADMAX (which stands for Multi-Attribute Decision Making And eXplaining) that facilitates the formulation of AHP problems on a high level, relieving users from the need to normalize data and encode the problem in a rigid form. The high-level problem representation supports the following three specific features.

- *Maintainability.* AHP problems can be dynamically adapted without the need to re-encode data.
- *Understandability.* Results (especially from sensitivity analyses) can be presented in terms of the problem description and not just through their encodings.
- *Explainability.* Solutions can be explained by direct comparison with rivaling alternatives, especially by identifying minimal sets of attributes that have made a difference for the decision.

Haskell's type system has been essential in managing the complexity of the domain and obtaining a succinct DSL design. First, mappings play a pivotal role as a uniform representation of data and are used to represent (1) individual

*This work is partially supported by the National Science Foundation under the grants CCF-1717300 and CCF-2114642.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487206

objects as records (mapping attributes to values), (2) a collection of objects (mapping names to records), (3) normalized data about objects, required to facilitate comparisons in the optimization process, and (4) differences between objects and differences between normalized data. Second, type classes facilitate a uniform naming of similar functionality in different contexts and also allow the overloading of generic functions for selecting and otherwise manipulating data of different types and dimensions. These representations offer not only convenience, but also support the dynamic evolution and adaptation of problem representations. In particular, the maintenance of normalized as well as original data provides a distinctive advantage over existing AHP libraries.

The rest of the paper is structured as follows. We introduce the problem domain of multi-attribute decision making and our basic representation with a simple (non-hierarchical) example in Section 2. We then demonstrate the hierarchical aspect of AHP and show how to work with multiple layers of attributes in Section 3. In Section 4 we illustrate how decisions computed by MADMAX can be explained through identifying a subset of attribute combinations that are most relevant for the decision. In Section 5 we present an evaluation of the explanation component of MADMAX, which shows a significant reduction in the size of explanations, and we also measure the runtime performance. We discuss related work in Section 6 and present conclusions in Section 7.

2 One-Dimensional Decision Making

Imagine you want to buy a new car, deciding between, say, a Honda CRV and a BMW X1. If the comparison is based on only one attribute, such as price or fuel economy, the decision is easy. More realistically, however, the decision must take into account several different attributes that cannot be directly compared with one another. For example, let's assume we want to base our decision on price, fuel economy, and safety rating, which are all measured in different units (money, miles-per-gallon, and a number between 1 and 10, respectively). One way of taking into account attributes from different domains is to normalize the data by mapping attributes into one common domain.

As a first step we can collect the relevant information in a data structure that maps the decision objects, cars, to records of their features and corresponding values. To this end, we define data types for the different features on which to collect information and the cars that we plan to evaluate.

```
data Car     = Honda | BMW | Toyota
data Feature = Price | MPG | Safety
```

Records are mappings from features to their values (type `Rec Feature`), and the information about cars is stored in a mapping of type `Info Car Feature` whose co-domain is given by feature records. The smart constructor `info` (see Figure

```
data Rec    a = Rec  {unRec  :: Map a Double}
data Info o a = Info {unInfo :: Map o (Rec a)}

info :: (Ord o,Ord a) => [(o,[(a,Double)])] -> Info o a
(!)  :: Eq o          => Info o a -> o      -> Rec a
diff :: (Ord o,Ord r) => Info o r -> o -> o -> Rec r
```

Figure 1. Data Representation via maps (based on `Data.Map`).

1)[1] constructs mappings of cars to feature records (the arrow `-->` is simply syntactic sugar for building pairs).

```
carFeatures :: Info Car Feature
carFeatures = info [
  Honda --> [Price --> 34000, MPG --> 30, Safety --> 9.8],
  BMW   --> [Price --> 36000, MPG --> 32, Safety --> 9.1]]
```

This representation already supports some interesting computations. With records defined as instances of `Num`, we can compare objects based on the differences for each attribute by simply using subtraction.

```
> diff carFeatures Honda BMW
{Price -> -2000.00, MPG -> -2.00, Safety -> 0.70}
```

The comparison shows that while the Honda is cheaper and has a slightly better safety rating, it's fuel economy is worse. Does that make the Honda or the BMW a better choice? It depends on how we value the different attributes and the corresponding differences in their values. To facilitate a comparison, we can normalize all attributes by mapping them to values in a common numeric interval[2] such as 0..100 where larger numbers represent better values. This normalization is achieved by the function `valuation`, see Figure 2.

```
> valuation carFeatures
{Honda -> {Price -> 51.43, MPG -> 48.39, Safety -> 51.85},
 BMW   -> {Price -> 48.57, MPG -> 51.61, Safety -> 48.15}}
```

Based on the normalized values we can compute total values for the cars to see how they compare.

```
> total $ valuation carFeatures
{Honda -> 151.67, BMW -> 148.33}
```

Of course, computing total values in this way assumes that each attribute is weighted equally, which is not always a realistic assumption. It is not difficult to implement functions for computing differently weighted averages of an attribute valuation. However, a more systematic approach for doing

[1]We mostly show only the type signatures and present implementations only when they contribute to a better understanding. For the complete code, see https://github.com/LambdaLand/MADMAX.

[2]The AHP method uses the interval [0..1], but for better readability we use the range 0 to 100 and remove the syntactically burdensome "`0.`" prefixes, which also makes 3 or 4 digits of precision more palatable.

so in the context of the AHP methodology is to modify the valuation by multiplying it with a weight vector. We will explain this method in the next section.

The reader may have already noticed the following curiosity about how the attributes have been turned into valuations: Whereas higher MPG values lead to higher valuations, higher prices lead to lower valuations, which is what we want (a high price is bad, a high MPG is good). Since this semantic information cannot be inferred automatically, we have to provide it explicitly, which we do by assigning a so-called *valence* to each attribute indicating whether large numbers actually mean a positive value contribution.

```
instance Valence Feature where
  valence Price = False
  valence _     = True
```

Based on valence, *valuation* replaces each value v_{ij} in a record $R_i = \{A_1 \mapsto v_{i1}, \ldots, A_k \mapsto v_{ik}\}$ that is part of a mapping $M = \{O_1 \mapsto R_1, \ldots, O_n \mapsto R_n\}$ by its normalized value $\overline{v}_{ij}$, which is defined as follows.[3] If attribute A_j has positive valence (A_j^+), then v_{ij} is normalized with respect to the sum of all values $v_{1j}, \ldots, v_{nj}$ used for that attribute. Otherwise (A_j^-), normalization uses the reciprocals of the values.

$$A_j^+ : \overline{v}_{ij} = \frac{v_{ij}}{\sum_{l=1}^{n} v_{lj}} \times 100 \qquad\qquad A_j^- : \overline{v}_{ij} = \frac{1/v_{ij}}{\sum_{l=1}^{n} 1/v_{lj}} \times 100$$

The multiplication by 100 scales the result to a number between 0 and 100. The function `valuation` is a straightforward implementation of this definition.

The normalization step achieves, first and foremost, an unbiased comparison of the attribute differences. Is the price difference of $2,000 more significant than a difference of 2 MPG fuel efficiency? Using `diff` on the normalized records we can infer from comparing the absolute values that the answer is "no" (again assuming equal weight for the attributes).

```
> let vd = diff (valuation carFeatures) Honda BMW
{Price -> 2.86, MPG -> -3.23, Safety -> 3.70}
```

Instead of computing the difference between the two values, we can apply `total` to the value difference directly, which produces the total of the attributes in form of an aggregated collection (a concept we will explain in Section 4).

```
total vd
{Price, MPG, Safety} : 3.34
```

Note that the values for the attributes are based on the current data, which consists of only 2 cars. If we add more cars, the total available value of 100 for each attribute will be split among more cars, and the differences between two specific cars will inevitably become smaller. A more subtle effect is that the relative importance of an attribute can change as well. For example, consider the following addition.

```
threeCars :: Info Car Feature
threeCars = carFeatures `union` info [
  Toyota --> [Price --> 27000, MPG --> 30, Safety --> 9.4]]
```

As expected, valuation yields smaller attribute values.

```
> valuation threeCars
{Honda ->  {Price -> 31.2, MPG -> 32.6, Safety -> 34.6},
 BMW ->    {Price -> 29.5, MPG -> 34.8, Safety -> 32.2},
 Toyota -> {Price -> 39.3, MPG -> 32.6, Safety -> 33.2}}
```

But more importantly, if we again compare the significance of the attributes, we see that the price difference between Honda and BMW, which hasn't changed, has lost in significance relative to the MPG difference.

```
> let vd3 = diff (valuation threeCars) Honda BMW
{Price -> 1.73, MPG -> -2.17, Safety -> 2.47}
```

The price difference was about 90% ($\approx 2.9/3.2$) of the value that MPG contributed when we looked at two cars. Since the overall price range has increased through the addition of the third car while the safety rating range remained the same, the

```
class Ord a => Valence a where
  valence :: a -> Bool
  valence _ = True

class (Bounded a,Enum a,Ord a) => Set a where
  members :: [a]
  members = enumFromTo minBound maxBound

class Aggregate a b | a -> b where
  agg :: ([Double] -> Double) -> a -> b

total :: Aggregate a b => a -> b
total = agg sum

type Val o a = Info o a
data Agg a = Agg [a] Double

valuation :: (Ord o,Set a,Valence a) => Info o a -> Val o a
total     :: Ord a => Val o a -> Rec o
total     :: Rec a -> Agg a
```

Figure 2. Valuations. The function `members` of the `Set` class enumerates all elements of an instance type, which is needed for iterations in several normalization functions. All attribute types must implement `Set`. The `Aggregate` class facilitates relating types based on numeric aggregation functions. Instances for the types `Info` and `Rec` allow the use of the function `total` to map each object to its total value. Another instance is for records and aggregated collections, which allows us to compute the total valuation for an individual record.

[3] We write $\overline{v}$, $\overline{R}$, and $\overline{M}$ for normalized values, records containing normalized values, and mappings that carry normalized records, respectively.

relative value has dropped to about 77% (≈ 1.7/2.2). To see the impact each attribute has on the overall value difference, we can compute what we call their *value difference impact* (*VDI*), and we can observe how the relative impacts change for a comparison of two cars when data is added, even if the data of the compared cars doesn't change.

```
> impact vd
{Price -> 29%, MPG -> 33%, Safety -> 38%}

> impact vd3
{Price -> 27%, MPG -> 34%, Safety -> 39%}
```

Computing the impact of attributes is a special case of focusing in a multi-dimensional comparison on the contributions of attributes from a specific dimension, which we will discuss in the next section.

In general, the relative contribution of different attributes to the overall valuation of individual items helps explain the computed rankings. We will discuss the explanation of comparisons and ranking decisions in Section 4.

3 Multiple Layers of Attributes

The basic form of valuation distributes 100 value points across k alternatives for each of n attributes, that is, the total number of points is $100n$, each attribute will on average have $100/k$ points, and each alternative will have on average $100n/k$ points. This representation views a data set through the lens of comparing n independent attributes. When we want to combine all attributes into one value to determine the best alternative, we can achieve this, as shown in the previous section, by simply extracting the total value with the function `total`, but we can also do this more systematically by combining the three attributes into one.

This operation is a special case of refining an attribute dimension of type `a` by an attribute dimension of another type `b`. In our example we want to refine the feature attributes by weights. To better explain the approach let us assume that we want to entertain different weightings (from which we can later select a specific one). Specifically, in addition to our own personal weighting scheme we also consider how experts weight the attributes. To this end, we first define a data type for the new attribute dimension capturing different opinions about what weighting scheme to use.

```
data Opinion = Personal | Expert
```

We can record the different preferences about how to weight attributes in the same way as the data about car features.

```
featureOpinions :: Info Feature Opinion
featureOpinions = info [
  Price  --> [Personal --> 5, Expert --> 3],
  MPG    --> [Personal --> 3, Expert --> 5],
  Safety --> [Personal --> 2, Expert --> 2]]
```

```
class Covers t a | t -> a where
  project :: t -> a

class Expand t a u | t a -> u where
  expand :: t -> a -> u

instance Covers (a,b) a where project = fst
instance Covers (a,b) b where project = snd
...

instance Expand (a,b) c (a,b,c) where expand (a,b) c = (a,b,c)
...

extendBy :: (Ord o,Ord a,Valence b,Set b,Ord u,
             Expand t b u,Covers t a) =>
             Val o t -> Info a b -> Val o u
only   :: (Eq a,Covers t a) => a -> Info o t -> Info o t
except :: (Eq a,Covers t a) => a -> Info o t -> Info o t
```

Figure 3. Functions for working with multi-dimensional attributes. The class `Covers` relates a tuple type to all its component types whereas the class `Expand t b u` is defined for three types if `u` is the tuple type that results from appending the type `b` to the tuple type `t`.

The weightings can be applied to the feature data by multiplying the two nested mappings. This essentially amounts to a matrix multiplication (which is how it is defined in AHP). To prepare the extension we convert the valuation of car features into one that wraps the feature records in a singleton tuple. This requirement facilitates the overloading of the `extendBy` function for arbitrary tuples, see Figure 3.

```
featureVal :: Val Car (OneTuple Feature)
featureVal = mkOneTuple (valuation carFeatures)

carOpinions :: Val Car (Feature,Opinion)
carOpinions = featureVal `extendBy` featureOpinions
```

The values for the two attributes `Personal` and `Expert` refine the values of the `Feature` attributes. The `Opinion` dimension also acts as a summary of the `Feature` dimension.

```
> carOpinions
{Honda ->
  {(Price,Personal)  -> 25.71, (Price,Expert)  -> 15.43
   (MPG,Personal)    -> 14.52, (MPG,Expert)    -> 24.19,
   (Safety,Personal) -> 10.37, (Safety,Expert) -> 10.37},
 BMW ->
   {(Price,Personal)  -> 24.29, (Price,Expert)  -> 14.57,
    (MPG,Personal)    -> 15.48, (MPG,Expert)    -> 25.81,
    (Safety,Personal) -> 9.63,  (Safety,Expert) -> 9.63}}
```

The valuation `carOpinions` distributes 100 points as before across 2 car alternatives, but now for only 2 attributes, `Personal` and `Expert`, which can be seen when computing the totals, which now sum to 200.

```
> total carOpinions
{Honda -> 100.59, BMW -> 99.41}
```

Extending a valuation $\overline{M} = \{O_1 \mapsto \overline{R}_1, \ldots, O_n \mapsto \overline{R}_n\}$ means to refine each record $\overline{R}_i = \{A_1 \mapsto \overline{v}_{i1}, \ldots, A_k \mapsto \overline{v}_{ik}\}$ according to a mapping $N = \{A_1 \mapsto S_1, \ldots, A_k \mapsto S_k\}$ that carries for each attribute A_j a record S_j, which itself consists of l attributes from a generally different domain, that is, $S_j = \{B_1 \mapsto w_{j1}, \ldots, B_l \mapsto w_{jl}\}$. First, we can normalize N by normalizing each record S_j in the same way as described earlier, which yields $\overline{S}_j = \{B_1 \mapsto \overline{w}_{j1}, \ldots, B_l \mapsto \overline{w}_{jl}\}$.

In a second step the refinement is achieved by multiplying $\overline{M}$ and $\overline{N}$, which, for each record $\overline{R}_i$, combines each A attribute with each B attribute from the corresponding record $\overline{S}_i$ to define a joined value from the two attributes. This multiplication yields the mapping $\overline{L} = \{O_1 \mapsto \overline{Q}_1, \ldots, O_n \mapsto \overline{Q}_n\}$ where each record $\overline{Q}_i$ has the following form

$$\overline{Q}_i = \{(A_1, B_1) \mapsto \overline{u}_{i11}, \ldots, (A_1, B_l) \mapsto \overline{u}_{i1l}, \\ \ldots, \\ (A_k, B_1) \mapsto \overline{u}_{ik1}, \ldots, (A_k, B_l) \mapsto \overline{u}_{ikl}\}$$

and where each value $\overline{u}_{ipq}$ $(1 \le p \le k, 1 \le q \le l)$ is defined as $\overline{u}_{ipq} = \overline{v}_{ip} \times \overline{w}_{pq}/100$.

A comparison of the cars based on these sums implicitly values expert and personal experience equally. If we want to see how the cars compare exclusively under our personal or the expert view, we can project the valuation onto specific attributes in the `Opinion` dimension, which produces either one of the two columns of `carOpinion`. Of course, we can also compute the totals for those projections to see the effect of the different weightings.

```
> total $ only Personal carOpinions
{Honda -> 50.60, BMW -> 49.40}

> total $ only Expert carOpinions
{Honda -> 49.99, BMW -> 50.01}
```

The personal weighting slightly favors the Honda whereas the expert weighting slightly favors the BMW. We can also see that the difference between the two cars under the personal valuation is slightly larger than under the expert valuation, which is consistent with the result of `total carOpinions`.

The functions `only` and its dual `except` are instances of the general function `filter` for extracting subsets of data; they are defined using the function `project` to focus on a particular component of a tuple type.

```
filter :: (t -> Bool) -> Info o t -> Info o t
```

To facilitate projection onto individual components of tuple types, the type of the attribute filtered upon (`a`) is covered by the tuple type of the data (`t`), again see Figure 3.

Finally, we can add one more dimension to our car selection problem to aggregate the different opinions about how

B_1	Price	MPG	Safety
Honda	0.51	0.48	0.52
BMW	0.49	0.52	0.48

B_2	Personal	Experts
Price	0.5	0.3
MPG	0.3	0.5
Safety	0.2	0.2

B_3	Weight
Personal	0.6
Expert	0.4

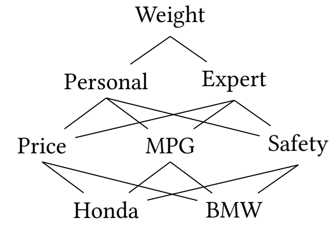

Figure 4. The AHP decision matrices/model for the car selection problem ($B_1 \approx$ *Feature*, $B_2 \approx$ *Opinion*, and $B_3 \approx$ *Weight*).

to weight features. In our example we weight the personal opinion with 60% and the expert opinion with 40%. The extension of the valuation happens similarly to the previous refinement step, except that the new dimension has only one attribute, for which we introduce a smart constructor.

```
weight :: a -> [(Weight,a)]
weight x = [Weighted --> x]

carsW :: Val Car (Feature,Opinion,Weight)
carsW = carOpinions `extendBy` info [
          Personal --> weight 0.6,Expert --> weight 0.4]
```

The total of the valuation yields the final verdict about how the cars' values compare according to the model.

```
> total carsW
{Honda -> 50.36, BMW -> 49.64}
```

MADMAX provides additional functions, for example, to change individual attribute values, to add and remove attributes, or to generate ranked lists of alternatives. The selection of functions shown here illustrates how to work with the DSL to model hierarchical decision problems. In particular, note the high degree of flexibility that the functional implementation brings to the table: We can easily change data on the fly and instantly recompute valuations, differences, ranking, etc. Moreover, we can dynamically extract different slices of the data (cf. `only`) and thus explore and look at the data from different angles through queries without having to change the representation.

Compare this to the AHP approach in which all relevant data is stored in a set of matrices. Figure 4 shows the AHP matrix representation of our car selection example. To change a decision model, these matrices have to be edited, and values have to be renormalized before new rankings can be computed. Specifically, suppose we want to add a third car to be considered in the comparison. We can't simply add another row to matrix B_1: we also have to change all existing entries,

renormalizing them based on the newly added information. Since in our approach valuations are derived from the original data through a simple function, we can regenerate the required normalized data automatically.

```
> valuation threeCars
{Honda ->  {Price -> 31.21, MPG -> 32.61, Safety -> 34.63},
 BMW ->    {Price -> 29.48, MPG -> 34.78, Safety -> 32.16},
 Toyota -> {Price -> 39.31, MPG -> 32.61, Safety -> 33.22}}
```

The hierarchical composition of the different levels is expressed in AHP through a DAG that shows all multiplicative combinations of the involved attributes as they are used to compute an overall value at the top for each of the alternatives at the bottom, see Figure 4. The graph provides a blueprint for the structure of the matrices and is typically designed at the outset of modeling a decision problem. Once we have the normalized values for the various features of the involved alternatives stored in the matrices, the matrix product $B_1 B_2 B_3$ gives the valuations of the alternatives.

The AHP approach is a proven, effective method for solving a fixed decision problem. The realization as an embedded DSL turns this approach into a flexible data decision exploration tool that allows users to dynamically build and adapt decision problems.

4 Explaining Decisions with Layered Dominating Sets

Modeling a decision problem and then computing a solution is the core of decision making. However, it is only one part of the whole process. Once an optimal alternative has been determined, questions often arise as to *why* the chosen alternative is best, especially, why is it better than the second-best alternative? All the AHP method can point to is the overall valuation score as an answer. However, in many cases this is not satisfying, and one often wonders what it is about the first choice that makes it better than the alternative(s).

To address this question we show how to *explain* decisions by identifying attribute sets that work for and against a specific choice and thus contribute to the decision. Specifically, we show how the concept of *minimal dominating sets* [6] can be employed for that purpose and how it can be nicely integrated into the overall structure of MADMAX.

In our example the valuation totals suggest that the Honda is a better choice than the BMW. This decision is the result of the aggregation of all attribute values along the different dimensions. But why exactly? What attributes are responsible for this evaluation? Notice that our attribute structure has become more complicated, since the refinement of the model has added the two dimensions `Opinion` and `Weight` to the original dimension `Feature`. Therefore, we can't simply talk about the value of, say, `Price` or `MPG` anymore. Instead, we need to consider combined value contributions such as (`Price`,`Personal`,`Weighted`). We can certainly print the complete valuation, but that doesn't help much in understanding how the final decision came about.

```
> carsW
{Honda ->
    {(Price, Personal,Weighted) -> 15.43,
     (MPG,   Personal,Weighted) ->  8.71,
     (Safety,Personal,Weighted) ->  6.22,
     (Price, Expert,   Weighted) ->  6.17, ...},
 BMW ->
    {(Price, Personal,Weighted) -> 14.57,
     (MPG,   Personal,Weighted) ->  9.29,
     (Safety,Personal,Weighted) ->  5.78,
     (Price, Expert,   Weighted) ->  5.83, ...}}
```

The first thing we notice is that the dimension `Weight` does not contribute anything to distinguishing between different attributes, because it has only one attribute `Weighted`. Its sole purpose (stemming from the AHP approach) is to consolidate the valuations into a normalized range between 0 and 100. We can simplify the attribute structure and shrink the dimensional space by simply removing the `Weight` dimension while keeping the values. The shrinking of valuations is implemented using an overloaded function `shrink` for removing a component of a tuple type, see Figure 5.

```
> let cars = shrinkVal carsW :: Val Car (Feature,Opinion)
{Honda ->
   {(Price,Personal)  -> 15.43, (Price,Expert)  ->  6.17,
    (MPG,Personal)    ->  8.71, (MPG,Expert)    ->  9.68,
    (Safety,Personal) ->  6.22, (Safety,Expert) ->  4.15},
 BMW ->
   {(Price,Personal)  -> 14.57, (Price,Expert)  ->  5.83,
    (MPG,Personal)    ->  9.29, (MPG,Expert)    -> 10.32,
    (Safety,Personal) ->  5.78, (Safety,Expert) ->  3.85}}
```

This is easier to read, but still not easy to process. To make better sense of all these numbers, we can try to identify those attributes of the preferred alternative whose values are "winning" over the attributes that are in favor of the competitor.

In a first step we compute again the differences of attribute values for the two alternatives to be compared.

```
> let vd = diff cars Honda BMW
{(Price,Personal)  ->  0.86, (Price,Expert)  ->  0.34,
 (MPG,Personal)    -> -0.58, (MPG,Expert)    -> -0.65,
 (Safety,Personal) ->  0.44, (Safety,Expert) ->  0.30}
```

The value differences tell us that fuel efficiency counts against the Honda whereas price and safety count in its favor, but we already knew that. However, we can now look in more detail at which attributes (or attribute combinations) really make a difference in the decision. Specifically, we can ask which sets of attributes with a positive value can overcome the negative impact of the `MPG` attribute. Such a set is called a *dominator* [6]. It turns out that even in our small

```
class Shrink t t' | t -> t' where shrink  :: t -> t'

instance Shrink (OneTuple a) () where shrink _ = ()
instance Shrink (a,b) b          where shrink = snd
instance Shrink (a,b) a          where shrink = fst
instance Shrink (a,b,c) (b,c)    where shrink (a,b,c) = (b,c)
...

class (Covers t a,Shrink t t') => Split t a t' where {}

instance Split (OneTuple a) a ()
instance Split (a,b)   a b
instance Split (a,b)   b a
instance Split (a,b,c) a (b,c)
instance Split (a,b,c) b (a,c)
...
```

Figure 5. Additional functions for manipulating tuple types.

example there are already 5 such sets. When we consider dominators, we are not so much interested in the valuations of the individual attributes as their totals.

```
> map total $ dominators vd
[{(Price,Personal),(Safety,Personal)} : 1.30,
 {(Price,Personal),(Price,Expert),(Safety,Personal)} : 1.64,
 {(Price,Personal),(Price,Expert),(Safety,Expert)} : 1.50,
 {(Price,Personal),(Safety,Personal),(Safety,Expert)} : 1.60,
 {(Price,Personal),(Price,Expert),(Safety,Personal),
  (Safety,Expert)} : 1.94]
```

In this case the function `total` is the instance of the `Aggregate` class for the type `Agg`, see Figure 2.

The simplest reason for why the Honda is preferred over the BMW is given by the dominator that has the fewest elements, called *minimal dominating sets* (or *MDS*). We have only one such MDS in our example.

```
> let honda:_ = mds vd
> total honda
[{(Price,Personal), (Safety,Personal)} : 1.30]
```

A solutions to an AHP decision problem can be effectively explained with an MDS as follows. First, a decision problem is given by a valuation for a multi-dimensional attribute set. The solution to the problem is the entry with the highest total valuation (the "winner"), and an explanation for this decision is the MDS with respect to the value difference between the winner and some other entry. In most cases an explanation would be sought with respect to the closest contestant, or the "runner-up," which is the entry with the second highest total valuation. In our example we have:

```
> winner cars
Honda

> runnerUp cars
BMW
```

```
data Dominance a     = Dominance (Agg a) (Agg a)
data Explanation o a = Explanation o o (Dominance a)

dominators :: Ord a => Rec a -> [Rec a]
mds        :: Ord a => Rec a -> [Rec a]
dominance  :: Ord a => Rec a -> Dominance a

explain    :: (Ord o,Ord a,Ord b,Shrink a b) =>
              Val o a -> Explanation o b
```

Figure 6. Functions for computing dominators and explanations.

We can combine the computation in a function `explain` that determines the two best choices in a valuation and compares them via a critical set of attributes, see also Figure 6.

```
> explain cars :: Explanation Car (Feature,Opinion)
Honda is the best option; it is better than BMW because
{(Price,Personal), (Safety,Personal)}:1.30 >
  |{(MPG,Personal), (MPG,Expert)}:-1.23|
```

The explanation says that the total positive value attributed to the personal opinion about price and safety is enough to overcome the negative value that results from the personal and expert opinion about fuel efficiency. We call the attribute set of the runner-up and its valuation the *barrier*, which is to be overcome by a dominator of the winner. The barrier is the strongest argument in favor of the runner-up, and any dominator is by definition a convincing counter argument.

Since our car example is relatively small, the computation of (minimal) dominating sets might not seem to be such a big help. In more realistic examples where the number of dimensions and attributes is larger, the valuations may grow considerably, and the reduction achieved by an MDS can be significant. We present an estimation of the explanatory simplifications that can be achieved by MDSs in Section 5.

4.1 Attribute-Focused Explanations

In addition to the sheer number of individual attribute values, another potential source of confusion is the multitude of different attribute combinations. Even in our small example the attribute combinations in the MDS are non-symmetric in the sense that the barrier contains one feature but two opinions whereas the MDS has two features but one opinion. We can make the roles of the different attributes and their dimensions more clear in two different ways.

First, we can simply ignore a refinement of one dimension by the other and look at the decision on a more coarse-grained level. For example, we can ask how the decision can be explained in terms of features only, which means to sum all valuations for one feature over different opinions. The focused value difference looks as follows.

```
> focus vd :: Rec Feature
{Price -> 1.20, MPG -> -1.23, Safety -> 0.74}
```

We can also ask directly for an explanation of the decision that is focused just on features.

```
> explain $ focus cars :: Explanation Car Feature
Honda is the best option; it is better than BMW because
{Price, Safety} : 1.94 > |{MPG} : -1.23|
```

As we can see, again, price and safety considerations outweigh fuel efficiency. But since we are focusing on features only, the opinion breakdown is not reported. While the explanation is simpler, it is also less precise, since it does not reveal that only the personal opinion were required for reaching the decision. Similarly, we can focus on opinions only.

```
> explain $ focus cars :: Explanation Car Opinion
Honda is the best option; it is better than BMW because
{Personal} : 0.72 > |{Expert} : -0.01|
```

This explanation is consistent with the computation of total valuations in which we project on personal and expert opinions, but it also shows, partially due to the weighting, that the personal opinions dominate the decision.

We can also analyze the impact of a specific dimension on the decision. The function `factor` computes the impact of the different attributes in one specific dimension on the valuation. The dimension is specified as the first component of a type annotation that splits a multi-dimensional mapping of type $(T_1, \ldots, T_n) \rightarrow T$ into one in which component T_k is lifted, that is, into a mapping of type $T_k \rightarrow (T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n) \rightarrow T$. We represent a factored valuation as a mapping from the type that was factored to the remaining types, see Figure 7. In the following example we apply `factor` to the MDS to isolate the `Feature` type.

```
> factor honda :: Factor Feature Opinion
{Price -> 66% {Personal -> 100%},
 Safety -> 34% {Personal -> 100%}}
```

The factor perspective on the MDS tells us that the price is about 2/3 responsible for selecting the Honda, and the safety rating counts for 1/3 of the decision, where for both of these aspects we only need to take into account the personal judgment, and the expert opinion doesn't really matter. In this case, since we only have 2 dimensions to work with, the focus on `Opinion` gives us the same information in a slightly different form.

```
> factor honda :: Factor Opinion Feature
{Personal -> 100% {Price -> 66%, Safety -> 34%}}
```

We can apply the same scrutiny to the attribute set of the runner-up, which, as we already mentioned, is called the *barrier* of the value difference.

```
data Factor a t  = Factor {unFactor :: Map a (Double,Rec t)}

factor :: (Ord t,Ord a,Ord t',Split t a t') =>
          Rec t -> Factor a t'
```

Figure 7. Functions for factoring valuations.

```
> let bmw = barrier vd
{(MPG,Personal) -> -0.58, (MPG,Expert) -> -0.65}

> factor bmw :: Factor Feature Opinion
{MPG -> 100% {Personal -> 47%, Expert -> 53%}}
```

We observe that `MPG` is the only attribute in favor of the BMW and that its value is the result of 47% of `Personal` opinion and 53% of `Expert` opinion. Again, we can obtain the same information in a different format by switching the focus from `Feature` to `Opinion`.

In summary, focusing and factoring provide a variety of perspectives on the attribute sets that are responsible for a decision. The insights can help a decision maker to get a better sense of how a decision is challenged and supported by the underlying data, which is not only useful for understanding a decision but also for presenting it, answering questions about it, and potentially even defending it.

4.2 Sensitivity Analysis

In addition to explaining a decision, one might also wonder how relevant or decisive a decision is, that is, how much better is the first alternative than the second (or even third)? This often leads to questions of how much specific attributes have to change for the decision to change. In the context of our car-buying scenario we might want to know, for example, by how much the price of BMW needs to be decreased such that it becomes the preferred option. Questions like this are sometimes called *sensitivity analysis queries* [21], which result, in this case, in a mapping of cars to associated changes in their prices that would lead to a different ranking.

A sensitivity analysis query can be expressed as a relationship between an attribute of a specific feature and the attributes of the feature it refines. For the price-change query, the query feature and attribute are `Feature` and `Price`, respectively, and the change feature is `Car`. Query and change features must always be consecutive levels in an AHP.

We capture the result of a sensitivity analysis in a mapping `Change` (see Figure 8). To account for the fact that in some cases no changes in a feature can change the outcome, the result type of the mapping is `Maybe Double`.

The main computation is performed by the auxiliary function `sensitivityNorm`, which takes as its first input (type `a`) a tuple of mappings that represent the AHP problem and defines the data space within which the sensitivity analysis takes place. In our example, this is the following value.

```
data Change a = Change (Map a (Maybe Double))

class (Set q, Set c,Ord c,Covers a (Info c q),Valence b) =>
      Sensitivity a o q c | a o c -> q where
  sensitivityNorm :: a -> (o,o) -> q -> Change c

sensitivity :: (Norm a,Ord b,Sensitivity o a q c,Limit q) =>
               a -> (o,o) -> q -> Change c
```

Figure 8. Functions for sensitivity analysis.

```
carData :: (Info Car Feature, Info Feature Opinion,
            Info Opinion Weight)
carData = (carFeatures,featureOpinions,weights)
```

The class `Norm` contains a function to normalize tuples of `Info` maps into tuples of normalized values. The second argument is a pair of alternative objects (of type `o`) whose ranks are to be swapped. In our example these are `Honda` and `BMW` of type `Car`. The third argument is an attribute of the query feature (of type `q`, in our case `Price` of type `Feature`) with respect to which the sensitivity analysis is to be performed. The class constraint `Limit` allows users to define upper and lower limits on the values of that type to be considered by the analysis to eliminate unreasonable change suggestions (such as a price of $0). Finally, the result type `c` represents the change feature of the sensitivity analysis query.

Since `sensitivityNorm` works with normalized values, we have to de-normalize the computed results and convert them back to the original units provided by the user. Otherwise, the computed results would be difficult to make sense of. This happens in the function `sensitivity`, which again makes use of the whole data space (here `carData`). We can thus answer the question about the price change by running the following sensitivity analysis.[4]

```
> sensitivity carData (Honda,BMW) Price
{Honda -> 1085.34, BMW -> -1124.72, Toyota -> -20479.90}
```

The result tells us different possible changes in the prices of cars that would change relative ranking of Honda over BMW. Specifically, if the price of `Honda` increased by at least 1085.34, or the price of the BMW decreased by at least 1124.72, or the price of Toyota decreased by at least 20479.90, then BMW would rank higher than Honda. Interestingly, a change in the price of Toyota can also swap the rankings of Honda and BMW. This is because the sum of the normalized attribute values for a given attribute should be 100. Changing the price of Toyota leads to re-normalization to enforce this condition, which leads to a change in the normalized values of Honda and BMW that may reverse their ranking.

[4]Note that the MADMAX pretty printer omits `Just` constructors when printing `Maybe` values.

Of course, we can perform the sensitivity analysis with respect to any other attribute of `Feature`. For example, the following query computes the necessary changes in fuel efficiency to change the ranking.

```
> sensitivity carData (Honda,BMW) MPG :: Change Car
{Honda -> -0.92, BMW -> 0.99, Toyota -> Nothing}
```

As expected, a decrease for the Honda and an increase for BMW in that attribute can flip the ordering. The entry for Toyota shows that no change in its fuel efficiency can affect a change in the ordering.

In addition to different attributes, we can perform the sensitivity analysis on other features as well. For example, we can ask "How dependent is the current ranking on the personal opinion?" In this case the input feature is `Opinion` and the output feature is `Feature`. While we are still trying to identify conditions for changing the ranking of the cars, we are now doing this indirectly by changing the value of the `Opinion` feature so that the induced change in `Feature` is enough to change the car ranking.

```
sensitivity carData (Honda,BMW) Personal :: Change Feature
{Price -> -3.98, MPG -> 2.95, Safety -> Nothing}
```

The result says the car ranking changes if the weight assigned to the `Price` attribute is decreased by 3.98 or if the weight assigned to the `MPG` attribute is increased by 2.95, but no weight change to the `Safety` attribute can alter the ranking.

All previous AHP implementation that we know of report the result of any sensitivity analysis in terms of their normalized encodings, which can be hard to interpret. One important contribution of MADMAX is the ability to present the results of sensitivity analyses in terms of the original values specified in the problem, which enhances understandability and makes sensitivity analyses more user friendly.

5 Evaluation

To assess the effectiveness of MDS explanations for AHP decisions, we have performed a number of experiments to estimate the reduction in complexity that they can be expected to deliver. In the following we describe the setup and results of these experiments.

First, we have to establish criteria to measure the efficacy of explanations. Without explanations users need to inspect all n trace components generated by the AHP process. The explanatory strength of an MDS comes from the fact that it can often reduce this number considerably, say, to m trace components. This aspect can be captured by defining the *explanatory ratio* of an MDS as m/n. The smaller the ratio, the fewer components users have to look at (relative to the unexplained original decision), thereby making it easier to understand the decision. A related aspect that seems a bit more intuitive is to express the reduction of the trace size

achieved by an MDS as the percentage of the original number of trace components. We thus define the *MDS reduction* as $R = (1 - m/n) \times 100$. For example, an explanation ratio of 0.15 translates into a reduction of 85%. This definition means: the higher the reduction, the more effective the MDS.

Next, we need test data to run our experiments on. Since there are no AHP benchmark data sets available, we need to generate data that reflects the examples found in real life. Judging from the examples reported in the literature, we infer that AHP models are hardly ever deeper than 6 levels. However, each dimension can be very wide: For example, an AHP model with 51 attributes in one dimension can be found in [11]. In general, it is not uncommon for an AHP to have about 10 attributes in one of the dimensions [13].

We therefore randomly generated data for AHP examples having between 3 to 6 levels and computed the reduction for each case. For AHP examples with 3 levels the number of components was limited to 30 because there is only 1 intermediate dimension, and the examples in the literature suggest that such a limit is a reasonable assumption. For an AHP examples with 4, 5, or 6 levels, we limited the total number of components in AHP to 100, with intermediate dimensions having between 2-10 components each. To generate graphs for each decision margin, we used 20,000 random inputs. For an AHP problem with a fixed number of levels, the random inputs vary in two regards: (a) the number of attributes at each level, and (b) the values of the decision matrices. Varying the number of attributes results in varying number of trace components. The average of reduction percentages of all the inputs with a given number of trace components, say x, is shown as the reduction percentage for x on the y-axis of the graph.

Another aspect that should be reflected in the test data is whether an explanation is necessary at all. For example, when the first alternative from an AHP process is better than the runner-up in every regard, no explanation is necessary. In contrast, an explanation is most helpful in cases when the two alternatives are really close, that is, when the priority values of the alternatives are similar. To reflect this situation, we filter out those cases whose first two alternatives are not close. We call the relative difference between the priority values of two alternatives their *decision margin*. Specifically, we consider different scenarios in which the decision margin is bounded 1%, 5%, 10%, 20%, and 30%.

Figure 9 shows how MDS reduction varies with the total number of components. We show graphs for AHPs with different number of levels containing plots for different decision margins. The plots show some interesting trends.

First, on average an MDS can prune the number of trace components by about 55-60% even when the top two alternatives are very close, that is, even for a decision margin as low as 1%.

Second, the reduction decreases with smaller decision margins, which makes some intuitive sense, since a greater value distance between alternatives provides more opportunities to explain the difference with fewer trace components. But unfortunately this also means that the efficacy of MDS explanation shrinks when they might be needed most.

Third, we can observe that with an increasing number of levels, the curves "move upwards," that is, for a given decision margin the reduction increases with the number of levels in the AHP problems. In other words, MDS explanations scale well with the structural complexity of AHP problems.

To assess the performance of MADMAX for AHP problems with a given number of levels, we measured the timing information for solving 20000 randomly generated AHP problems with the same number of levels. Performing the experiment on a collection of problems gives us better insight into the average performance of MADMAX. For AHP examples with 4, 5, and 6 levels we constrained the total number of components to be less than 100 as was the case in the MDS evaluation. For AHP examples with 3 levels, we again constrained the number of components to 30, since it is uncommon to find an AHP with 3 levels having more than 30 components. We performed these experiments on a laptop with Intel *i-7* processor with a clock speed of 2.60 GHz, 4 cores, and 16 GB RAM. The timing results are shown in Figure 10. The evaluation shows that the average time taken to solve the problem increases linearly with the number of levels. Moreover, MADMAX runs fast: Even the most computationally intensive AHP problems with six levels can be solved on average in less than 4 milliseconds.

6 Related Work

Since MADMAX is a DSL for AHP decision making with explanations, we compare it first with other AHP decision-making tools. After that we also compare our MDS-based explanations with other explanation approaches.

First, AHP problems can be represented as tables in Excel [1], which is quite effective when getting the priorities for alternatives is the only requirement. However, this approach is limited and doesn't provide sensitivity analyses, visualizations, or explanations of the results. Most other AHP tools fall into one of the two categories: (1) *UI-based tools*, which allow users to describe the shape of the AHP model as well as the decision matrices in an interactive way, and (2) *language-specific libraries*, in which the problem is encoded using representations offered by the host language.

The most prominent tool in the first category is *Expert Choice* [7]. Other tools in this category are *TransparentChoice* [10] and *Super Decisions* [8]. These tools have built-in facilities to perform sensitivity analysis on the various attributes of the AHP problem. The presence of a graphical user interface and built-in functionalities makes these tools convenient to use, which is essential for non-technical users. However, most of these tools are sold as end-user products and can't be adapted to specific user needs. A major disadvantage is

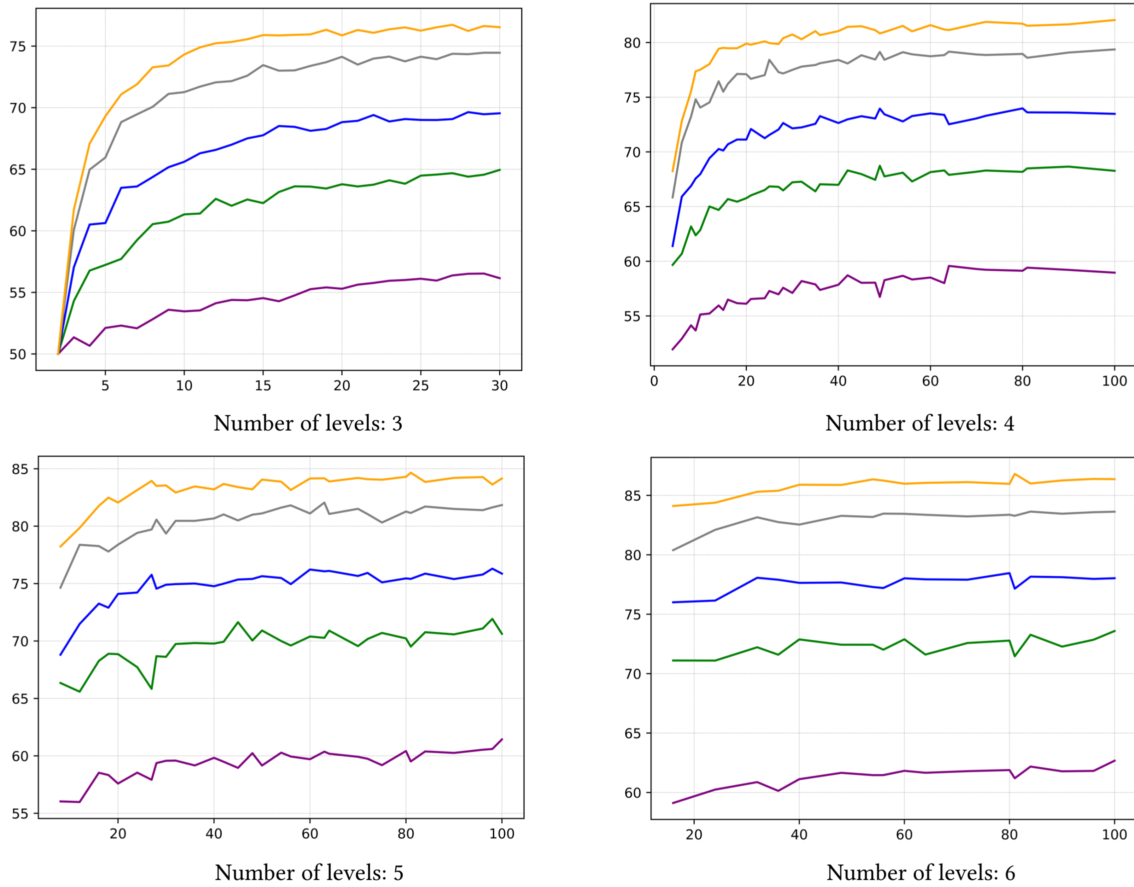

Figure 9. MDS reduction dependent on the number of components for AHP problems with different number of levels. Decision margins: $\leq 30\%$, $\leq 20\%$, $\leq 10\%$, $\leq 5\%$, $\leq 1\%$

Levels	Mean (sec)	Std Dev (sec)
3	24.26	0.17
4	42.88	1.51
5	63.04	1.02
6	70.77	0.92

Figure 10. Performance of MADMAX on AHP Problems

that these tools don’t integrate with existing programming languages. In contrast, MADMAX allows users to integrate AHP decision making in their software. Perhaps the biggest advantage of MADMAX over the aforementioned tools is the ability to provide explanations for the decisions in addition to sensitivity analysis.

The second category of AHP tools are language-specific AHP libraries, such as the *ahp* library for R [9], the *pyAHP* library for Python [12], and the *hahp* library for Haskell [5]. However, these libraries require verbose and clunky encodings of AHP problems. For example, the ahp library in R asks the user to specify the AHP model in a separate file in a special format (yaml), which is then imported in the main R program. Similarly, the pyAHP and hahp libraries encode the AHP model in JSON format. Creating such encodings is time consuming and prone to errors. Moreover, it is cumbersome to make even minor changes to the encodings of models. In contrast, MADMAX allows users to specify the AHP model succinctly in the source language without the need for a special syntax. Moreover, none of these packages offer explanation primitives.

Sensitivity analysis [21] is the tool of choice employed by decision makers to comprehend the results of various MADM methods, including AHP. Sensitivity analysis is usually the only explanation mechanism available to a decision maker. MADMAX also offers functions for performing sensitivity analysis. An advantage of the representation in MADMAX is that we can report the results of a sensitivity analysis in terms of the original data whereas other tools only report numbers on normalized data, which is often harder to interpret.

In any case, despite being useful, a potential limitation of sensitivity analysis is that it can only analyze the impact of one attribute at a time, keeping other attributes values constant. Thus, sensitivity analysis produces a number of localized explanations. In comparison, our explanation method,

which is based on value differences, is global in the sense that the MDS mechanism takes into consideration the combined impact of various attributes in the decision making, leading to generally more accurate and comprehensive explanations.

The idea of MDS explanation was first proposed in [6] in the context of explaining the results of dynamic programming algorithms. We have adopted this notion and extended it to work with multiple layers of attributes. This extension allows us to explain hierarchical decompositions of data, which is not possible in the original approach.

Program traces are often used to explain computation results. However, traces can become very large for non-trivial programs. Using *program slicing* one can try to remove non-relevant parts of the trace. Specifically, *dynamic slicing* is a technique for isolating segments of a program that potentially contribute to the value computed at a point of interest. For example, Biswas describes the generation of a dynamic slice for a higher-order programming language [4]. Perera et al. describe the use of dynamic slicing on traces to specifically generate explanations for the executions of functional programs [14]. Their approach is based on eliminating those parts from a trace that do contribute to user-selected parts of outputs. This approach has been extended to imperative functional programs in [15]. Although the approach is effective, the resulting traces can still be quite large and include many details that while technically relevant for the computation are not contributing to the explanation. We have recently shown how to generate more succinct program traces that can be tailored to user needs by offering users the ability to apply specific trace filters that can be defined with the help of a trace query language [2, 3]. MADMAX does not produce traces as explanations. Instead of collecting intermediate values over the execution of a program, value decompositions maintain a more granular representation of values that are still aggregated. Our approach requires some additional work on the part of the programmers in decomposing the inputs (even though in our implementation we have tried to minimize the required effort). An advantage of our approach is that we only record the information relevant to an explanation in contrast to generic tracing mechanisms, which generally have to record every computation that occurs in a program, and require aggressive filtering of traces afterwards.

7 Conclusions

We have presented the Haskell-embedded DSL MADMAX that facilitates the high-level representation of (potentially hierarchical) multi-attribute decision-making problems. MADMAX specifically supports the AHP decision-making method.

MADMAX offers two distinctive advantages over previous approaches. First, it supports the gradual adaptation and evolution of AHP problems, since users are freed from the (re-)encoding of data. Second, it reports results in a more user-friendly way: On the one hand, results are presented in terms of the problem description (instead of encodings). On the other hand, results can be succinctly explained by direct comparison with rivaling alternatives using the concept of minimal dominating sets.

References

[1] Addinsoft. [n. d.]. XLSTAT: An Excel Addin for AHP. https://www.xlstat.com/en/solutions/features/analytic-hierarchy-process.

[2] D. Bajaj, M. Erwig, D. Fedorin, and K. Gay. 2021. A Visual Notation for Succinct Program Traces. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*. To appear.

[3] D. Bajaj, M. Erwig, D. Fedorin, and K. Gay. 2021. Adaptable Traces for Program Explanations. In *Asian Symp. on Programming Languages and Systems*. To appear.

[4] S. K. Biswas. 1997. *Dynamic Slicing in Higher-order Programming Languages*. Ph. D. Dissertation. University of Pennsylvania.

[5] Y. Dubromelle and J. Prunaret. 2016. hahp: An AHP library for Haskell. https://github.com/Taeradan/hahp.

[6] M. Erwig and P. Kumar. 2021. Explainable Dynamic Programming. *Journal of Functional Programming* 31, e10 (2021).

[7] E. Forman. 1997. Expert Choice. https://www.expertchoice.com/2020.

[8] Creative Decisions Foundation. 1996. Super Decisions. http://www.superdecisions.com/about/.

[9] C. Glur. 2018. ahp: An AHP Library for R. https://cran.r-project.org/web/packages/ahp/.

[10] TransparentChoice Limited. 2013. TransparentChoice. https://www.transparentchoice.com/.

[11] L. Liu, P. Berger, A. Z. Zeng, and A. Gerstenfeld. 2008. Applying the analytic hierarchy process to the offshore outsourcing location decision. *Supply Chain Management* 13 (2008), 435–449.

[12] A. Mishra. 2018. pyAHP: An AHP library for Python. https://github.com/pyAHP/pyAHP.

[13] N. Pan. 2008. FUZZY AHP APPROACH FOR SELECTING THE SUITABLE BRIDGE CONSTRUCTION METHOD. *Automation in Construction* 17 (2008), 958–965.

[14] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. 2012. Functional Programs That Explain Their Work. In *17th ACM SIGPLAN Int. Conf. on Functional Programming*. 365–376.

[15] W. Ricciotti, J. Stolarek, R. Perera, and J. Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proc. ACM Program. Lang.* 1, Article 14 (2017), 28 pages.

[16] R. W. Saaty. 1987. The Analytic Hierarchy Process—what it is and how it is used. *Mathematical Modelling* 9, 3 (1987), 161–176.

[17] T. L Saaty. 2002. Decision making with the Analytic Hierarchy Process. *International Journal of Services Sciences* 1 (2002), 83–98.

[18] T. L. Saaty. 2003. The negotiation and resolution of the conflict in South Africa: The AHP. *ORiON* 4 (12 2003). https://doi.org/10.5784/4-1-488

[19] T. L. Saaty and Y. Cho. 2001. The decision by the US congress on China's trade status: a multicriteria analysis. *Socio-Economic Planning Sciences* 35, 4 (2001), 243 – 252.

[20] V. Tang and E. Collar. 1992. IBM AS/400 new product launch process ensures satisfaction. *Long Range Planning* 25, 1 (1992), 22 – 27. http://www.sciencedirect.com/science/article/pii/002463019290306M

[21] E. Triantaphyllou and A. Sánchez. 1997. A Sensitivity Analysis Approach for Some Deterministic Multi-Criteria Decision-Making. *Decision Sciences* 28 (1997), 151–194.

[22] E. Z. Zavadskas, Z. Turskis, and S. Kildienė. 2014. State of art surveys of overviews on MCDM/MADM methods. *Technological and Economic Development of Economy* 20, 1 (2014), 165–179.