Visual Explanations of Probabilistic Reasoning

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Eric Walkingshaw
Oregon State University
walkiner@eecs.oregonstate.edu

Abstract

Continuing our research in explanation-oriented language design, we present a domain-specific visual language for explaining probabilistic reasoning. Programs in this language, called explanation objects, can be manipulated according to a set of laws to automatically generate many equivalent explanation instances. We argue that this increases the explanatory power of our language by allowing a user to view a problem from many different perspectives.

1 Introduction

In this paper we present a domain-specific visual language for explaining probabilistic reasoning. This language is an *explanation-oriented language*, a language whose primary goal is not to describe the computation of values, but to provide explanations of how and why those values are produced. A significant feature of the language, and a contribution to explanation-oriented language design in general, is the ability to automatically derive many equivalent explanations from a single initial explanation. This allows a user to examine a problem from many different points of view, increasing the explanatory power of the language at no additional cost to the explanation creator.

Probabilistic and statistical reasoning is an important component of scientific research and many practical situations in every day life, but it is often difficult for people with little or no corresponding educational background. Even simple conditional probability problems can cause confusion among laypeople, and disbelief of counter-intuitive solutions often remains despite elaborate justifications.

Consider the following puzzle: Three coins are flipped. Given that two of the coins have come up heads, what is the probability that the third has come up tails? Many people respond that the probability is 50%. But it is, in fact, 75%.

To understand the above puzzle, one's best recourse is to ask somebody who already understands the problem for an explanation. Personal explanations are ideal because explainers can rephrase explanations, answer questions, clarify assumptions, and provide related examples as further illustration. Unfortunately, personal explanations are a comparatively scare resource—they are not always available and cannot be easily shared or reused.

When a personal explanation is unavailable, one might seek other explanatory material on the web or in a textbook. These explanations have much higher availability, shareability, and reusability. A web-based explanation, in particular, can be accessed any number of times from anywhere in the world. The trade-off is that these explanations lack the adaptability of a personal explanation.

The goal of our DSL is to combine the positive aspects of both personal and electronically-available explanations. An explanation designer, who understands the probability problem to be explained, could use the language to create an explanation object and post it on the web, making it widely available and reusable. By applying transformations we can take this single explanation and automatically generate many alternative, equivalent explanations. Consumers of this explanation, who do not yet understand the problem, could navigate between these alternatives, making the explanation object flexible and adaptive.

Our goal is not, however, to replace personal or web-based explanations, but rather to complement them. For example, a web page might contain textual and graphical explanations of the problem, links to background material, and an explorable explanation object.

2 Previous Work

This work furthers our exploration and definition of the explanation-oriented language paradigm, and expands on previous work in the domain of probability and probabilistic reasoning. We first promoted explainability as a primary design criterion in [4], where we presented a visual language for defining and explaining strategies in game theory.

Our work in probabilistic reasoning began in [3], where we presented a DSL embedded in Haskell (DSEL) for creating and manipulating probabilistic values. In this language, probabilistic values are represented as probability distributions, essentially lists of values paired with associated probabilities. Building on this work, we designed another DSEL for describing *explanations* of probabilistic reasoning which is presented in [5], which also contains the initial ideas for the visual notation presented here.

Both the explanation DSEL and this visual notation rely heavily on a "story-telling" model of explanations. In this model, an explanation is a sequence of steps that guide the

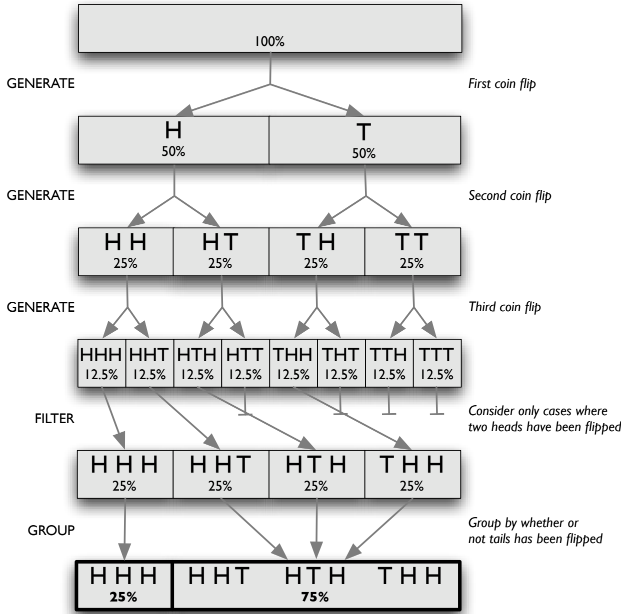

Figure 1. Explaining the three-coins problem

user from some initial state to the explanandum[1], where each step is a well-understood operation that transforms the explanation state. In the next section we introduce our visual language and provide a corresponding formal notation which can be used to define an *explanation semantics* and derive theorems for generating alternative explanations.

3 Visual Explanations

In order to realize the story-telling metaphor, we must identify a representation of state within the domain, and a set of composable operations. Figure 1 demonstrates our adaption of this model to the domain of probabilistic reasoning with an explanation of the three-coins problem introduced in Section 1. The state, shown at each step of the explanation, is a probabilistic distribution, while operations correspond to the annotated transitions between states. We refer to the overall structure as a distribution graph.

Distributions. A probability distribution of type $\langle A \rangle$ is a function $D : A \to [0..1]$, such that $\sum_{(x,p)\in D} p = 1$. In our formal notation, we give distributions explicitly by writing the probabilities of values as superscript percentages. For example, $coin = \langle H^{60}, T^{40} \rangle$ is the distribution of a loaded coin that lands on heads with 60% probability. In this example, *coin* has type $\langle C \rangle$ where type C contains values $\{H, T\}$.

Visually, we represent distributions using the common metaphor of spatial partitioning. A horizontal area is partitioned into blocks which contain values of the appropriate type; the area of each block is proportional to the probability of the contained value.

[1]The thing that is to be explained.

Spatial partitioning is a good metaphor for representing probability distributions since it captures many abstract probabilistic axioms. For example, the sum of areas of all partitions equals the area of the original rectangle (which represents 100% probability), and as the area of one partition increases, the area of other partitions must necessarily decrease. From the perspective of operations which transform this state, we can view the probability space as a resource which operations can split, merge, and redistribute amongst values.

Distribution Graphs. A distribution graph is given by a sequence of distributions $D_1 \dots D_k$ where each D_{i+1} is derived from D_i by a *distribution transformation*, and the blocks of D_i and D_{i+1} are connected by a set of *multi-edges*. (To simplify our notation in the following, we write $A^{(n..m)}$ for the sequence $A_n \dots A_m$; for the special case $n = 1$, we write more concisely $A^{(m)}$).

In the visual language, a transformation is indicated by the operation name on the left (e.g. generate, filter, group) and an annotation describing the transformation on the right. Formally, a distribution transformation is some function $f : \langle A \rangle \to \langle B \rangle$. A multi-edge is a pair (V, W) where $V \subseteq D_i$ and $W \subseteq D_{i+1}$, and either $|V| = 1$ or $|W| = 1$ (or both). Thus, each multi-edge corresponds to a set of plain edges. Together, a distribution transformation and set of multi-edges form an *edge level*, $E_i = f_i \succ M_i$, where M_i is the set of all multi-edges between D_i and D_{i+1}.

A distribution graph of level k is a pair $G = (D^{(k)}, E^{(k-1)})$ that contains a list of k distributions and a list of $k-1$ edge levels. Note that although each transformation has an associated annotation, we do not formally represent these or consider annotation transformations; this could be addressed in future work.

As a notational aside, in our application domain we often deal with heterogeneous lists, whose types we write as $T^{(k)}$ for a list with k elements of types $T_1 \dots T_k$.[2] We also use this notation for individual lists. For example, $3HT$ represents a result of a die roll followed by two coin flips. Assuming the type $R = \{1, \dots, 6\}$, the type of this value is RCC. We use juxtaposition to denote the addition of elements to lists and the concatenation of two lists.

In the rest of this section we present a subset of available operations for transforming distributions. For a complete listing, please refer to our previous work in [5]. For each operation we give a high-level description and define an explanation semantics which illustrates the evolution of distributions over the operation.

Generators. Used primarily to introduce new probabilistic events into a distribution graph, a generator function f takes a value and injects it into a distribution. That is, $f = \lambda x.xD$ where xD is short for $\{(xy, p) \mid (y, p) \in D\}$.

[2]A heterogeneous list of length k is isomorphic to a k tuple.

For example, to introduce the throw of a loaded coin we can define the following generator $throwCoin : A \to \langle AC \rangle$.

$$throwCoin = \lambda x.x\,coin = \lambda x.\langle xH^{60}, xT^{40} \rangle$$

The distribution transformation from D_1 to D_2 induced by a generator $f = \lambda x.xD$ is attained by applying f to every element in D_1 and scaling the probabilities, as follows.

$$\lambda D_1.\{(xy, pq) \mid (x,p) \in D_1, (y,q) \in D\}$$

In the visual language, generator edges have one tail, connected to a block in D_1, and many heads, connected to the corresponding blocks in the sub-distribution of D_2.

The explanation semantics of a generator f are defined as a function that maps a distribution to a distribution graph.

$$\begin{aligned} \llbracket f \rrbracket &= \lambda D_1.(D_1 D_2, E_1) \\ &\quad \text{where } D_2 = f(D_1) \\ &\qquad\qquad E_1 = f \succ \{(x, D_2|_x) \mid x \in D_1\} \\ &\qquad\qquad D_2|_x = \{(x'y, p) \in D_2 \mid x' = x\} \end{aligned}$$

Filters. If p is a predicate on A, the filter $\llparenthesis p \rrparenthesis$ is a distribution transformation from D_1 to D_2 of type $\langle A \rangle \to \langle A \rangle$, which removes blocks from D_1 and redistributes the area of eliminated blocks proportionally among all remaining blocks in D_2. Edges from eliminated blocks in D_1 end with terminating bars, while edges from passing blocks connect to the corresponding blocks in the result. The explanation semantics of filtering is defined as follows.

$$\begin{aligned} \llbracket \llparenthesis p \rrparenthesis \rrbracket &= \lambda D_1.(D_1 D_2, E_1) \\ &\quad \text{where } c = 1 - \textstyle\sum_{(x,q) \in D, \neg(p(x))} q \\ &\qquad\qquad D_2 = \{(x, q/c) \mid (x,q) \in D_1, p(x)\} \\ &\qquad\qquad E_1 = \llparenthesis p \rrparenthesis \succ \{(\{x\}, \{x\}) \mid x \in D_2\} \end{aligned}$$

Groups. A grouping operation introduces a simplified view of a distribution, hiding some of the underlying structure by visually merging some blocks. This is performed based on a function f which maps elements of distribution D_1 onto some type. All elements from D_1 that are mapped to the same value are grouped together in the resulting distribution D_2; this is expressed by the notation D_1/f.

A grouped distribution has the type $\langle\langle A \rangle\rangle$[3] and is visually represented by a thicker border. A distribution remains grouped until the grouping is removed by an ungroup operation (see below). If the input distribution is not already a grouped distribution, grouping has the type $\langle A \rangle \to \langle\langle A \rangle\rangle$. Otherwise, the type is $\langle\langle A \rangle\rangle \to \langle\langle A \rangle\rangle$, which means that the additional group operation extends the existing grouping.

Edges connect each block in D_1 to the corresponding block in D_2, meaning that each block in D_1 will have exactly one outbound edge while blocks in D_2 may have many inbound edges.

[3] Strictly speaking, the innermost distribution is not a proper distribution because the probabilities do not sum up to 100%.

The explanation semantics of grouping is defined below.

$$\begin{aligned} \llbracket \curlyvee f \rrbracket &= \lambda D_1.(D_1 D_2, E_1) \\ &\quad \text{where } D_2 = \{(X, \textstyle\sum_{(x,p) \in X} p) \mid X \in D_1/_f\} \\ &\qquad\qquad E_1 = \curlyvee f \succ \{(\{x\}, X) \mid X \in D_2, x \in X\} \end{aligned}$$

The definition of any distribution transformation is lifted in the obvious way to grouped distribution. Given a grouped distribution $D' = \curlyvee f(D)$, the application of a distribution transformation g to D' is essentially moved to the underlying distribution D, that is, $g(D') = \curlyvee f(g(D))$.

Ungroups. Ungrouping simply removes the view introduced by a grouping operation, restoring the underlying distribution. The operation takes a grouped distribution of type $\langle\langle A \rangle\rangle$ and takes the union of all groups to produce a distribution of type $\langle A \rangle$.

$$\begin{aligned} \llbracket \Upsilon \rrbracket &= \lambda D_1.(D_1 D_2, E_1) \\ &\quad \text{where } D_2 = \{(x,p) \mid X \in D_1, (x,p) \in X\} \\ &\qquad\qquad E_1 = \Upsilon \succ \{(X, \{x\}) \mid X \in D_1, x \in X\} \end{aligned}$$

Note that group and ungroup are not truly inverses since the composed view generated by consecutive groupings will be removed by a single ungroup operation.

Maps. A map transforms a distribution by applying a function f to all values in the distribution. A distribution transformation can be obtained by grouping with f and then replacing each group with the common value the elements in each group are mapped to. When f is one-to-one, a mapping does not affect the structure of the distribution, only the values within each block.

$$\begin{aligned} &\llbracket *f \rrbracket = \lambda D_1.(D_1 D_2, E_1) \text{ where} \\ &D_2 = \{(\{y \mid (x,p) \in X, f(x) = y\}, \textstyle\sum_{(x,p) \in X} p) \mid X \in D_1/_f\} \\ &E_1 = *f \succ \{(\{u\}, \{v\}) \mid u = (x,p) \in D_1, v = (y,q) \in D_2, f(x) = y\} \end{aligned}$$

As in other operations, edges connect blocks in D_1 with the corresponding blocks in D_2.

In the next section we develop theorems for rearranging, merging, and introducing these operations to automatically generate alternative but equivalent explanations.

4 Generating Alternative Explanations

One of the major contributions of this work is the ability to take a single explanation and automatically generate many alternative explanations of the same problem. We believe that by providing a navigable explanation object with many explanation instances we can greatly increase explanatory power compared to static images. In this section we present and discuss a number of theorems for automatically generating alternative explanations.

Many operations can be merged with adjacent operations of the same type. We refer to this as *operator fusion*. For example, the first three generators in the example in Figure 1 can be fused to form the generator in Figure 2.

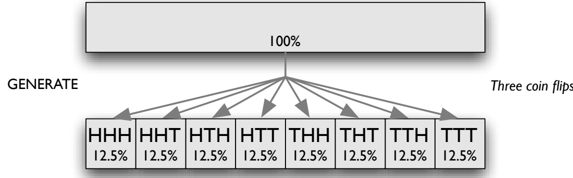

Figure 2. Three generators fused into one

The composition of two operations is defined by joining the resulting distribution graphs of each. This is captured in the explanation semantics of composition below.

$$
\begin{aligned}
[\![f \triangleright g]\!] &= \lambda D_1.(D^{(1..n)}, E^{(1..n-1)}) \\
&\quad \text{where } (D^{(1..k)}, E^{(1..k-1)}) = [\![f]\!] D_1 \\
&\qquad\qquad (D^{(k..n)}, E^{(k..n-1)}) = [\![g]\!] D_k
\end{aligned}
$$

Generator fusion, as demonstrated in Figure 2, follows directly from the definition of composition.

Theorem 1 *Generator Fusion*
$\lambda x.xD \triangleright \lambda y.yD' = \lambda x.\{(xy, pq) \mid (x,p) \in D, (y,q) \in D'\}$

Generator fusion reduces visual noise, allowing users to focus on more important and interesting parts of the explanation. While it reduces the vertical complexity of the story by reducing the number of steps, it increases the horizontal complexity—the initial area is partitioned many more times in a single step. Horizontal vs. vertical complexity is a recurring trade-off between alternative explanations.

Theorems for the fusion of other operators presented in Section 3 are very straightforward. Adjacent maps can be fused by simply composing their associated functions.

Theorem 2 *Map Fusion*
$*f \triangleright *g = *(g \circ f)$

Adjacent filters can be fused by filtering with the conjunction of their predicates.

Theorem 3 *Filter Fusion / Splitting*
$\llparenthesis p \rrparenthesis \triangleright \llparenthesis p' \rrparenthesis = \llparenthesis p \wedge p' \rrparenthesis$

Theorem 3 defines two explanation transformations. Applied left-to-right it allows explanations to be shortened through filter fusion, at the expense of a more complicated filter predicate. Applied right-to-left it can yield a simplified explanation of a filtering process, by splitting it into two operations, at the expense of an additional explanation step.

A similar dual transformation exists for groupings.

Theorem 4 *Group Fusion / Splitting*
$\curlyvee f \triangleright \curlyvee g = \curlyvee(g \circ f)$

Another explanation transformation involves a process called *filter lifting* which is demonstrated by an alternative

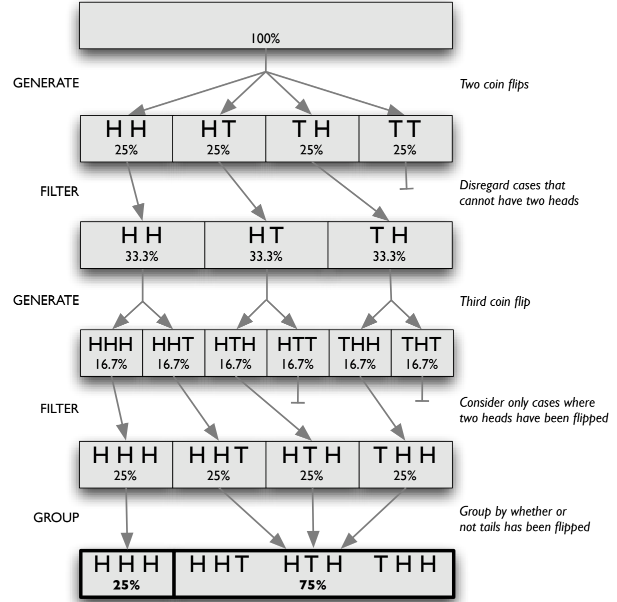

Figure 3. Example of filter lifting

explanation for the three-coins problem given in Figure 3. If all descendants of some block are eliminated by a downstream filter, then we can introduce a filter immediately below that block, filtering it out.

Theorem 5 *Filter Lifting*
If $p(xy) \implies p'(x)$, *then* $\lambda x.xD \triangleright \llparenthesis p \rrparenthesis = \llparenthesis p' \rrparenthesis \triangleright \lambda x.xD \triangleright \llparenthesis p \rrparenthesis$

Often we want to transform an explanation to make the effects of a particular part of the story more explicit. We can do this by introducing a group-ungroup pair to isolate the effects in a process called *group bracketing*. This is demonstrated in Figure 4, where we have replaced the filter operation in Figure 1 with a group, filter, and ungroup. In this example, the predicate of the original filter is used as the grouping function, generating two groups; one passes the filter and the other is removed. The ungrouping then restores the underlying distribution.

Group bracketing is defined by the following theorem.

Theorem 6 *Group Bracketing*
If g *does not contain a grouping or ungrouping operation, then* $g = \curlyvee f \triangleright g \triangleright \overline{\curlyvee}$

There are more laws that allow the lifting of group operations, and commutation of filters and groups with maps, but we must omit these due to lack of space.

5 Related Work

In Section 2 we introduced our previous work in the domain of probability and probabilistic reasoning. Although there are other languages which support the computation of

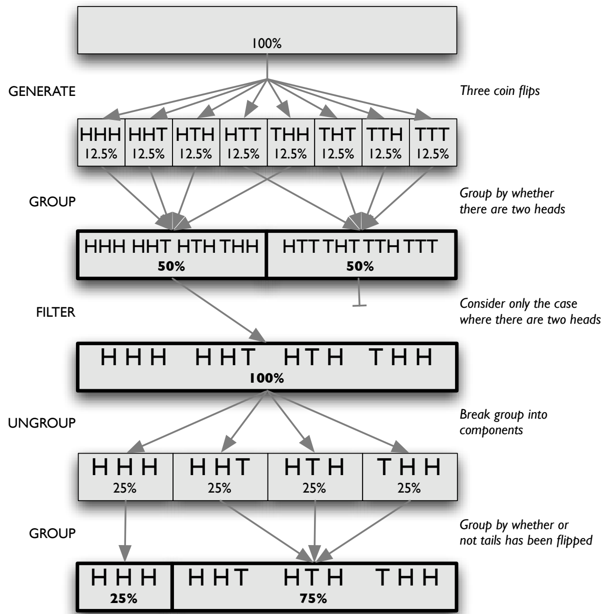

Figure 4. Example of group bracketing

probabilities, such as IBAL [10], there does not seem to be any other work on explaining these computations. Domain-specific explanation support is not completely new, however, and can be found in the field of algorithm animation [8], where the workings of computer algorithms are explained with custom-made or (semi-)automatically generated animations [7]. The work of Blumenkrants et al. is particularly relevant, where the use of the story-telling metaphor was demonstrated to increase the explanatory power of their algorithm animations [2].

Code debuggers represent a class of widely used explanation systems. Very generally, the goal while debugging is to obtain an explanation of some program behavior, usually as part of an effort to fix some fault. While debuggers help users find this information in the code (often after much time and effort), most operate at such a low level that the output and effects they produce could scarcely be considered an explanation. The WHYLINE system [9] inverts the debugging process, allowing users to ask questions about program behavior, and responding by pointing to parts of the code responsible for the outcomes. Although this system improves the process significantly, it can still only point to places in the program, limiting its explanatory power. We have extended this idea in the domain of spreadsheets by allowing users to express expectations about the outcomes of cells, then generating change suggestions that would produce the desired results [1].

Finally, our language is related to dataflow langauges, in which data is incrementally modified by passing through a directed graph of operations [6]. Our language could be viewed as a dataflow language with the single complex data type of probability distributions and the operations described in Section 3. Each operation has a single distribution as input and a single distribution as output, allowing only the possibility of linear graphs. In [5] we discuss branching operations which would make these probability flow graphs somewhat more complex.

6 Conclusions and Future Work

This language continues our work in the area of explanation-oriented programming and language design. A primary contribution of this work is the formulation of laws for deriving equivalent explanations of a particular problem. In order to fully realize the potential of explorable explanation objects, however, we need an algorithm for generating alternatives, and heuristics for identifying those that will be most useful. In future work we hope to generalize the notation for modeling arbitrary explanations. We will also investigate the user-interface aspects of the language, which would support the development of an end-user tool for exploring explanation objects.

References

[1] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th IEEE Int. Conf. on Software Engineering*, pages 251–260, 2007.

[2] M. Blumenkrants, H. Starovisky, and A. Shamir. Narrative Algorithm Visualization. In *ACM Symp. on Software Visualization*, pages 17–26, 2006.

[3] M. Erwig and S. Kollmansberger. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

[4] M. Erwig and E. Walkingshaw. A Visual Language for Representing and Explaining Strategies in Game Theory. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2008.

[5] M. Erwig and E. Walkingshaw. A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conference on Domain-Specific Languages*, 2009. To appear.

[6] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[7] V. Karavirta, A. Korhonen, and L. Malmi. Taxonomy of Algorithm Animation Languages. In *ACM Symp. on Software Visualization*, pages 77–85, 2006.

[8] J. Kerren, A. Stasko. Algorithm Animation – Introduction. In S. Diehl, editor, *Revised Lectures on Software Visualization*, LNCS 2269, pages 1–15. 2001.

[9] A. J. Ko and B. A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *IEEE Int. Conf. on Software Engineering*, pages 301–310, 2008.

[10] Ramsey, N. and Pfeffer, A. Stochastic Lambda Calculus and Monads of Probability Distributions. In *29nd Symp. on Principles of Programming Languages*, pages 154–165, 2002.