A Visual Language for Explaining Probabilistic Reasoning

Martin Erwig, Eric Walkingshaw

School of EECS, Oregon State University, Corvallis, OR 97331, USA

Abstract

We present an explanation-oriented, domain-specific, visual language for explaining probabilistic reasoning. Explanation-oriented programming is a new paradigm that shifts the focus of programming from the computation of results to explanations of how those results were computed. Programs in this language therefore describe explanations of probabilistic reasoning problems. The language relies on a storytelling metaphor of explanation, where the reader is guided through a series of well-understood steps from some initial state to the final result. Programs can also be manipulated according to a set of laws to automatically generate equivalent explanations from one explanation instance. This increases the explanatory value of the language by allowing readers to cheaply derive alternative explanations if they do not understand the first. The language is comprised of two parts: a formal textual notation for specifying explanation-producing programs and the more elaborate visual notation for presenting those explanations. We formally define the abstract syntax of explanations and define the semantics of the textual notation in terms of the explanations that are produced.

1. Introduction

Probabilistic reasoning is often difficult to understand, especially for people with little or no corresponding educational background. Even simple questions about conditional probabilities can have counterintuitive solutions, causing confusion and even disbelief among laypeople despite elaborate justifications. Consider, for example, the following conditional probability problem: *Three coins are flipped. Given that two of the coins show heads, what is the probability that the third coin shows tails?* Many people respond that the probability is 50%, but it is, in fact, 75%.

If you do not understand the solution to the above problem, an explanation will be provided shortly. If you do understand, pretend for a moment that you do not—what would you do? Your best recourse might be to simply ask someone that does understand. A good personal explanation is ideal because the explainer can rephrase the explanation, answer questions, clarify assumptions, and provide related examples as further illustration. Unfortunately, good personal explanations are a comparatively scarce resource; they are not always available, and cannot be easily shared or reused.

Email addresses: `erwig@eecs.oregonstate.edu` (Martin Erwig), `walkiner@eecs.oregonstate.edu` (Eric Walkingshaw)

If you cannot get a personal explanation, you might refer to a probability textbook or seek other explanatory material on the web. These explanations have much higher availability and reusability; a web-based explanation, in particular, can be accessed any number of times from almost anywhere in the world. The trade-off is that these explanations lack the flexibility and adaptability of a personal explanation. They are rarely presented in terms of the specific problem at hand and cannot respond to the questions and confusions of a particular person.

In this paper we bridge this gap with a domain-specific, visual language called *Probula*[1] for explaining probabilistic reasoning problems like the three coins problem above. Using Probula, we can produce visual explanations for a wide range of conditional probability problems, combining the flexibility and accessibility benefits of personal and electronic explanations. We also provide a set of laws for transforming explanations created in Probula into alternative, equivalent explanations. This adds some of the adaptability of personal explanations, allowing users who do not understand an initial explanation to view the problem from different perspectives, to reduce understood parts of the explanation to focus on more difficult parts, and to add and remove abstraction. However, our goal is *not* to replace personal or web-based explanations, but to complement them. For example, a web page might provide a textual explanation along with a Probula explanation as a supplementary resource, or a teacher might use a Probula explanation as a visual aid for a verbal explanation, employing automatically generated alternatives to illustrate different points.

Probula is an example of an explanation-oriented language—a language that supports *explanation-oriented programming* (EOP). EOP is a new paradigm where the focus of programming is shifted from describing the computation of values to providing explanations of how and why those values are produced. A case for EOP is made in Section 2, along with a brief account of our previous work in this area.

A Probula explanation of the three coins problem is given in Figure 1. The explanation is read from top to bottom, like a story describing how an initial probability distribution is transformed into the solution of the problem. This *story-telling model* is the central metaphor on which the language is built. The idea is that each step in the story is small and easy to understand, and that by following each of these well-understood steps, the reader can see how the ultimate conclusion is reached. In Section 3 we motivate the choice of the story-telling model for explaining probabilistic reasoning and describe its impact on the design of Probula.

We begin the explanation in Figure 1 with an empty probability distribution, that is, a distribution with a 100% probability of no event. The first three steps of the explanation are all generator steps, which introduce new events into a distribution. In this case, each generator represents a coin flip. The filter step encodes our conditional constraint that we consider only cases where two of the three coins show heads. Finally, we group the possibilities in the resulting distribution into two cases according to the question posed by the problem—whether or not the remaining coin shows tails.

The Probula language consists of two levels, each represented by a distinct notation. The visual notation shown in Figure 1 represents the *story-level* and is the more

[1]A portmanteau of *probability* and *fabula*, the Latin word for *story*.

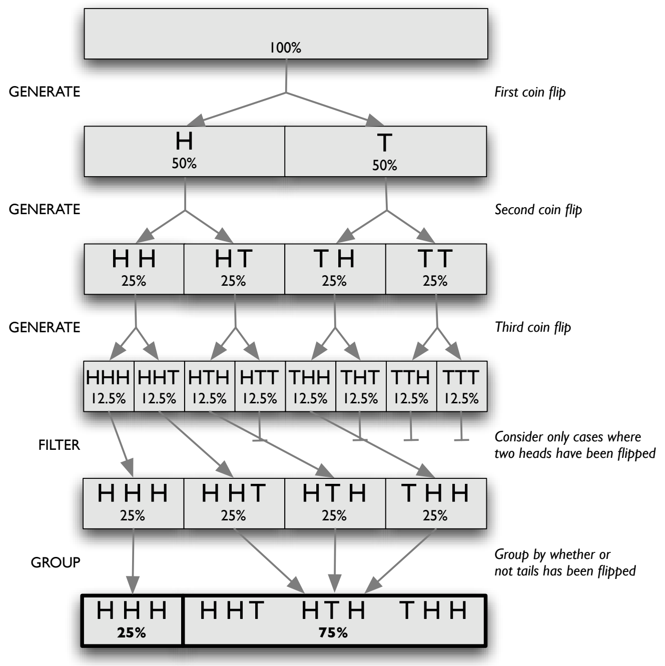

Figure 1: Explanation of the three-coins problem.

important and interesting of the two. An object at this level is a *story*—a view on a transformable explanation that is intended to be read by explanation consumers. The second level is the *plot-level*, represented by a simple textual notation. An object in this language is a *plot*, also called an *explanation program*, which is a formal specification of a story written by the explanation creator. A plot can be instantiated or executed with an initial distribution to produce a story.

In Section 4 we formally define the notations of stories and plots and relate the two through a third notation, a mathematical representation of *distribution graphs*. A distribution graph is essentially a sequence of probabilistic distributions, where each distribution is a group of nodes (one for each value in the distribution), and where the nodes in each adjacent distribution are possibly connected by edges. Specifically, in Section 4, we equate distribution graphs to the abstract syntax [9] of the visual story notation, then define the denotational semantics of plots also in terms of distribution graphs. In other words, a distribution graph represents a story, and the meaning of a plot is the story it produces. We also discuss and motivate many aspects of the concrete syntax of the visual notation, beyond the overarching story-telling metaphor.

Each step of a story is represented at the plot-level by an operation (such as generate, filter, or group) that, given a preceding distribution D_i, extends the distribution graph by a new distribution D_{i+1} and new edges connecting nodes in D_i and D_{i+1}. In Section 5 we define the semantics of all six operations provided by Probula in terms of distributions graphs, as described above.

Finally, a story represents only a single instance from a field of potential explanations for the underlying probabilistic reasoning problem that it explains. Through the use of the above-mentioned transformation laws, we can automatically transform the underlying plot for this story into several "equivalent" alternatives. In Section 6 we define this notion of plot equivalence, then enumerate and formally describe all such transformations. We also discuss the trade-offs that each of these transformation presents, both informally and through the development and application of a simple measure of vertical vs. horizontal distribution graph complexity.

The rest of the paper consists of a discussion of related work in Section 7, followed by conclusions and directions for future work in Section 8.

This work is an extension and consolidation of two earlier papers on explaining probabilistic reasoning [12, 13]. Prior to these papers, we presented a domain-specific embedded language (DSEL) in Haskell for creating and manipulating (but not explaining) probabilistic values [10]. In [12] we reused this DSEL in the design of a second Haskell DSEL for explaining probabilistic reasoning that became the basis for Probula. That paper also provides the first examples of Probula's visual notation along with an extended discussion of philosophical research on explanations and a lengthier motivation for the story-telling metaphor than is provided here. We formalized a subset of the visual notation of Probula in [13] and also provided a formal semantics for a subset of the operations used for creating Probula explanations. That work also contained the first set of laws for transforming a Probula explanation into equivalent alternatives.

In this paper, we improve and complete the formalization of the visual notation begun in [13] and provide a formal semantics for all of the supported operations. In particular, we formalize for the first time the branching operation and the selection of representative examples, and improve the treatment of grouping operations. We also define a formal specification language for generating these explanations and clarify the relationship of this formal notation to the visual notation. Finally, we provide a more complete and detailed description of the explanation-generating transformation laws.

2. Explanation-Oriented Programming

EOP is fundamentally motivated by two simple observations. The first is that programs often produce unexpected results. The second is that programs have value not only as tools for instructing computers, but also as a medium of communication between people.

When a program produces an unexpected result, a user is presented with several questions. First, is the result correct? If so, what is the user's misunderstanding? If not, where is the bug and how can it be fixed? In these situations an explanation of how the result was generated or why it is correct would be very helpful. Unfortunately, such explanations are difficult to come by; in many cases a user's only recourse is to go through a long and exhausting debugging process. For less critical problems, the user may simply give up. With a large and growing class of end-user programmers [33], the ability to provide good explanations is more important than ever.

Although many tools for explaining programs exist, such as debuggers and software visualization tools, these often suffer from a fundamental disconnect from the languages they are meant to explain. If a language provides explanation tools at all, they are typically designed post hoc, as an add-on to the language itself. This forces these tools to reflect a notion of computation that was not designed for explainability (rather, usually to maximize other properties like efficiency and expressiveness), leading to explanations that are difficult to produce and have low explanatory value.

In an explanation-oriented language, the focus on explaining programs is shifted to an earlier stage in language and tool development process, making it a primary design goal of the language itself. In devising syntax and semantics, language designers consider not only the production of values, but also explanations of how those values are obtained and why they are correct.

This idea has a huge potential to innovate language design. Besides the obvious applications to debugging and program analysis, it also suggests an entirely new class of domain-specific languages where the explanation itself, rather than a final value, is the primary output of a program. This represents a stronger sense of explanation orientation and reflects the second observation above, that formal languages are an effective medium for communicating ideas with other people. Probula is an example of such a language, for producing explanations in the domain of probabilistic reasoning.

Using a strongly explanation-oriented language like Probula, an explanation designer, who is an expert in the application domain, can create and distribute explanation artifacts for specific problems in the domain. These artifacts can be customized and explored by non-expert explanation consumers who do not understand the problems or who are trying to gain a greater understanding of the domain itself. There are many possible application domains for such languages, including the explanation of medical procedures, electrical systems, computer algorithms, and all kinds of scientific phenomena. In this paper we address the application domain of probabilistic reasoning, something which is generally not well understood but of great scientific and practical importance, and can thus benefit greatly from a language for creating explanations.

3. The Story-Telling Model of Explanations

The visual notation of Probula relies on several interrelated metaphors. In this section we motivate the most important of these that likens the process of explaining a phenomenon to that of telling a story. This story-telling metaphor fundamentally directed the design of Probula. In this section, we will motivate this design decision and demonstrate its impact on the design.

In our previous work we have conducted a brief survey of the study of explanations and their representation [12]. Our criteria for selecting a model for explaining probabilistic reasoning was that the model should be (1) *simple and intuitive* since the language is intended to be used not just by philosophers, mathematicians, or scientists, but by laypeople who likely have no background in explanation theory or formal modeling; and (2) *constructive*, in the sense that the model should identify specific components of an explanation and their relationships, so that these can in turn be realized by specific language constructs. These criteria rule out several potential models of explanation, such as advanced statistical models [27, 31], models based on physical laws [32], and those based on process theory [6], all of which are too complicated for our purposes. Also ruled out are unificationist models [24], which do not provide a constructive approach (and are also quite complicated).

We believe that the most promising explanation models are those based on *causation*; that is, to explain a phenomenon means to identify what caused it. The idea that explanations reveal *causes* for why things have happened seems intuitive and goes back at least to Plato [30]. Popular models in the philosophical study of causation are based on *causal graphs* [16, 37]. In these models, explanations are represented by directed graphs with variables as nodes (typically representing events or states), where an edge leads from X to Y if X has a direct effect on Y. In our previous work we have worked extensively with one such representation called *neuron diagrams* [27], formalizing and extending the language in [14] and designing a Haskell DSEL for creating and analyzing neuron diagrams in [36].

While causal graphs are a useful and expressive explanation model, they also place a navigational burden on the user. That is, a user must decide which nodes, edges, or paths to look at and in which order. A more linear representation is simpler and easier to understand but also less expressive.

Our explanation-oriented focus and the selection criteria stated above lead us to choose a more linear view of causation and ultimately to the *story-telling model* of explanation. Specifically, we represent an explanation as a sequence of points in a story, where each point corresponds to some state. We move from point to point through the story by an interleaved sequence of state-modifying steps that guide the reader from the initial state to the *explanandum*.[2] In designing domain-specific languages, we often embrace limited computational generality to achieve a close fit with the target domain. Likewise for explanation-oriented DSLs, we embrace a less general explanation model if it fits the needs of our domain, leading to simpler and thus more understandable explanations. The intuitiveness of this model is supported by empirical evidence that suggests that presenting facts and explanations in the form of a story makes them more convincing and understandable [28].

The story-telling model is also constructive in that it suggests a path for realization within a particular domain. Specifically, we must identify (1) some notion of state within the domain that will be manipulated throughout the story, and (2) a set of composable operations for defining the transitions at each step. Tying this back to the linearized notion of causal graphs, we must essentially realize the representation of the nodes (which correspond to states in the story-telling model) and the edges (which correspond to operations). The explanation of the three-coins problem presented in Section 1 demonstrates our adaption of the story-telling model to the domain of probabilistic reasoning. The state, shown at each point in the explanation, is a probabilistic distribution, while operations correspond to the annotated steps between states.

The story-telling model also suggests a distinction between two different levels of notation. An explanation is sufficiently *defined* by its initial state and the sequence of operations that eventually transform it into the explanandum, which we call the *plot*. For example, the plot of an explanation of how an omelette is made might list the ingredients and directions for how to make one. However, the *presentation* of an ex-

[2]The thing that is to be explained.

planation, or the *story*, must also include intermediate states generated by each step. For example, a story explaining the creation of an omelette might show a sequence of illustrations demonstrating each preparation and cooking step that transforms the ingredients into the finished omelette. This distinction between definition and presentation, or plot and story, is reflected in the two notations of Probula.

4. A Language for Explaining Probabilistic Reasoning

In this section we describe how each of the language features motivated by the story-telling model—state, state-transforming operations, and a distinction between story and plot—are represented both visually and formally. We begin with the representation of state as probability distributions in Section 4.1 and of distribution-modifying operations in Section 4.2, which together lead to a notion of distribution graphs that serve as the abstract syntax of stories in Probula. In Section 4.3 we introduce a limited form of branching to Probula stories, enabling the exploration of choices between probabilistic outcomes in explanations, something which is common in probabilistic reasoning. Finally, in Section 4.4 we bring everything together by formalizing the abstract syntax of stories as distribution graphs, presenting a simple textual notation for representing plots, and a semantics function for generating a story (distribution graph) from a plot and an initial distribution. Throughout this section, we also discuss the concrete syntax of the visual notation, pointing out and motivating the important design decisions and visual metaphors used.

4.1. Probability Distributions

Each point in a Probula story is represented by a typed, discrete probability distribution. A distribution D over a discrete random variable of type A is a set of pairs (x, p) where $x : A$ and $\sum_{p \in rng(D)} p = 1$.[3] We write the type of D as $\langle A \rangle$. For readability, when we write distributions explicitly we render each pair (x, p) as x^p where p is expressed as a percentage, and list all such pairs in D between angle brackets. For example, $lcoin = \langle H^{60}, T^{40} \rangle$ is the distribution of a loaded coin that lands on heads with 60% probability. In this example, $lcoin$ has type $\langle C \rangle$ where type C has values $\{H, T\}$.

Visually, we represent a distribution using the common metaphor of spatial partitioning. A horizontal area is partitioned into *blocks*, where each block corresponds to, and is labeled by, a different value-probability pair in the distribution (therefore, we often refer to these pairs in the formal notation also as blocks) and where the area of each block is proportional to its probability. Spatial partitioning is a good metaphor for representing distributions since it directly captures many abstract probabilistic axioms. For example, the sum of the areas (probabilities) of all blocks equals the area of the entire distribution (which represents 100% probability), and as the area of one block increases, the area of other blocks must necessarily decrease. From the perspective of operations that will transform this state, we can view the probability space as a resource that operations can split, merge, and redistribute amongst values.

[3]This representation is isomorphic to a *probability mass function*, $A \to [0..1]$. Also note that we access the domain and range of a set of pairs with $dom(\cdot)$ and $rng(\cdot)$, respectively.

In addition to standard distributions, we also introduce a notion of *grouped distributions*, an example of which can be seen as the explanandum (final state) of the explanation in Figure 1. A grouped distribution is a distribution whose blocks have been consolidated into one or more *groups*. In the visual notation, we represent grouped distributions by drawing thick borders around each group and by eliminating the thinner lines between blocks in the same group.

We indicate a grouped distribution $\boldsymbol{D}$ in the formal notation with boldface. An ungrouped distribution $D : \langle A \rangle$ can be grouped (by the group operation, see Section 5.2) according to a grouping function $g : A \to T$ that maps values in D onto some type T whose elements can be checked for equality. All values that map to the same $t : T$ will be put in the same group. For example, the grouping function used to produce the explanandum in Figure 1 is a predicate from sequences of coin values to a boolean value indicating whether or not one of the coins is tails, which we write as $C^* \to Bool$. We represent a group by a pair (t, X) where $X = \{(x, p) \mid (x, p) \in D,\ g(x) = t\}$. A grouped distribution $\boldsymbol{D}$ is then a set of groups. We write the type of $\boldsymbol{D}$, a distribution of type $\langle A \rangle$ grouped according to a type T, as $\langle A \rangle^T$. For example, the type of the final grouped distribution in the example is $\langle C^* \rangle^{Bool}$.

In the visual notation, a group is labeled by the set of the values it contains and the sum of their probabilities. While the latter is not represented explicitly in the formal notation it can be easily derived. Note also that the grouping value t is not shown explicitly in the visual notation.

Grouped distributions give us a way to visually organize or categorize the blocks in a distribution for the purpose of enhancing an explanation, but they do not significantly affect the underlying distribution from a mathematical perspective. A grouped distribution $\boldsymbol{D}$ is just a lightweight view imposed on the original distribution D, and we can easily recover D by simply removing this view, $D = \bigcup rng(\boldsymbol{D})$; this is the purpose of the *ungroup* operation presented in Section 5.5. Alternatively, we can consider plain distributions as a special case of grouped distributions, grouped by the identity function. That is, $D : \langle A \rangle$ is similar to a grouped distribution $\boldsymbol{D} : \langle A \rangle^A$, where $\boldsymbol{D} = \{(x, \{(x, p)\}) \mid (x, p) \in D\}$. Because of these similarities and since it is convenient, we often refer to both plain and grouped distributions as simply "distributions" (and range over these with D), unless the distinction is important. We always distinguish between the types of plain and grouped distributions, however, so despite the similarity above $\langle A \rangle \neq \langle A \rangle^A$.

4.2. Operations

While each point in a story is represented by a distribution, the structure and progression of the story (its plot) is fundamentally determined by its operations, which describe the steps from one point in the story to the next. An operation can therefore be viewed as a function from one distribution to another. In the following we refer to the ith operation in the plot as o_i and the function implementing this operation as f_i; we use D_{i-1} for the (plain or grouped) distribution at the preceding point in the story and D_i for the subsequent distribution, generated by applying $f_i(D_{i-1})$.

At the non-visual plot level, f_i is a sufficient description of an operation. In the visual story notation, however, operations are a bit more intricate. In addition to the name of the operation (generate, filter, group, etc.) and a brief annotation describing the transformation it performs, the visual representation of a step from one point to the next includes edges between the blocks in the distributions D_{i-1} and D_i. This reflects the convention of data flow diagrams, making explicit the flow of values (possibly modified by the intermediate operation) from one distribution to the next. The basic story telling model implies a representation of state and a set of state-manipulating operations, with the idea that each operation will be small and simple enough to understand in isolation. In Probula, we extend that basic model with an explicit representation of the state *transformations* produced by the operations. In other words, instead of just showing each point in the story (as we did in our textual DSEL [12]), edges in Probula describe *how* each step produces its subsequent story point, allowing readers to direct their effort on more important aspects of the explanation.

In addition to the subsequent distribution D_i, each operation o_i generates a set of edges E_i where each edge is a pair (v, w) where $v \in dom(D_{i-1})$ and $w \in dom(D_i) \cup \{\perp\}$. An edge $(v, \perp)$ represents a *terminating edge*, an edge from D_{i-1} that does not connect to a block in D_i but ends instead with a horizontal bar. This indicates a value in D_{i-1} that does not flow to the next distribution, for example, because it was filtered out, as seen in the filter step of the explanation in Figure 1. Note that since values in a distribution are unique, we can uniquely identify a block in a plain distribution by its value. For grouped distributions, we connect edges to groups rather than blocks, and each group can be uniquely identified by its representative value t.

The meaning of an operation o_i from the perspective of story generation is therefore not just f_i, but a function from D_{i-1} to the pair (E_i, D_i). We call this function the *story semantics* of the operation, written $[\![o_i]\!]$. In Section 5 we will define the story semantics of all of the operations in Probula.

Note that although each operation has an associated annotation, we do not capture this in the formal representation above. While it would be trivial to add an annotation value to the representation and story semantics of each operation, this would clutter the notation without adding any real benefit. More significantly, however, we do not consider the automatic transformation of annotations in Section 6. Instead annotations are considered a purely *secondary notation* [15], that is, a way to convey information outside the constraints of the formal notation.

We call a sequence of distributions and block/group-connecting edges a *distribution graph*, and these form the abstract syntax of Probula's visual notation. A simplified view of distribution graphs follows directly from the definitions of distributions and operations above. A plot can be viewed as a sequence of operations $o_1, \ldots, o_k$, from which a distribution graph $(D_0, E_1, D_1, \ldots, E_k, D_k)$ can be generated by mapping the story semantics across the operations and reducing the list with function application and the initial distribution D_0 (saving the intermediate edges and distributions). Now we add another wrinkle to Probula stories, however, in order to represent a new class of probabilistic reasoning problems.

4.3. Story Branching

Many probabilistic reasoning problems involve a question of choice—a scenario is established and the reader is asked which course of action has the best expected benefit to the actor in the scenario. One of the most famous (and famously misunderstood)

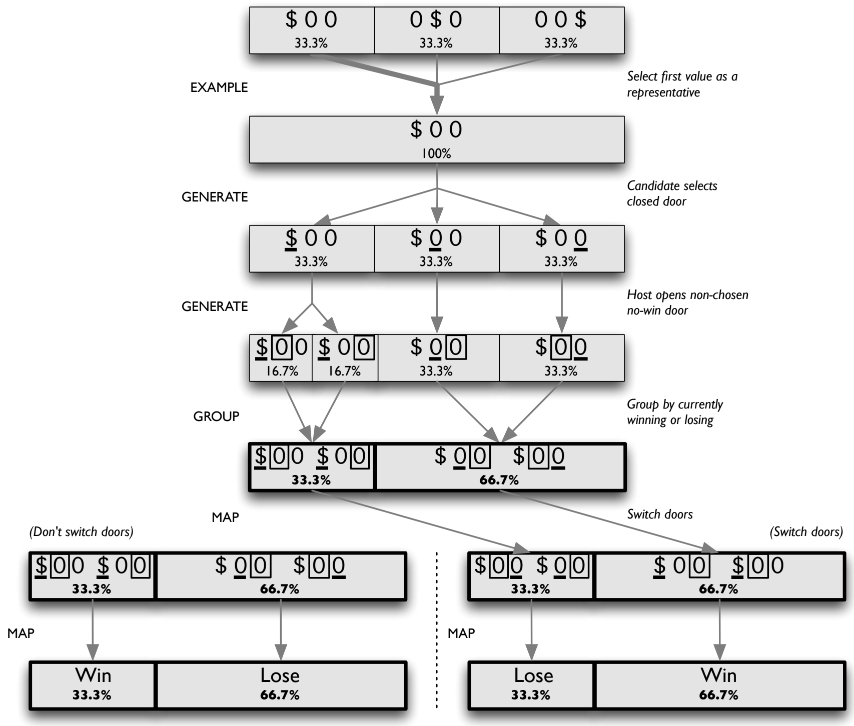

Figure 2: Explanation of the Monty Hall Problem.

examples is the so-called "Monty Hall Problem" [10, 17, 26]. In this scenario, a game show host presents a contestant (the actor) with three doors, one of which hides a prize. The contestant selects one of the doors, then the host opens one of the remaining two doors that does not contain a prize. The contestant is then presented with a choice: stay with the originally selected door or switch to the other closed door? Most people believe that switching doors makes no difference, but this is incorrect—switching doors doubles the contestant's chance of winning from $33\frac{1}{3}\%$ to $66\frac{2}{3}\%$.

A Probula explanation for the Monty Hall Problem is presented in Figure 2. It introduces two new operations and the concept of *story branching*. The story reads as follows: Initially, there are three possibilities to consider—either the prize is hidden behind the first, second, or third door—and each case is equally likely. This is represented by an initial distribution where each value is a triple with elements corresponding to doors; $ indicates the door containing the prize (unknown by the contestant), and 0 indicates a door without a prize. In the first step, we select one of these cases as a *representative example* of the problem since for the purposes of probabilistic analysis, it does not matter *which* door the prize is behind, just that one of the three doors contains a prize. In other words, the three cases are isomorphic.[4] The next two generator steps represent the contestant initial selecting a door and the host opening a prizeless door. The selected door is represented visually by underlining, while the opened door is outlined in a small rectangle. Note that when the contestant has chosen the door with the prize, the host can open either one of the other two doors, while he has no such choice in the other two cases (he must open the remaining door not containing the prize). The generator will therefore split the block in half in the first case, and leave the probability unchanged in the other two cases. Next, we group the results according to whether or not the contestant is currently unknowingly winning the prize or not. This brings us to the choice point, represented in Probula by a branch in the story. In the left branch, the contestant *does not* switch doors, while in the right branch, the contestant switches. Finally, we map the potential outcomes in each branch onto the values "Win" and "Lose", indicating whether or not the contestant won the prize. By comparing the explananda at the end of each branch, we can see that switching doors leads to a 66.7% chance of winning, while not switching doors leads to only a 33.3% chance of winning.

Visually, we represent story branching by showing both possible paths for the story to take, separated by a dotted line. One branch is always preferred in that the edges from the last unbranched distribution lead to the first distribution in only one of the two branches, and step annotations are shown only for the step to the preferred branch. This is simply to reduce noise in the visual notation. An interactive implementation would allow readers to change the preferred branch. Additional annotations (in parentheses) are provided to label each alternative in the branch, to help the reader understand the choice that the branch represents.

Branching fundamentally extends the expressiveness of Probula by allowing us to explain a class of real-world-applicable problems concerned with making the best choices in the face of probabilistic information. However, the verbose notation clearly will not scale beyond only a very small number of branches. This is an intentional design decision, consistent with our emphasis on explainability over other qualities. By embracing the mostly-linear story-telling model, we embraced explainability over expressiveness. With branching, we give some of that expressiveness back, but our particular choice of notation sacrifices scalability to avoid compromising the simplicity and explicitness of the explanations. These sorts of design trade-offs are expected when designing languages for usability [15].

Having demonstrated branching in the visual notation at the story level, in the next subsection we extend the formal representations of plots and distribution graphs to also incorporate branching. We also tie this entire section together with a semantics function for producing stories from a plot and an initial distribution.

4.4. Distribution Graphs, Plots, and Stories

As described in Section 1, the visual notation of Probula tells a story intended to be read by an explanation consumer—someone who presumably does not understand the problem being explained. For this end-user, the visual notation is the only aspect

[4]Using Theorem 11, users can generate alternative explanations if they are not convinced of this. For an example, see Figure 10 in Section 6.7.

of Probula they need ever see. However, we have made a point of distinguishing the visual notation of *stories* in Probula from the formal/textual notations of *plots* and *distribution graphs*, which have different intended audiences. Plots are created by explanation producers—domain experts who understand the problem to be explained and want to explain it to others. A plot is essentially an explanation-producing program while a story is the result of executing that program. Distribution graphs are a formal representation of the abstract syntax of stories, and are used mainly by us, as language creators, to define the language of Probula, formally describe the relationship between plots and stories, and discuss the transformation of stories and plots. In this subsection we will complete the definitions of plots and distribution graphs, and relate the two by extending the story semantics of operations (see Section 4.2) to plots.

In Section 4.2 we said that the plot of a story (its structure and progression) is fundamentally determined by its operations. In Section 5 we catalog and define these operations; here, we consider their organization. Primarily, we compose operations into plots via a right-associative sequencing operator $\triangleright$, terminated by the end-of-plot symbol ε. To produce branched stories like the explanation of the Monty Hall Problem, a plot branching operator $\vdots$ is provided. A simplified grammar for plots is given below, using o to range over operations.

$$\begin{array}{rcll} P & ::= & \varepsilon & \textit{End of Plot} \\ & | & o \triangleright P & \textit{Plot Sequence} \\ & | & P \vdots P & \textit{Plot Branch} \end{array}$$

The grammar is simplified in that it does not encode the syntactic constraints that only some operations can be applied to grouped distributions—while these constraints could be encoded in the grammar, we prefer to enforce them separately to keep the grammar clear and simple. Note also that the syntax allows for empty plots (ε) that produce trivial, single-distribution explanations, and that it does not restrict branching in any way, despite the lack of scalability in corresponding stories. It is the explanation creators' responsibility to use branching judiciously, to keep explanations understandable.

The formal notation of distribution graphs follows directly from the definition of plots, and the notations are almost structurally identical. We reuse the sequence and branch symbols from plots directly since their meaning is similar here and since it is always clear from the context which language we refer to.

$$\begin{array}{rcll} G & ::= & D & \textit{Explanandum} \\ & | & (D,E) \triangleright G & \textit{Story Sequence} \\ & | & G \vdots G & \textit{Story Branch} \end{array}$$

Graphs terminate in an explanandum and each step in a story sequence is represented by a pair of a preceding distribution and a set of edges connecting it to the next distribution in the sequence. Note that this representation introduces a small amount of redundancy when representing branches since the distribution immediately preceding a branch in the story will be represented multiple times (once at the beginning of each branch) in the graph. This representation is chosen because it leads to a more straightforward semantics function.

We define the *story semantics* of an operation o_i in Section 4.2 to be a function $[\![o_i]\!] : D \to (E,D)$ from a preceding distribution D_{i-1} to a pair containing the subsequent

Syntactic Category	Semantics Domain	Description of Semantics Function
D	$A \to [0..1]$	probability mass function
o	$D \to (E,D)$	story semantics of an operation
P	$D \to G$	story semantics of a plot
G	(D,D^*)	limits of a distribution graph

Figure 3: Summary of denotations of constructs in the formal notation.

distribution D_i and a set of edges E_i connecting the two distributions. We use this definition to extend the notion of story semantics to plots. The story semantics of a plot P is a function from an initial distribution to the distribution graph representing the abstract syntax of the generated story, $[\![P]\!] : D \to G$. In the following, $[\![\cdot]\!](D)$ means to apply the function returned by $[\![\cdot]\!]$ to D.

$$\begin{aligned}
[\![\varepsilon]\!] &= \lambda D.D \\
[\![o \triangleright P]\!] &= \lambda D.(D,E) \triangleright [\![P]\!](D') \quad \text{where } (E,D') = [\![o]\!](D) \\
[\![P : P']\!] &= \lambda D.[\![P]\!](D) : [\![P']\!](D)
\end{aligned}$$

For the base case, we simply return the argument distribution as the explanandum when the end of the plot is reached. For branches, we recursively compute the story semantics of each branch and compose the results with a corresponding branch in the graph. For plot sequences, we compute the story semantics for the current operation and apply that function to the preceding distribution, yielding the next distribution and a set of connecting edges. We use these to create a story sequence in the graph and then recursively compute the semantics of the rest of the plot. Using this semantics function we can generate a story given a plot and an initial distribution.

Finally, when describing transformations in Section 6, we will need to talk about whether or not a transformation preserves the meaning of the affected part of the distribution graph. This poses two requirements: (1) a notion of a *distribution subgraph*, and (2) a definition of the semantics (meaning) of a subgraph. The first is straightforward; given a distribution graph $(D_0,E_1) \triangleright \ldots \triangleright (D_{i-1},E_i) \triangleright \ldots \triangleright D_n$ we can obtain the subgraph corresponding to steps j through $k \leq n$ as $(D_{j-1},E_j) \triangleright \ldots \triangleright D_k$. For the second requirement, there are many possibilities. We define define the semantics of a (sub)graph G to be its *limits*, written $lim(G)$, defined as the pair of G's initial distribution and a sequence D^* of the explanandum at the end of every branch in G. For the simple and common case where G is a k-step story without branching, the limits of the graph will therefore be the pair (D_0,D_k). Thus, when we say that the meaning of a subgraph is preserved over some transformation, we mean that the subgraph begins and ends in the same place, although it will presumably change in the middle. This is important for ensuring the locality of transformations.

We conclude this section with a summary of the semantics of the major constructs introduced throughout this section, given in Figure 3.

$$\begin{array}{lcll} o & ::= & \triangle e & \textit{Generate} \\ & | & \curlyvee g & \textit{Group} \\ & | & {*}f & \textit{Map} \\ & | & \mathbf{?}p & \textit{Filter} \\ & | & \curlywedge & \textit{Ungroup} \\ & | & \mathbf{!}x & \textit{Example} \end{array}$$

Figure 4: Probula operations.

5. Catalog of Operations

At the heart of the story-telling model is the idea that a story can be told as a sequence of steps where each step can be easily understood on its own, allowing the reader to focus on the progression of the explanation from the beginning to the explanandum. For explaining the domain of probabilistic reasoning, we identified six types of these steps, implemented as distribution manipulating (and distribution graph generating) operations. In this section we will discuss each of these operations in turn, defining them formally and describing their representation in the visual notation. We will also describe the syntactic constraints on each operation (mentioned in Section 4.4), since some operations can only be applied to either plain or grouped operations, but not both.

Probula's six operations are listed in the syntax definition in Figure 4, in the order that they will appear in this section. Operations are distinguished in the formal notation by a prefixed symbol and may include an argument, usually a function, defining the behavior of the operation. In the syntax, the metavariables e, g, f, and p all range over functions, but each have different constraints on their types. These constraints will be discussed in the context of their associated operations and are also summarized later in Section 5.7. The metavariable x ranges over distribution values.

Recall from Section 4.2 and Figure 3 that the story semantics of an operation o_i is a function from the preceding distribution D_{i-1} to a pair of the subsequent distribution D_i and the connecting edges E_i. In this section, we will define the story semantics of each operation explicitly. When discussing the interaction of types in an operation, however, we will usually refer to just the *distribution transformation*—the function from D_{i-1} to D_i, omitting the edges which carry no interesting type information. In Section 5.7 we will summarize the types of the distribution transformations of all of the operations presented in this section.

5.1. Generate

A generator $\triangle e$ introduces new probabilistic events into a plain distribution. The event generating function $e : A \to \langle B \rangle$ maps the values in the preceding distribution $D : \langle A \rangle$ to distributions of type $\langle B \rangle$, where B is usually derived in some way from A. The distributions produced by e are concatenated and scaled according to the probabilities of the original values in D, in order to produce the resulting distribution $D' : \langle B \rangle$.

As an example, consider the definition of the second generator in the explanation of the Monty Hall Problem in Figure 2, in which the host opens a door that was not chosen and does not contain the prize. This operation can be defined as $\triangle \mathit{openDoor}$ with the function *openDoor* defined below (using pattern-matching).

$$\begin{aligned}
\mathit{openDoor}(\underline{\$}\,0\,0) &= \langle \underline{\$}\boxed{0}0^{50}, \underline{\$}\,0\boxed{0}^{50} \rangle \\
\mathit{openDoor}(\$\,\underline{0}\,0) &= \langle \$\,\underline{0}\boxed{0}^{100} \rangle \\
\mathit{openDoor}(\$\,0\,\underline{0}) &= \langle \$\boxed{0}\underline{0}^{100} \rangle
\end{aligned}$$

If the contestant has selected the door with the prize, there is a 50% chance of the host opening either remaining door. However, if the contestant has chosen a door without the prize, then there is a 100% chance of the host opening the other prizeless door. The effect of applying $\triangle \mathit{openDoor}$ to its preceding distribution can be seen in Figure 2.

Often a generator simply extends a distribution of sequences A^* with a new probabilistic A value. For example, each of the three generators in the three coins problem in Figure 1 adds a new coin flip to the distribution. We can represent a coin flip with the following event generating function $\mathit{flipCoin} : C^* \to \langle C^* \rangle$ that appends heads or tails to the sequences in the existing distribution with 50% probability each.

$$\mathit{flipCoin}(x) = \langle xH^{50}, xT^{50} \rangle$$

For these simple sequence-extending cases, we can automatically derive an event generating function of type $A^* \to \langle A^* \rangle$ given just the distribution of the new value to append $D : \langle A \rangle$. Since this case is very common, we introduce as syntactic sugar the notation $\triangle D$ for $\triangle e$ where $e = \lambda x.\{(xy, p) \mid (y, p) \in D\}$. Using this, we can define the first three steps of the plot for the three coins problem as $\triangle \mathit{coin} \triangleright \triangle \mathit{coin} \triangleright \triangle \mathit{coin}$ where $\mathit{coin} : \langle C \rangle$ is the distribution $\langle H^{50}, T^{50} \rangle$.

The story semantics of a generate operation $\triangle e$ is given explicitly below. Edges are defined between every value x in D and every y produced by $e(x)$. The probabilities in the resulting distribution D' are scaled by multiplying the original probabilities in D with those in the distributions generated by e. We also have to add the probabilities for identical y values that are produced by different invocations of the generator. This is done in the definition of D''.

$$\begin{aligned}
[\![\triangle e]\!] &= \lambda D.(E, D'') \\
&\quad \text{where } \begin{aligned}[t] E &= \{(x, y) \mid x \in \mathit{dom}(D),\ y \in \mathit{dom}(e(x))\} \\
D' &= \{((x, y), pq) \mid (x, p) \in D,\ (y, q) \in e(x)\} \\
D'' &= \{(y, \textstyle\sum_{((x,z),p) \in D', z = y} p) \mid y \in \mathit{rng}(\mathit{dom}(D'))\}
\end{aligned}
\end{aligned}$$

Visually, we represent edges in generate steps by collapsing the common tails of edges from the same source x, emphasizing the idea that generators partition or split blocks in the probability space when a value in D leads to more than one value in D''.

5.2. Group

A group operation $\curlyvee g$ transforms a plain distribution D into a grouped distribution $\boldsymbol{D}'$. As described in Section 4.1, grouping introduces a simplified view of a distribution, hiding some of its underlying structure by visually merging sets of blocks into groups in order to emphasize some aspect of the explanation. Unlike generators (and most other operations), grouping does not fundamentally modify the underlying probability distribution. The operation is provided purely to support the explanation of probabilistic situations rather than their computation.

The operation's argument is a grouping function $g : A \to T$ that maps values in $D : \langle A \rangle$ onto some type T whose elements can be checked for equality. When several values in D map to the same $t : T$, their blocks will be placed in the same group (represented by t) in the resulting grouped distribution $\boldsymbol{D'} : \langle A \rangle^T$. These blocks will then be shown merged in the visual representation, and labeled by the sum of their probabilities. For example, we can represent the final step of the explanation of the three coins problem in Figure 1 by the operation $\curlyvee \mathit{hasTails}$, where the predicate $\mathit{hasTails} : C^* \to \mathit{Bool}$ is true if its argument coin sequence contains at least one tails value and false otherwise. In the visual notation, the blocks of the three sequences that satisfy *hasTails* are grouped together in the explanandum, and the probability of the group is the sum of its component blocks. The remaining block (that does not satisfy the predicate) is isolated in its own group. We also distinguish grouped distributions visually by drawing the distribution with thick borders indicating that the view imposed by the group overrides the standard representation of the distribution.

Recall from Section 4.1 that a grouped distribution is represented by a set of groups. Each group is a pair (t, X), where t is the grouping value produced by g (*true* or *false* in the *hasTails* example above), and X is the set of blocks contained in the group. We build a grouped distribution $\boldsymbol{D'}$ from a preceding plain distribution D and the grouping function g in the story semantics of group operations below.

$$
\begin{aligned}
[\![\curlyvee g]\!] &= \lambda D.(E, \boldsymbol{D'}) \\
&\quad \text{where } \; E = \{(x, g(x)) \mid x \in \mathit{dom}(D)\} \\
&\quad \phantom{\text{where }} \; \boldsymbol{D'} = \{(t, \{(x,p) \mid (x,p) \in D,\ g(x) = t\}) \mid t \in \{g(x) \mid x \in \mathit{dom}(D)\}\}
\end{aligned}
$$

We connect edges to blocks in plain distributions and to groups in grouped distributions, using values to address blocks and grouping values to address groups (see Section 4.2). This is demonstrated in the definition of E, where we produce an edge from each block in D to its corresponding group in $\boldsymbol{D'}$.

There are some differences between the structural representation of grouped distributions in the abstract syntax (distribution graphs) and their rendering in the concrete syntax (visual notation). One is that a group's representative value t is not reflected in the visual notation. We could imagine, for example, replacing the values displayed in a group by their common grouping value t. This would introduce a mechanism for abstraction that might sometimes be useful in representing more complex problems. We choose not to do this for a couple reasons: (1) similar (but irreversible) functionality is provided already by the map operation, discussed in Section 5.3; and (2) there is a trade-off between introducing abstraction and mapping closely to our domain of probability distributions [15]. For a language focused on explainability, we again prefer simplicity, concreteness, and a high closeness of mapping over generality and scalability. Grouping values are needed in the abstract syntax, however, for edge addressing, as seen above, and for the filtering of grouped distribution, discussed in Section 5.4.

Another difference between the abstract and concrete representations of grouped distributions is that in the visual notation we display the probability of a group as the sum of the probabilities of the blocks it contains. This reflects that the primary purpose of grouped distributions, from an explanation standpoint, is to reduce the need for readers to perform hard mental operations [15] by organizing distributions into the relevant cases explicitly. We do not represent this sum explicitly in the abstract syntax since it can be easily derived.

5.3. Map

While the generate and group operations may only be applied to plain distributions, the map operation (and filter in the next section) can be applied to *either* plain or grouped distributions. When applied to a plain distribution, a map operation $*f$ transforms $D : \langle A \rangle$ by applying the function $f : A \to B$ to each value in the distribution, producing a new distribution $D' : \langle B \rangle$. When applied to a grouped distribution $\boldsymbol{D} : \langle A \rangle^T$, map behaves similarly, applying f to every value within every group, preserving the grouping and producing a distribution of type $\boldsymbol{D'} : \langle B \rangle^T$. We will consider the plain distribution case first and adapt this to grouped distributions afterward.

If f is one-to-one, the structure of D is unchanged by a map operation. That is, D' will have the same number of blocks as D and each will have the same probability—only the values in D will be changed. This is equivalent to the traditional functional programming view of mapping a function over a list or other data structure.

If f is not one-to-one, however, the blocks of A values that map to the same B value will be merged; that is, the probability of a value $y : B$ in D' will be the sum of the probabilities of all the x values in D for which $f(x) = y$. For example, consider the following distribution of coin sequences, produced by applying the generator sequence $\triangle coin \triangleright \triangle coin$ to an empty initial distribution.

$$twoCoins = \langle HH^{25}, HT^{25}, TH^{25}, TT^{25} \rangle : \langle C^* \rangle$$

Suppose we also have a function $countTails : C^* \to Int$ that returns the number of tails values in a sequence of coin flips. Applying $*countTails$ to $twoCoins$ yields the following plain distribution of integers.

$$numTails = \langle 0^{25}, 1^{50}, 2^{25} \rangle : \langle Int \rangle$$

In this way, a map can function similarly to a group operation, except that it actually changes the values and structure of the underlying distribution, so the original cannot easily be recovered. If instead we apply $\curlyvee countTails$ to $twoCoins$, we get a distribution of coin sequences, grouped by integers.

$$\langle (0, \{HH^{25}\}), (1, \{HT^{25}, TH^{25}\}), (2, \{TT^{25}\}) \rangle : \langle C^* \rangle^{Int}$$

This overlap in functionality leads to a design decision for explanation creators of whether to employ a map or group operation when combining events.

When choosing between map or group, if subsequent steps in the story *fundamentally rely* on the combined result (that is, if the combined value is needed as input in a later generate or map step) then a map operation must be used. On the other hand, if the combined result is needed only for explanatory purposes, to categorize the events into equivalence classes, then a group may be better since it directly shows the values that make up the cases instead of abstracting these away in the combined value. To demonstrate this, consider again the final step of the explanation of the three coins problem in Figure 1. If we had applied *$hasTails$ instead of $\curlyvee hasTails$ at this step, we would have produced the less useful plain distribution $\langle false^{25}, true^{75}\rangle$ instead of the grouped distribution shown.

To define the story semantics of the map operation, we first introduce a helper function $map : (A \to B) \times \langle A\rangle \to \langle B\rangle$ that maps a function over a plain distribution, merging blocks that map to the same values by summing their probabilities, as described above.

$$map(f, D) = \{(y, \sum_{\substack{(x,p)\in D \\ f(x)=y}} p) \mid y \in \{f(x) \mid x \in dom(D)\}\}$$

With the heavy lifting off-loaded to map, the definition of the story semantics for $*f$ when applied to a plain distribution is mostly trivial.

$$\begin{aligned} [\![*f]\!] &= \lambda D.(E, D') \\ &\quad \text{where } E = \{(x, f(x)) \mid x \in dom(D)\} \\ &\quad\phantom{\text{where }} D' = map(f, D) \end{aligned}$$

The edges for a map operation connect each block in D to its corresponding block in D'. Thus, every block in D will have one departing edge while blocks in D' could have potentially many arriving edges, if f is not one-to-one.

When applying $*f$ to a grouped distribution $\boldsymbol{D}$, we just apply our helper function map to the subdistribution X contained in every group (t, X), thereby mapping f first over the groups in $\boldsymbol{D}$, then over the values in those groups. Note that each X is not properly a distribution since its probabilities do not sum to 1, but that this use of map is still valid since X is structurally identical to a distribution and since map does not rely on its argument representing a complete sample space.

$$\begin{aligned} [\![*f]\!] &= \lambda \boldsymbol{D}.(E, \boldsymbol{D}') \\ &\quad \text{where } E = \{(t, t) \mid (t, X) \in \boldsymbol{D}\} \\ &\quad\phantom{\text{where }} \boldsymbol{D}' = \{(t, map(f, X)) \mid (t, X) \in \boldsymbol{D}\} \end{aligned}$$

A subtle feature of this implementation is that maps act locally with regard to groups. That is, if two values in $\boldsymbol{D}$ map to the same value through f, they will only be merged in $\boldsymbol{D}'$ if they were already in the same group. If they were in different groups in $\boldsymbol{D}$, however, they will remain in different groups in $\boldsymbol{D}'$ and thereafter, until a subsequent ungroup operation (see Section 5.5).

Concerning edges for maps applied to grouped distributions, we draw only a single edge between corresponding groups in $\boldsymbol{D}$ and $\boldsymbol{D}'$, regardless of the effect of f on the values contained within. For example, see the last two steps of the Monty Hall explanation in Figure 2. This is to reduce visual noise in the explanation and provides a way for explanation creators to indicate that all values in a group are affected in a similar way by the map operations.

5.4. Filter

Like map, the filter operation can be applied to either plain or grouped distributions, and has a different story semantics for each case. Again, we consider the case for plain distributions first, then adapt it to grouped distributions.

A filter operation $\mathbf{?}p$ transforms a plain distribution $D : \langle A \rangle$ by removing blocks from D according to the predicate $p : A \to \mathit{Bool}$. The area of the removed blocks is redistributed proportionally among the remaining blocks in the resulting distribution $D' : \langle A \rangle$. For example, recall the distribution $\mathit{numTails} = \langle 0^{25}, 1^{50}, 2^{25} \rangle$ from the previous subsection. Applying the filter $\mathbf{?}\mathit{isPos}$ to *numTails*, where $\mathit{isPos} : \mathit{Int} \to \mathit{Bool}$ is true if its argument is greater than zero, yields the distribution $\langle 1^{66.7}, 2^{33.3} \rangle$. Note that the ratio of the probabilities of the values that pass through the filter is preserved in the resulting distributions.

The story semantics of the filter operation when applied to a plain distribution is defined below. In the definition, the constant c represents the proportion of the probability space that is not filtered out by the predicate. This is used to scale the probabilities of the remaining values.

$$\begin{aligned}
[\![\mathbf{?}p]\!] &= \lambda D.(E, D') \\
&\quad \text{where } \begin{aligned}[t] c &= \textstyle\sum_{(x,q) \in D, p(x)} q \\
E &= \{(x,x) \mid x \in \mathit{dom}(D), p(x)\} \cup \{(x, \bot) \mid x \in \mathit{dom}(D), \neg p(x)\} \\
D' &= \{(x, q/c) \mid (x,q) \in D, p(x)\}
\end{aligned}
\end{aligned}$$

As described in Section 4.2, edges from blocks eliminated by a filter end with terminating bars while edges from blocks whose values pass the predicate are connected to their corresponding blocks in the subsequent distribution.

For grouped distributions, filters work slightly differently, filtering out whole groups of values at once. Thus, the type constraints on a filter operation $\mathbf{?}p$ applied to a grouped distribution $\boldsymbol{D} : \langle A \rangle^T$ are slightly different. The predicate p must have type $T \to \mathit{Bool}$ instead of $A \to \mathit{Bool}$, since the predicate is on the grouping values rather than the values in the underlying distribution. Often, a filter is performed immediately after a group, where the grouping function is the predicate and the argument to the filter operation is just $\lambda t.t = \mathit{true}$. This accentuates the effect of the filter, grouping together all of the elements that will and won't pass the filter, and presenting the sum of each groups probabilities. Section 6.5 provides an example of this explanation strategy.

The story semantics of filtering a grouped distribution are given below. Again, c accumulates the proportion of the probability space that passes the filter and is used to scale the probabilities of values in the resulting distribution.

$$\begin{aligned}
[\![\mathbf{?}p]\!] &= \lambda \boldsymbol{D}.(E, \boldsymbol{D}') \\
&\quad \text{where } \begin{aligned}[t] c &= \textstyle\sum_{(t,X) \in \boldsymbol{D}, q \in \mathit{rng}(X), p(t)} q \\
E &= \{(t,t) \mid t \in \mathit{dom}(\boldsymbol{D}), p(t)\} \cup \{(t, \bot) \mid t \in \mathit{dom}(\boldsymbol{D}), \neg p(t)\} \\
\boldsymbol{D}' &= \{(t, \{(x, q/c) \mid (x,q) \in X\},\ (t,X) \in \boldsymbol{D},\ p(t)\}
\end{aligned}
\end{aligned}$$

Edges for the grouped distribution case are similar to the plain distribution case, except connecting between groups rather than blocks.

5.5. Ungroup

An ungroup operation $\curlywedge$ removes the view imposed by a previous group operation by transforming a grouped distribution back into a plain distribution. That is, it defines the distribution transformation $\langle A\rangle^T \rightarrow \langle A\rangle$. This is almost, but not quite as simple as taking the union of the subdistributions contained in each group, as originally described in Section 4.1. The hitch is that sometimes a map operation can map values in different groups to the same value—we must merge these during the flattening process by summing their probabilities as if the map had been applied to a plain distribution. To do this, we can reuse the *map* helper function defined in Section 5.3.

In the following story semantics for the ungroup operation, we build an intermediate plain distribution of type $\langle T \times A\rangle$ in which each value x is prefixed by the value t of its group in the previous distribution. This allows us to flatten the distribution while ensuring that every value is unique. We can then attain the desired distribution of type $\langle A\rangle$ by invoking the helper function *map* with the function *snd* that returns the second value of a pair. This will combine values that were the same but in different groups in the original grouped distribution, accumulating their probabilities.

$$\begin{aligned}
[\![\curlywedge]\!] &= \lambda \boldsymbol{D}.(E,D') \\
&\quad \text{where } E = \{(t,x) \mid (t,X) \in \boldsymbol{D},\ x \in \mathit{dom}(X)\} \\
&\quad \phantom{\text{where }} D' = \mathit{map}(\mathit{snd}, \{((t,x),p) \mid (t,X) \in \boldsymbol{D},\ (x,p) \in X\})
\end{aligned}$$

We produce an edge leading to each block in the ungrouped distribution from its containing group in the grouped distribution.

An example use of the ungroup operation will be provided in Section 6.5.

5.6. Representative Example Selection

Like the group and ungroup operations, the final operation in Probula is not strictly needed for computation with probabilistic distributions (unlike the other three operations), but rather to support the creation of effective explanations. The operation $\mathbf{!}x$ means to select the value x as a *representative example* from the previous plain distribution. This operation is used when all of the cases in a distribution are isomorphic and equally probable, simplifying subsequent distributions throughout the explanation by assuming a single case rather than considering all cases simultaneously.

A typical example of this operation's use is provided in the first step of the explanation of the Monty Hall Problem in Figure 2. In the initial distribution, we represent that the prize could be hidden behind any one of the three doors, each with equal probability. The important observations are that (1) it does not really matter *which* door hides the prize and (2) the probabilities of winning or losing are the same for any one case as for the problem as a whole. We could confirm this by removing the second step of the plot and observing that the number of values in subsequent distributions would be multiplied by three (and their probabilities correspondingly divided by three), but that the final probabilities in the explananda would remain unchanged.

Given the above description, it should be obvious that there are significant constraints on when we can employ the representative example operation. Fortunately, these constraints are easy to check and enforce. Given a preceding distribution D and a plot $\mathbf{!}x \triangleright P$ where $(x,p) \in D$, we require that $\forall (y,q) \in D$, $p = q$ and the limits of the distribution graphs $[\![P]\!](\langle x^{100}\rangle)$ and $[\![P]\!](\langle y^{100}\rangle)$ are the same. The second of these constraints reveals the story semantics of the operation, given below, in which we simply create a new distribution where the value x has 100% probability.

$$\begin{aligned} [\![!x]\!] &= \lambda D.(E, D') \\ \text{where } E &= \{(y,x) \mid y \in dom(D)\} \\ D' &= \langle x^{100} \rangle \end{aligned}$$

The edges generated by a representative example step are also trivial, but we represent them in the visual concrete syntax by merging their heads and making the line from the chosen example block bold.

The representative example operation requires a somewhat deeper understanding or intuition of probabilistic reasoning to accept in isolation. In Section 6.7 we present a simple transformation for changing the particular example chosen as a representative. This can help to convince the readers of an explanation who do not immediately see the isomorphism between the cases.

5.7. Summary

The operations presented in this section can be separated into two classes of three operations each, according to their primary role in the creation of explanations of probabilistic reasoning. The generate, map, and filter operations represent probabilistic *computations*. These operations manipulate the values and probabilities in distributions directly; that is, they are primarily concerned with the computation of results essential to the explanation. The group, ungroup, and example operations are concerned instead with the *presentation* of explanations. Group and ungroup manipulate a grouping view overlaid on a distribution while the example operation reduces a set of equivalent cases to a single representative case. These operations organize the computed results in order to make the important points clear and the explanation understandable.

In Figure 5 we summarize the type information for each operation, including the types of the distribution transformations (from input to output) and of the arguments to each operation. Whether an operation is applied to a plain or grouped transformation can be determined from the type of the input distribution ($\langle A\rangle$ is plain, $\langle A\rangle^T$ is grouped). Note that there are two entries for the map and filter operations since they can be applied to either plain or grouped distributions, and their implementations, constraints, and distribution transformations differ depending on this context.

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

6. Generating Alternative Explanations

One of the most important features of Probula is the ability to algorithmically transform a single explanation, defined by an explanation creator, into many alternative, equivalent explanations of the same probabilistic reasoning problem. This allows us to bridge the gap between the two qualitatively different kinds of explanations discussed in Section 1. On one end we have explanations given directly from one person to another. Personal explanations are extremely adaptable. The explainer can answer

Name	Syntax	Input	$\rightarrow$	Output	Type Constraints
Generate	$\triangle e$	$\langle A \rangle$	$\rightarrow$	$\langle B \rangle$	$e : A \rightarrow \langle B \rangle$
Group	$\curlyvee g$	$\langle A \rangle$	$\rightarrow$	$\langle A \rangle^T$	$g : A \rightarrow T$
Map	$*f$	$\langle A \rangle$	$\rightarrow$	$\langle B \rangle$	$f : A \rightarrow B$
Map†	$*f$	$\langle A \rangle^T$	$\rightarrow$	$\langle B \rangle^T$	$f : A \rightarrow B$
Filter	$\mathbf{?}p$	$\langle A \rangle$	$\rightarrow$	$\langle A \rangle$	$p : A \rightarrow Bool$
Filter†	$\mathbf{?}p$	$\langle A \rangle^T$	$\rightarrow$	$\langle A \rangle^T$	$p : T \rightarrow Bool$
Ungroup	$\curlywedge$	$\langle A \rangle^T$	$\rightarrow$	$\langle A \rangle$	
Example	$\mathbf{!}x$	$\langle A \rangle$	$\rightarrow$	$\langle A \rangle$	$x : A^\ddagger$

†Group preserving variant.
‡See also the constraints in Section 5.6.

Figure 5: Summary of operations and their types.

questions, rephrase points, clarify assumptions, and provide alternative examples. The drawbacks of personal explanations are that they are inconsistent, time-consuming to share, and often difficult to come by in the first place. At the other end are what might broadly be called static explanations, those given in text or pictures, provided in a book or on the web. These explanations are much less adaptable but make up for it with higher consistency, availability, and shareability. In the middle, and intending to complement both, are explanation stories like those provided by Probula.

Probula explanations can be presented as simple illustrations, as throughout this paper, conferring the benefits of static explanations. But they are also highly adaptable, able to transformed through a set of laws into alternative explanations that tell the same basic story in a different way, emphasizing one point or another. In this section we will present these laws, define what it means for an explanation to tell the same basic story, and define some simple metrics for discussing the differences between tellings.

6.1. A Story and its Telling—Fabula and Sujet

Narrative is one of the most important and fundamental ways of sharing knowledge and ideas. It is the basis not only of arts like literature and cinema, but also of history, journalism, and the entire institution of academic authorship. It is the idea that information can be conveyed as a story, that the same essential story can be told in different ways, and that *how* that story is told impacts how it is perceived and understood.

The basic advantage of personal explanations can be boiled down to the fact that a personal explanation's narrative is *flexible*—it can be modified on the fly to suit the particular needs of the situation. If the same person gave an explanation of the Monty Hall Problem to three different people, she would likely tell three different stories. The important point of this section, however, is that while these stories might differ in their telling, they must also, as explanations of the same problem, share some common essence. In narratology, this point is sometimes captured in the terms *fabula*, meaning the basic facts or "raw material" of a story, and *sujet*, meaning how the story is organized and told [5]. In our scenario, the fabula of the three personal explanations is constant while the sujet is likely to change between tellings, depending on the individual needs of the listeners and other factors.

In Probula, a plot and the story it generates correspond to a particular sujet—one telling or view of a larger class of related stories with a common fabula. The laws presented in this section represent transformations from one sujet in this class to another. Through the combination and repeated application of these laws, we can view a Probula explanation not just as a single story, but as a navigable space of different stories that all explain the same basic problem.

The laws are defined as transformations of a part of a Probula plot (a *subplot*), usually translating one sequence of operations into another. To help ensure that a law preserves the underlying fabula of generated stories, we compare the *limits* of the distribution subgraph (see Section 4.4) generated by the affected subplot, before and after the transformation. If the limits are the same, we say that the transformation is *limit preserving*. While limit preservation does not capture the preservation of fabula directly, it does so indirectly by guaranteeing that a story transformation is strictly local. Specifically, a limit preserving transformation affects only the intermediate distributions produced by a subplot and nothing else in the story.

Transformation locality is important for several reasons. First, it enables the *composition* of transformation laws—new Probula explanations can be generated not only by applying the laws described in this section directly, but by composing these laws into larger transformations, broadening the space of potential explanations. Second, it allows us to consider the effect of a transformation independent of its context. In the next subsection we will define a couple of simple metrics to support the isolated analysis of a transformation and to frame its effects on the perception and understandability of its generated stories.

6.2. Measures of Story Complexity

In order to quantify the effect of an explanation transformation, we introduce the notions of the horizontal and vertical complexity of a distribution (sub)graph. These metrics are *not* intended to measure the effectiveness or understandability of an explanation directly, but to provide a way to quantify the effect that a transformation has on an explanation, to support the comparison of transformations and other discussion.

The *vertical complexity* of a (sequential) distribution subgraph is simply the number of steps it contains. The *horizontal complexity* of a subgraph is the total number of edges it contains divided by the number of steps. For example, consider the subgraph corresponding to the first three generate steps in the explanation of the three coins problem in Figure 1. The vertical complexity of this subgraph is 3, while the horizontal complexity is $(2+4+8)/3 = 4.67$. Note that the merging of the heads and tails of edges in the concrete syntax has no effect on the computation of the horizontal complexity of a subgraph.

Vertical complexity intends to capture the intuition that the length of an explanation affects its understandability. A longer explanation presents more individual steps that must be examined and understood, and increases the cognitive load when considering the explanation as a whole. Horizontal complexity represents instead the average difficulty of understanding each step in the explanation. The reasoning is that steps with

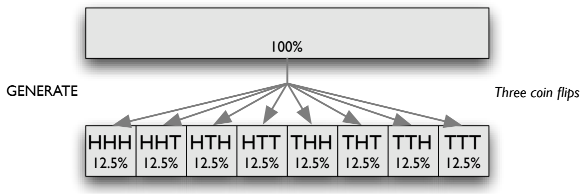

Figure 6: The three generators from Figure 1 fused into one.

more edges require more effort for readers to untangle and accept. Again, however, neither of these metrics are intended to absolutely quantify understandability—we cannot take a graph with a vertical complexity of 4 and conclude that it is easier to understand than an unrelated graph with a vertical complexity of 5 (even if their horizontal complexities are identical). Instead we use complexity to discuss the *relative* understandability of graphs related through a transformation law.

As we will see in the rest of this section, most interesting transformations present a trade-off between the vertical and horizontal complexity of an explanation. This means that there is usually not a unique "best" explanation to be achieved; different explanations fill different roles and have different benefits and drawbacks. Using the transformation laws presented below we can move between these different explanations. This not only enables users to get the particular static explanations that work best for them, but also to explore the explanation space and view many alternative explanations for the same problem, increasing the value of Probula explanations further.

6.3. Operation Fusion

The first class of transformations we will consider follows from the observation that often adjacent operations of the same kind can be merged by combining the effects of their argument functions. We call this *operation fusion*. For example, consider again the first three generate steps in the example in Figure 1. In Figure 6 we show how these can be fused into a single generate step that represents the flipping of all three coins at once. Adjacent generators, maps, and filters can all be merged. In this subsection we will consider each of these in turn.

Recall that a plot transformation is considered valid if it preserves the limits of the distribution subgraph generated by the affected subplot. When fusing two adjacent generators $\triangle e_1$ and $\triangle e_2$, we must therefore identify an event generating function e_3 such that if $[\![\triangle e_1 \triangleright \triangle e_2 \triangleright P]\!](D_0) = (D_0, E_1) \triangleright (D_1, E_2) \triangleright (D_2, E_3) \triangleright G$, then $[\![\triangle e_3 \triangleright P]\!](D_0) = (D_0, E_2') \triangleright (D_2, E_3) \triangleright G$. Such a function is defined in the following theorem for fusing adjacent generate steps. Note that, throughout this section, we use the symbol $\rightsquigarrow$ to indicate a directed, limit-preserving plot transformation and we omit the common subsequent plot P from the notation. That is, we write a plot transformation $o_1 \triangleright \ldots \triangleright o_j \triangleright P \rightsquigarrow o_1' \triangleright \ldots \triangleright o_k' \triangleright P$ more concisely as $o_1 \triangleright \ldots \triangleright o_j \rightsquigarrow o_1' \triangleright \ldots \triangleright o_k'$.

Theorem 1. *Generator Fusion*
$\triangle e_1 \triangleright \triangle e_2 \rightsquigarrow \triangle \lambda x. \{(z, qr) \mid (y, q) \in e_1(x),\ (z, r) \in e_2(y)\}$

Any number of adjacent generators can be fused by repeatedly applying this theorem.

That generator fusion is limit preserving follows directly from the story semantics of plots from Section 4.4 and of generate operations in Section 5.1. The initial distribution D_0 is trivially preserved. The preservation of D_2 can be confirmed by computing and comparing the semantics of the LHS and RHS of the transformation. For the LHS, we first compute the intermediate distribution D_1 and use that to compute the semantics of D_2, as follows.

$$\begin{aligned} D_1 &= [\![\triangle e_1]\!](D_0) = \{(y, pq) \mid (x, p) \in D_0,\ (y, q) \in e_1(x)\} \\ D_2 &= [\![\triangle e_2]\!](D_1) = \{(z, pqr) \mid (y, pq) \in D_1,\ (z, r) \in e_2(y)\} \end{aligned}$$

For the RHS we compute D_2 directly.

$$\begin{aligned} D_2 = [\![\triangle e_3]\!](D_0) &= \{(z, pqr) \mid (x, p) \in D_0,\ (z, qr) \in e_3(x)\} \\ \text{where } e_3 &= \lambda x.\{(z, qr) \mid (y, q) \in e_1(x),\ (z, r) \in e_2(y)\} \end{aligned}$$

Since the two resulting distributions are also identical, the transformation is limit preserving. We will not work through the limit-preservation proofs of all the transformations as verbosely, but they can all be confirmed likewise by computing and comparing the story semantics.

By fusing two generate steps into one, generator fusion trivially reduces the vertical complexity of a story by one. To compute the change in horizontal complexity, we first observe that the number of edges at any one generate step is equal to the number of values in the produced distribution (D_2 above). For example, in Figure 1, the sizes of the first three distributions after the initial distribution are 2, 4, and 8, and these are also the number of edges in each of the three generate steps. Therefore, the horizontal complexity of the story produced by the LHS of Theorem 1 is $(|D_1| + |D_2|)/2$, while the horizontal complexity of the RHS is $|D_2|$. Additionally, we know that $|D_2| \geq |D_1|$, since generators can only add values to a distribution. Therefore, the horizontal complexity of the RHS is greater than (or equal to) that of the LHS by $(|D_2| - |D_1|)/2$.

In qualitative terms, the trade-off between vertical and horizontal complexity with generator fusion is a trade-off between many small steps that incrementally build up a complex distribution or fewer larger steps that do the same. We expect that the explanatory value of each view is not constant throughout the process of reading and understanding an explanation. When an explanation is first presented, it might be helpful to see how a complex distribution is produced from smaller steps. Once this aspect is understood, however, the additional generators become visual noise that distract from the more subtle and interesting parts of an explanation. Through generator fusion, the user can take advantage of both views—seeing the more verbose explanation initially and fusing generators together as they are understood.

Like generators, adjacent maps and filters can also be fused. For maps, we use the composition of the two mapped functions as the argument to the new map operation.

Theorem 2. *Map Fusion*
$*f_1 \triangleright *f_2 \rightsquigarrow *(f_2 \circ f_1)$

We fuse adjacent filters by filtering with the conjunction of their predicates.

Theorem 3. *Filter Fusion*
$\mathbf{?}p_1 \triangleright \mathbf{?}p_2 \rightsquigarrow \mathbf{?}\lambda x.p_1(x) \wedge p_2(x)$

That these transformations are limit preserving can be confirmed in the same way as generator fusion, by expanding the story semantics of both sides of the transformations and comparing. As with generator fusion, these transformations trivially reduce the vertical complexity of a story by one and increase (though again, not strictly) the horizontal complexity of the explanation.

The number of edges in a filter or map step is determined not by the size of the produced distribution, as with generators, but rather by the size of the *input* distribution. Assume a similar D_0 and D_2 as above, for representing the identical limits of the generated distribution graphs before and after map or filter fusion, and a similar D_1 for representing the intermediate distribution produced by the plot on the LHS only. Then the horizontal complexity of the graph before each transformation is $(|D_0| + |D_1|)/2$ while the horizontal complexity after each transformation is $|D_0|$. In this case, we have the partial inequality $|D_0| \geq |D_1|$, since maps and filters can only *remove* values from distributions. So the horizontal complexity after each transformation is greater than before by $(|D_0| - |D_1|)/2$. These differences present a similar trade-off for explanation consumers as generator fusion.

Note that given a suitable representation, it is also possible to *split* previously fused generators, maps, or filters. However, such transformations are not possible in general (that is, on operations that were not previously fused) unless we impose further restrictions on these operations, making their arguments somehow decomposable.

6.4. Operation Commutation

In addition to fusing adjacent operations of the same kind, many operations can be commuted with each other. These transformations present less clear explainability trade-offs than operation fusion (or the transformations we will look at in the next subsection) but they are often useful as preparatory steps to setup other transformations. Therefore, we will present some commutation transformations here, relatively briefly.

Since filters do not affect the types of the distributions they modify, they can be freely commuted with each other. Note that we use $\leftrightsquigarrow$ to indicate a limit-preserving, *undirected* plot transformation.

Theorem 4. *Filter-Filter Swap*
$\mathbf{?}p_1 \triangleright \mathbf{?}p_2 \leftrightsquigarrow \mathbf{?}p_2 \triangleright \mathbf{?}p_1$

That this transformation preserves the limits of the corresponding distribution subgraph is obvious—in either case, the resulting distribution contains only the values that pass both predicates. Observe that this theorem can also be viewed as a corollary of Theorem 3 (Filter Fusion) since applying that transformation to both $\mathbf{?}p_1 \triangleright \mathbf{?}p_2$ and $\mathbf{?}p_2 \triangleright \mathbf{?}p_1$ will result in the same fused filter operation.

Like all commutation transformations, filter swapping has no effect on the vertical complexity of the produced graph; no operations are added or removed, they are simply rearranged. The transformation's effect on horizontal complexity depends on the specific predicates p_1 and p_2. If fewer values in D_0 pass p_1 than p_2, the graph produced by the LHS will have a lower horizontal complexity, while the opposite will be true if p_2

filters more values out of D_0 than p_1. Although the transformation impacts horizontal complexity inconsistently, swapping the order of two filters might still help a user to understand the exact effect that each filter has on the resulting distribution. Once these are understood, the filters might be fused with Theorem 3.

Unlike filters, maps do affect the types of their argument distributions, meaning that maps cannot be as freely commuted as filters. However, often we can push a map $*f$ below another operation o by composing f with the function associated with o. In other words, we can perform the mapping described by f implicitly within o, in order to delay the actual map step. These transformations should be used judiciously since they can obscure the original meaning of o, but they are safely limit preserving and may be useful to setup other transformations.

The following theorem provides a limit-preserving transformation that pushes a map operation below a filter operation.

Theorem 5. *Map-Filter Swap*
$*f \triangleright \mathbf{?}p \leadsto \mathbf{?}(p \circ f) \triangleright *f$

Assuming that the input distribution to this subplot is $D_0 : \langle A \rangle$, then $f : A \to B$ and $p : B \to Bool$. By composing p and f, we change the type of the predicate passed to the filter to $A \to Bool$, allowing it to be applied directly to D_0. The intermediate distribution of the subgraph produced by the LHS will have type $\langle B \rangle$, while on the RHS it will have type $\langle A \rangle$. The final distribution graphs will be the same, however, and be of type $\langle B \rangle$.

To illustrate map-filter swapping, consider the following plot *posTails*. Recall from Sections 5.3 and 5.4 that the function $countTails : C^* \to Int$ returns the number of tails in a sequence of coin values. The function $isPos : Int \to Bool$ returns whether its argument is positive or not.

$$posTails = *countTails \triangleright \mathbf{?}isPos \triangleright \varepsilon$$

This plot, given an initial distribution of coin sequences, produces an explanation of the number of tails in each sequence that contains at least one tails. For example, given the initial distribution $\langle HH^{25}, HT^{25}, TH^{25}, TT^{25} \rangle$, the map step produces the intermediate distribution $\langle 0^{25}, 1^{50}, 2^{25} \rangle$, and the filter step produces the final distribution $\langle 1^{66.7}, 2^{33.3} \rangle$. This is exactly the running example from Sections 5.3 and 5.4.

Applying the map-filter swap transformation results in the following alternative plot $posTails'$.

$$posTails' = \mathbf{?}(isPos \circ countTails) \triangleright *countTails \triangleright \varepsilon$$

Given the same initial distribution, the filter step eliminates the *HH* sequence, producing the intermediate distribution $\langle HT^{33.3}, TH^{33.3}, TT^{33.3} \rangle$, and the map step produces the same final distribution $\langle 1^{66.7}, 2^{33.3} \rangle$.

Map-filter swapping decreases the horizontal complexity of the produced graph since moving the filter up can only decrease the number of edges produced by the map step. However, this is a case where horizontal complexity doesn't tell the whole story since the predicate itself is now more complicated.

Using the same strategy as Theorem 5, we can also push maps below group operations. This transformation is captured in the following theorem.

Theorem 6. *Map-Group Swap*
$*f \triangleright \curlyvee g \rightsquigarrow \curlyvee(g \circ f) \triangleright *f$

As with map-filter swapping, this transformation will produce a graph with a dubious decrease in horizontal complexity.

To illusrate map-group swapping, consider the following variant of our previous example, where we use the *isPos* function to group, rather than filter the intermediate distribution.

$$\mathit{posTailsGrp} = *\mathit{countTails} \triangleright \curlyvee \mathit{isPos} \triangleright \varepsilon$$

Given the initial distribution $\langle HH^{25}, HT^{25}, TH^{25}, TT^{25}\rangle$, the map step produces the intermediate distribution $\langle 0^{25}, 1^{50}, 2^{25}\rangle$, and the group step produces the grouped distribution $\{(\mathit{false}, \{0^{25}\}), (\mathit{true}, \{1^{50}, 2^{25}\})\}$, which has type $\langle C^{*}\rangle^{\mathit{Bool}}$.

Applying the map-group swap transformation to the *posTailsGrp* plot yields the following equivalent plot.

$$\mathit{posTailsGrp}' = \curlyvee(\mathit{isPos} \circ \mathit{countTails}) \triangleright *\mathit{countTails} \triangleright \varepsilon$$

If we apply this to the same initial distribution, we get the following intermediate distribution $\{(\mathit{false}, \{HH^{25}\}), (\mathit{true}, \{HT^{25}, TH^{25}, TT^{25}\})\}$, which will map to the same resulting grouped distribution as we obtained with *posTailsGrp*.

Not all pairs of operations containing a map can be swapped, however. Recall from Section 5.3 that the map operation merges values only locally within groups. For example, if we apply the operation **countTails* to the grouped distribution $\{(\mathit{false}, \{HH^{25}, TH^{25}\}), (\mathit{true}, \{HT^{25}, TT^{25}\})\}$, we get the grouped distribution $\{(\mathit{false}, \{0^{25}, 1^{25}\}), (\mathit{true}, \{1^{25}, 2^{25}\})\}$, in which the two blocks that map to 1 are not merged since they are in different groups. This means that although we can lift a group above a map by simulating the map in the group operation, as in Theorem 6, we cannot lift an arbitrary map above a group, since it could result in blocks being merged in the intermediate distribution. For the same reason, we cannot swap map and ungroup operations.

Finally, since filters on grouped distributions affect groups rather than their contained values (see Section 5.4), we also cannot commute filter and ungroup operations.

6.5. Advanced Transformations

In this subsection we move from the simple combining and rearranging of operations to more subtle and complex transformations. Like the fusion operations, the laws presented below pose interesting trade-offs for explainability.

The first transformation we will consider involves a process called *filter lifting* and is demonstrated by an alternative explanation for the three-coins problem shown in Figure 7. The basic idea is that if all of the descendants of some block in a distribution D_i (anywhere in a distribution graph) are eliminated by a downstream filter, then we can introduce a new filter immediately below D_i that filters that block out. In the three-coins example, by referring to the original explanation in Figure 1, we can see that all of the descendants of the *TT* block in the D_2 distribution are eliminated by the filter two steps below. In the transformed explanation in Figure 7, we introduce a new filter immediately after this value is generated, filtering it and its descendants out

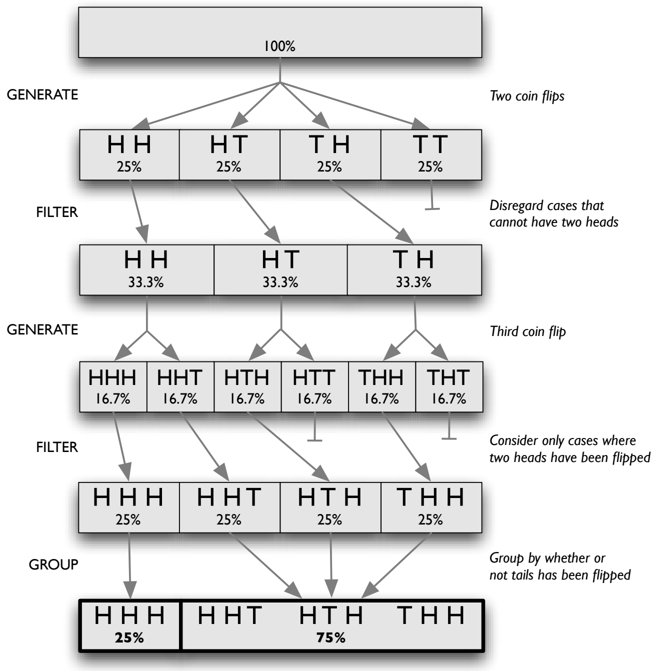

Figure 7: Example of filter lifting.

two steps early (note that we have also fused the first two generators). We capture this transformation in the following theorem.

Theorem 7. *Filter Lifting*
Given preceding distribution D, if $\forall y$ *descended from* $x \in dom(\llbracket \triangle e \rrbracket(D))$, $\neg p(y)$, *then* $\triangle e \triangleright \ldots \triangleright \mathbf{?}p \rightsquigarrow \triangle e \triangleright \mathbf{?}\lambda z.x \neq z \triangleright \ldots \triangleright \mathbf{?}p$

Filter lifting increases the vertical complexity of a distribution graph by one, but offers the potential to significantly reduce the horizontal complexity. That filter lifting always decreases horizontal complexity by some amount is easy to see—the number of edges at any step in a story is a function of the size of one of its adjacent distributions, and every distribution between $\triangle e$ and $\mathbf{?}p$ will be strictly smaller after filter lifting. Exactly how much the horizontal complexity decreases depends on how many steps are between the generator and the filter and how many new values are generated from the eliminated value x. For the three-coins example, filter lifting reduces the horizontal complexity of the relevant subgraph of the explanation from $(8+8)/2 = 8$ to $(4+6+6)/3 = 5\frac{1}{3}$. This reduction in horizontal complexity reflects the observation, encoded in the transformed explanation, that values that will eventually be eliminated are in some sense irrelevant. By introducing an additional step to eliminate these values immediately after they arise, we can invest a little explanatory effort early to avoid the effort of considering their descendants over and over, at each step.

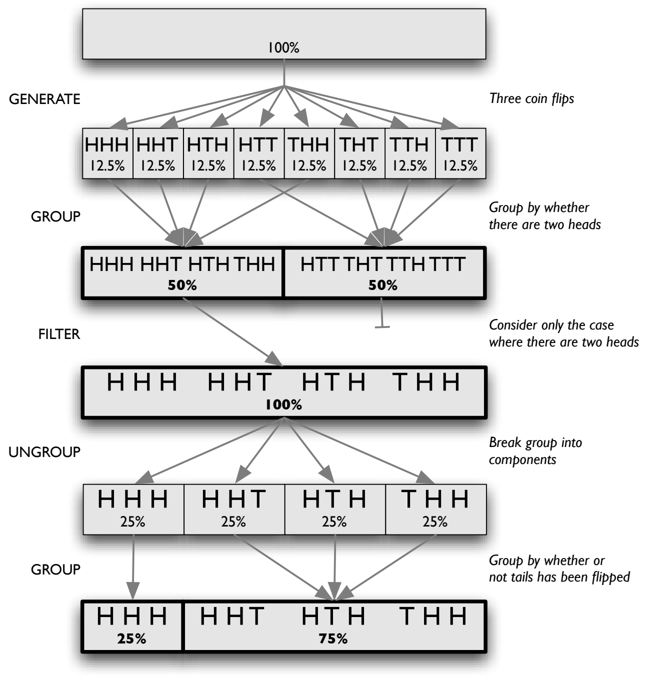

Figure 8: Example of group bracketing.

Note that we can also preemptively eliminate several such irrelevant values from a single distribution with just one filter by alternating applications of Theorem 7 (filter lifting) and Theorem 3 (filter fusion).

The next two transformations we present can be used to emphasize a particular step in an explanation's story, drawing attention to it and making its effects more explicit. We do this by introducing a group-ungroup pair around the step we wish to emphasize, using a process called *group bracketing*.

First we consider the bracketing of a filter operation. This is demonstrated in Figure 8, where we have bracketed the filter step in the three coins explanation from Figure 1. When bracketing a filter, the predicate of the filter is used as the grouping function, producing a distribution divided into two groups—those that will pass the filter and those that will fail. The filter step is then greatly simplified, eliminating one group and passing the other on as the subsequent distribution, which is then ungrouped.

Group bracketing of a filter operation is captured formally in the following theorem. Note that we cannot bracket a filter if we are already in the context of a group operation since groups cannot be nested. Also, recall from Section 5.4 that filters on grouped distributions are defined in terms of the grouping value rather than the underlying elements, so we change the filter predicate from p to $\lambda t.t = \mathit{true}$ (equivalently, the identity function) since each group will be represented by the result of p applied to each of its contained elements.

Theorem 8. *Group Bracketing of a Filter*
If not already in the context of a group operation, then
$\mathbf{?}p \rightsquigarrow \curlyvee p \triangleright \mathbf{?}\lambda t.t \triangleright \curlywedge$

Group bracketing increases the vertical complexity of a story by two, adding a group and ungroup operation. It also decreases the horizontal complexity. If D and D' represent their common limits, with $|D| \geq |D'|$, the horizontal complexity of the unbracketed filter is $|D|$; for the bracketed version, there are $|D|$ edges for the group step, 2 edges for the filter step, and $|D'|$ edges for the ungroup step, for a horizontal complexity of $(|D| + 2 + |D'|)/3$, which is clearly less than $|D|$.[5] The qualitative trade-off is that we accentuate and simplify the filter step at the cost of the additional complexity introduced by the new group and ungroup steps. Through bracketing, a user can see all of the values that either pass or fail a filter, together and at a glance, rather than having to scan the outgoing edges of all values in the unbracketed filter step.

Similarly, we can also bracket a map operation in a group-ungroup pair. In this case, we use the mapped function first as a grouping function (assuming the function maps to a type that can be checked for equality), then apply the map, then ungroup.

Theorem 9. *Group Bracketing of a Map*
Given $f : A \to B$, if B can be checked for equality, and if not already in the context of a group operation, then
$*f \rightsquigarrow \curlyvee f \triangleright *f \triangleright \curlywedge$

This operation offers little benefit if f is one-to-one, increasing the vertical complexity by two without changing the horizontal complexity. However, if f maps several values in D to the same value in D', the benefit is similar to bracketing filters. The horizontal complexity will be decreased, and values in D that map to the same value in D' will be put in the same group, making it easier to see which values in D will be merged in D'.

6.6. Transforming Branched Plots

Throughout this section we have considered only the transformation of sequential plots. Since transformations are local, these trivially apply also to branched plots as long as the transformed subplot is entirely before the branch point or contained within a particular branch. This view is overly strict, however, ruling out many seemingly valid transformations that span branch points. For example, the subplot $*f_1 \triangleright (*f_2 \vdots *f_3)$ is clearly equivalent (in a limit preserving sense) to the subplot $*(f_2 \circ f_1) \vdots (*f_3 \circ f_1)$. In the second we have fused $*f_1$ with $*f_2$ in the left branch and $*f_1$ with $*f_3$ in the right branch. We can't do this by applying Theorem 2 directly, however, because neither pair of map operations are in direct sequence—the branch point gets in the way.

Rather than extend each transformation law to account for branching, we instead introduce a new law that can be composed with existing laws like map fusion to achieve the desired effect. We call the new transformation *branch unzipping*, since it visually resembles the unzipping of a zipper, and in Figure 9 we demonstrate its use as a

[5] Actually there are two degenerate cases where this is not true. If $|D| = |D'| = 2$ or if $|D| = |D'| = 1$, then group bracketing preserves or increases the horizontal complexity, respectively.

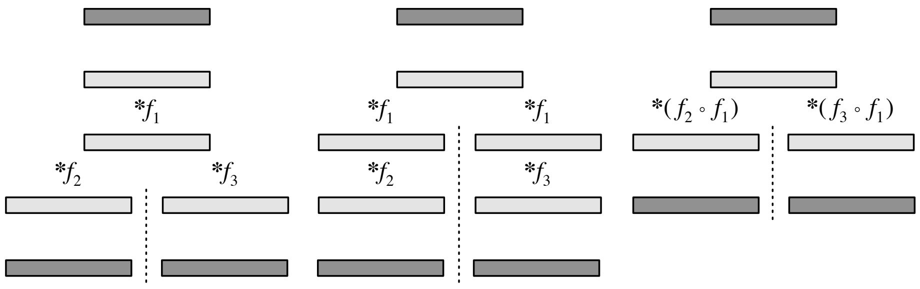

Figure 9: Example of unzipping a branching story to enable operation fusion.

preparatory step for fusing the maps in the above example. In this figure we show an abstracted view of three stories, with the relevant steps annotated by their operations, and the distributions in the relevant subgraphs colored a lighter gray (the darker distributions are provided to give a sense of context). The leftmost story corresponds to the story generated by our initial subplot, $*f_1 \triangleright (*f_2 \,\vdots\, *f_3)$. In the second story we unzip one level of the story, pushing $*f_1$ out of the trunk and duplicating it in each branch. Finally, in the third story, we have applied map fusion twice, once to each branch, corresponding to our final subplot $*(f_2 \circ f_1) \,\vdots\, (*f_3 \circ f_1)$.

The dual of branch unzipping is *branch zipping*, which allows us to lift a duplicated operation at the top of each branch into the trunk of an explanation. We capture both of these transformations in the following theorem.

Theorem 10. *Branch Unzipping/Zipping*
$o \triangleright (P_L \,\vdots\, P_R) \leftrightsquigarrow o \triangleright P_L \,\vdots\, o \triangleright P_R$

Recall from Section 4.4 that the limits of a distribution subgraph G containing branches are a pair of the initial distribution in G and a sequence of the final distribution in each branch. That branch (un)zipping preserves the limits of the generated subgraph follows directly from an application of the story semantics of plots (also Section 4.4) to the LHS and RHS of above the law.

To analyze the complexity impact of these transformations, we must extend our definitions of vertical and horizontal complexity to branched stories. Branching represents complexity mostly on the horizontal axis. Therefore, we consider the vertical complexity of a branched story to be simply the average of the number of steps along each path in the story, while for horizontal complexity we sum the edges across all steps, in all branches, and divide by the (averaged) vertical complexity.

Using these definitions, we see that neither transformation has any effect on the vertical complexity of a story. Meanwhile, zipping decreases horizontal complexity proportional to the number of edges generated by o, while unzipping increases horizontal complexity by the same amount. As the example above demonstrates, however, the complexity introduced by unzipping may be temporary when used as a preparatory step for other transformations.

Finally, note that although we applied map fusion to both branches after unzipping the above example, there is no requirement that we do so. For example, we can unzip

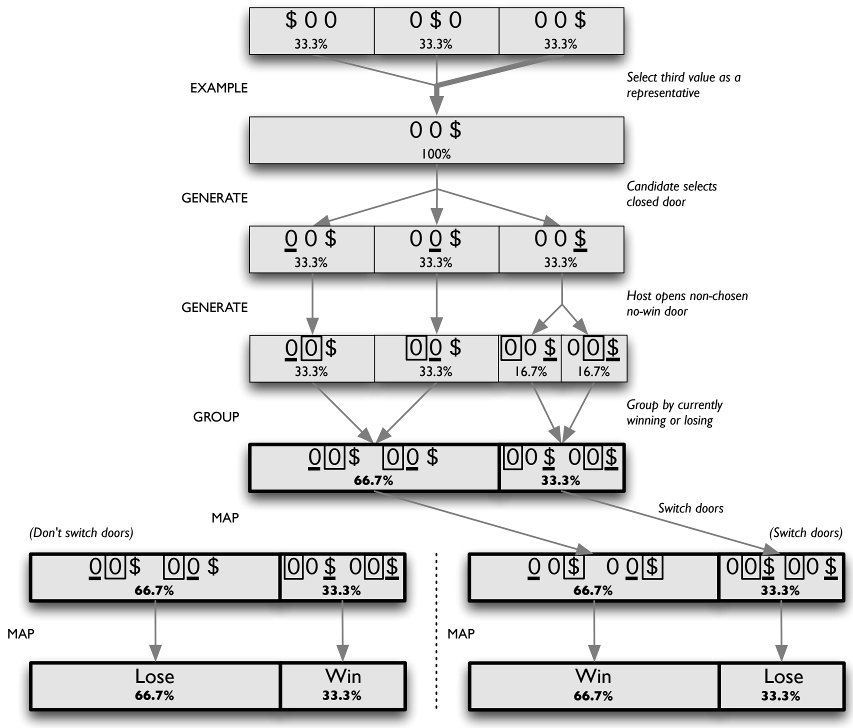

Figure 10: Monty Hall Problem explained with a different representative example.

the subplot $*f_1 \triangleright (*f_2 : o_3)$ (where o_3 is presumably not a map) to $(*f_1 \triangleright *f_2) : (*f_1 \triangleright o_3)$, then apply map fusion only to the left branch to produce $*(f_2 \circ f_1) : (*f_1 \triangleright o_3)$. In this case we have decreased vertical complexity by only half a step, instead of the full step decrease in the original example, while also increasing horizontal complexity.

6.7. Changing the Representative Example Selection

The final law we consider follows from the discussion in Section 5.6. Recall that the operation $!x$ indicates that x is a *representative example* of the preceding distribution D, and that we can therefore replace D with the distribution $\langle x^{100} \rangle$. A representative value x is one that is both isomorphic to, and equally probable as, every other value y in D. This operation is used as the first step of the Monty Hall Problem shown in Figure 2, in which we choose the case where the prize is hidden behind the left door as a representative example.

Since x is isomorphic to all such values y, we can freely replace $!x$ with $!y$ without changing the meaning of the explanation. For example, in the Monty Hall Problem we can equally well choose the case where the prize is behind the center door or the right door. Figure 10 shows an alternative explanation of the Monty Hall Problem in which we have chosen a different representative example. This transformation is captured generally in the following law.

Theorem 11. *Select Representative Example*
Given $x \in dom(D_0)$ *and* $y \in dom(D_0)$*, then*
$!x \leftrightsquigarrow !y$

Since all values must be isomorphic as a precondition to the use of the representative example operation, the limits are trivially preserved by this transformation up to a similar isomorphism in the result distribution.

While changing the representative example has no effect on the complexity of the story, it is an important transformation for supporting the understanding of explanations that use this rather subtle operation. It enables explanation readers to confirm for themselves that the particular representative example chosen does not determine the solution, and to see how different examples produce slightly different distributions but an overall similar structure.

Effective explanations must necessarily be given from a particular perspective, using particular examples, and telling a particular story. When constructing a static explanation, the explanation creator must fix these aspects once and for all. A personal explainer can be more flexible, changing their explanation during creation, as needed by a particular explanation consumer. The transformation laws provided in this section help Probula span this gap, enabling an initially fixed static explanation to be automatically adapted to tell new stories about the same problem, with different examples and from different perspectives.

7. Related Work

Much of the most pertinent related work has been discussed already in Section 2 on the explanation-oriented programming paradigm, and Section 3 on the story-telling model of explanation. In this section we recall this discussion, filling in the gaps, and discuss several other areas of related work as well.

The story-telling model of explanation follows in the tradition of more general ideas from the philosophy of science that emphasize the importance of causality in explanations [7, 35, 19]. The philosophical study of causation is in turn a large area of research, which we have discussed briefly in Section 3, and at greater length in our previous work [12, 14]. Of particular interest are the various visual notations that have been used in this research for representing graphs of causally related events [16, 27, 37]. These causal graph notations are usually used as informal diagramming tools, to explain causal situations and support discussion, but are rarely defined as formal visual languages. Probula differs from these languages in two important ways. First, it is mainly linear, a design decision that we discussed in Section 3. Second, it is typed and much more structured. Nodes in causal graphs sometimes represent events and sometimes states, and this ambiguity often leads to misunderstandings and obscures subtle distinctions in relationships between nodes [18]. In contrast, points in a Probula story are always distributions of typed values, and relationships are described in terms of a small set of well-defined operations with explicit type constraints. This makes the meaning of each point and each step in the story much clearer than in less structured representations.

Throughout the paper we have given different explanations in Probula of the Monty Hall problem and the three coins problem. The fallacies induced by these two problems are part of a large class of fallacies that arise from a misunderstanding of statistical independence and conditional probability. The *gambler's fallacy* and the *hot hand fallacy* are two contradictory fallacies related to independence [2]. Given a sequence of heads results when flipping a fair coin, the gambler's fallacy expects a higher probability of the next flip being tails, while the hot hand fallacy expects the opposite. The *inverse fallacy* is the false assumption that $P(A|B)$ is approximately equal to $P(B|A)$ [34]. Relatedly, the *base-rate fallacy* is the failure to consider the underlying probability of an event when given some conditional evidence [3]. For example, consider a population in which 1 in 10 people have a virus, and a test for that virus that is accurate 90% of the time. If a random person tests positive for the virus, the fallacy is that they are 90% likely to have the virus. This fails to take into account the lower probability of having the virus in the first place, and thus that there will be more false positives than false negatives. In fact, the odds of the person having the virus, given that they tested positive, is 50%. These fallacies are widespread not only among laypeople, but among professions where an expert understanding of conditional probability is critical, such as clinicians [8]. Probula explanations can help to debunk these fallacies by providing explanations that show how the correct (though perhaps counterintuitive) answer is derived.

In Section 2 we discussed our previous work on computing with probability distributions [10] and supporting and explaining probabilistic reasoning [12, 13]. Although there are other languages that support *computation* with probabilistic values, such as IBAL [29] and the OCaml DSEL designed by Kiselyov and Shan [23], 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 [22], where many ideas and approaches have been developed to illustrate the working of algorithms through custom-made or (semi-)automatically generated animations [21]. 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 [4].

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 a bug. 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 [25] 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 the basic idea of the WHYLINE to 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]. From a philosophical perspective, the generated change suggestions represent counterfactual statements about the state of the spreadsheet [27]. Counterfactual reasoning as a basis for explanation is closely related to the story-telling metaphor [12].

Probula, and the story-telling model in general, are also related to the idea of dataflow languages, in which data is incrementally modified by passing through a directed graph of operations [20]. Our language could be viewed as a dataflow language with the single, parametrically-polymorphic data type of probability distributions (plain or grouped), with the operations described in Section 5 and an additional operation to support branching.

Finally, the idea of elevating explainability as a design criterion for languages was first proposed in [11] where we have presented a visual language for expressing strategies in game theory. The guiding principle of the design of this language was the concept of *traceabilty*. This is the idea that language designers should consider not only how to represent programs, but also how to represent the *execution* of programs (program traces), and how to relate programs to their traces in order to support program understanding. This idea developed into the notion of explanation-oriented programming as described in this paper, and Probula reflects these original design ideas well. Probula's visual notation is an integrated representation of the execution of a Probula program, as the distributions that it generates, combined with a representation of the operations that produce that execution.

8. Conclusions and Future Work

We have presented Probula, a domain-specific, visual language for explaining probabilistic reasoning problems. This language continues our research in the new paradigm of explanation-oriented programming, where the focus of programming is not on the computation of results, but on explaining how and why those results were computed. We believe that this shift of focus reveals a new, fruitful, and under-explored area for language design. Domain-specific and visual languages are especially well-suited for explanation orientation for several reasons.

Domain-specific languages allow us to express explanations in terms and structures appropriate to the domain; for example, in Probula, we use probabilistic distributions as a basic construct in explanations of probabilistic reasoning problems. This makes explanations in the language more understandable by making them more concrete and by taking advantage of users' existing domain knowledge. It also makes the explanations more useful and externally applicable since they can be used not only to understand the specific problems presented, but as tools to better understand the domain itself. This is one of the primary design goals of Probula.

Visual languages are a powerful medium for explanation since we have many more potential dimensions for encoding information than is available in textual languages (position, color, length, shape, etc.). This not only gives us more flexibility and expressiveness in the design of explanations, but also makes it easier to directly reuse existing notations from the domain, when applicable. Furthermore, it gives us access to a rich language of visual metaphors and symbols. We have presented a few such examples in Probula, such as the use of spatial partitioning to encode probability, and connection to represent the flow of data.

A significant contribution of this work is the formulation of laws which can be used to transform an explanation into many equivalent explanations of the same problem. In this way we can combine the accessibility benefits typical of static explanations with the adaptability benefits of personal explanations. Through the composition of explanation transformations, we can derive from a single explanation a large, explorable space of related explanations.

In order to fully realize the adaptability benefits of personal explanations, however, we must also consider how users interact with this explanation space. This is an important topic for future work. In the most straightforward view, users might directly manipulate an explanation within some tool, in order to generate related alternatives—manually fusing generators, for example, or changing the representative example selection. In the context of personal explanations, direct manipulation corresponds to responding to a user's question. But personal explainers also adapt automatically to a user's needs, when they sense that the listener is confused, for example. Therefore, we might also consider developing heuristics to determine when a particular transformation will be most useful for a particular user, which the tool can suggest or apply automatically.

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] P. Ayton and I. Fischer. The Hot Hand Fallacy and the Gambler's Fallacy: Two Faces of Subjective Randomness? *Memory & Cognition*, 32:1369–1378, 2004.

[3] M. Bar-Hillel. The Base-Rate Fallacy in Probability Judgments. *Acta Psychologica*, 44(3):211–233, 1980.

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

[5] P. Cobley. Narratology. In M. Groden, M. Kreiswirth, and I. Szeman, editors, *The Johns Hopkins Guide to Literary Theory and Criticism, 2nd ed.* John Hopkins University Press, London, 2005.

[6] P. Dowe. *Physical Causation*. Cambridge University Press, Cambridge, UK, 2000.

[7] M. Dummett. Bringing About the Past. *Philosophical Review*, 73:338–359, 1964.

[8] D. M. Eddy. Probabilistic Reasoning in Clinical Medicine: Problems and Opportunities. *Judgment Under Uncertainty: Heuristics and Biases*, pages 249–267, 1982.

[9] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

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

[11] 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.

[12] M. Erwig and E. Walkingshaw. A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conference on Domain-Specific Languages*, LNCS 5658, pages 335–359, 2009.

[13] M. Erwig and E. Walkingshaw. Visual Explanations of Probabilistic Reasoning. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 23–27, 2009.

[14] M. Erwig and E. Walkingshaw. Causal Reasoning with Neuron Diagrams. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2010.

[15] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[16] J. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach, Part I: Causes. *British Journal of Philosophy of Science*, 56(4):843–887, 2005.

[17] E. C. R. Hehner. Probabilistic Predicative Programming. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 169–185. Springer Berlin/Heidelberg, 2004.

[18] C. Hitchcock. What's Wrong with Neuron Diagrams? In J. K. Campbell, M. O'Rourke, and H. Silverstein, editors, *Causation and Explanation*, pages 69–92. MIT Press, Cambridge, MA, 2007.

[19] P. Humphreys. *The Chances of Explanation*. Princeton University Press, Princeton, NJ, 1989.

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

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

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

[23] O. Kiselyov and C. Shan. Embedded Probabilistic Programming. In *IFIP Working Conference on Domain-Specific Languages*, LNCS 5658, pages 360–384, 2009.

[24] P. Kitcher. Explanatory Unification and the Causal Structure of the World. In P. Kitcher and W. Salmon, editors, *Scientific Explanation*, pages 410–505. University of Minnesota Press, Minneapolis, MN, 1989.

[25] 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.

[26] C. Morgan, A. McIver, and K. Seidel. Probabilistic Predicate Transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

[27] J. Pearl. *Causality: Models, Reasoning and Inference (2nd ed.)*. Cambridge University Press, Cambridge, UK, 2009.

[28] N. Pennington and R. Hastie. Reasoning in Explanation-Based Decision Making. *Cognition*, 49:123–163, 1993.

[29] 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.

[30] D. Ruben. *Explaining Explanation*. Routledge, London, UK, 1990.

[31] W. Salmon. *Scientific Explanation and the Causal Structure of the World*. Princeton University Press, Princeton, NJ, 1984.

[32] W. Salmon. Causality without Counterfactuals. *Philosophy of Science*, 61:297–312, 1994.

[33] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[34] G. Villejoubert and D. Mandel. The Inverse Fallacy: An Account of Deviations from Bayes's Theorem and the Additivity Principle. *Memory & Cognition*, 30:171–178, 2002.

[35] G. von Wright. *Explanation and Understanding*. Cornell University Press, Ithaca, NY, 1971.

[36] E. Walkingshaw and M. Erwig. A DSEL for Studying and Explaining Causation. In *IFIP Working Conference on Domain-Specific Languages*, 2011. To appear.

[37] J. Woodward. *Making Things Happen*. Oxford University Press, New York, NY, 2003.