Probabilistic Functional Logic Programming

Sandra Dylus[1], Jan Christiansen[2], and Finn Teegen[1]

[1] University of Kiel
{sad,fte}@informatik.uni-kiel.de
[2] Flensburg University of Applied Sciences
jan.christiansen@hs-flensburg.de

Abstract. This paper presents *PFLP*, a library for probabilistic programming in the functional logic programming language Curry. It demonstrates how the concepts of a functional logic programming language support the implementation of a library for probabilistic programming. In fact, the paradigms of functional logic and probabilistic programming are closely connected. That is, we can apply techniques from one area to the other and vice versa. We will see that an implementation based on the concepts of functional logic programming can have benefits with respect to performance compared to a standard list-based implementation.

1 Introduction

The probabilistic programming paradigm allows the succinct definition of probabilistic processes and other applications based on probability distributions, for example, Bayesian inference as used in machine learning. The idea of probabilistic programming has been quite successful. There are a variety of probabilistic programming languages supporting all kinds of programming paradigms. For example, the programming languages Church [12] and Anglican [21] are based on the functional programming language Scheme, ProbLog [9] is an extension of the logic programming language Prolog, and Probabilistic C [17] is based on the imperative language C. Besides full-blown languages there are also embedded domain specific languages that implement probabilistic programming as a library. For example, FACTORIE [16] is a library for the hybrid programming language Scala and Erwig and Kollmansberger [10] present a library for the functional programming language Haskell. We recommend the survey by Gordon et al. [13] about the current state of probabilistic programming for further information.

This paper presents *PFLP*, a library providing a domain specific language for probabilistic programming in the functional logic programming language Curry [2]. PFLP makes heavy use of functional logic programming concepts and shows that this paradigm is well-suited for implementing a library for probabilistic programming. In fact, there is a close connection between probabilistic programming and functional logic programming. For example, non-deterministic choice and probabilistic choice are similar concepts. Furthermore, the concept of call-time choice as known from functional logic programming coincides with (stochastic) memoization [8] in the area of probabilistic programming. We are

not the first to observe this close connection between functional logic programming and probabilistic programming. For example, Fischer et al. [11] present a library for modeling functional logic programs in the functional language Haskell. As they state, by extending their approach to weighted non-determinism we can model a probabilistic programming language.

Besides a lightweight implementation of a library for probabilistic programming in a functional logic programming language, this paper makes the following contributions.

- We investigate the interplay of probabilistic programming with the features of a functional logic programming language. For example, we show how call-time choice and non-determinism interplay with probabilistic choice.
- We discuss how we utilize functional logic features to improve the implementation of probabilistic combinators.
- On one hand, we will see that an implementation of probability distributions using non-determinism in combination with non-strict probabilistic combinators can be more efficient than an implementation using lists.
- On the other hand, we illustrate that the combination of non-determinism and non-strictness with respect to distributions has to be handled with care. More precisely, it is important to enforce a certain degree of strictness in order to guarantee correct results.
- Finally, this paper is supposed to foster the exchange between the community of probabilistic programming and of functional logic programming. That is, while the connection exists for a long time, there has not been much exchange between the communities. We would like to take this paper as a starting point to bring these paradigms closer together. Thus, this paper introduces the concepts of both, the functional logic and probabilistic programming, paradigms.

Last but not least, we also want to state a non-contribution. We do not plan to compete against full-blown probabilistic languages or mature libraries for probabilistic programming. Nevertheless, we think that this library is a great showcase for languages with built-in non-determinism, because the functional logic approach can be superior to the functional approach using lists.

2 The Basics

In this section we discuss the core of the PFLP library[3]. The implementation is based on a Haskell library for probabilistic programming presented by Erwig and Kollmansberger [10]. We will not present the whole *PFLP* library, but only core functions. The paper at hand is a literate Curry file. We use the Curry compiler KiCS2[4] by Braßel et al. [5] for all code examples.

[3] We provide the code for the library at `https://github.com/finnteegen/pflp`.

[4] We use version `0.6.0` of KiCS2 and the source is found at `https://www-ps.informatik.uni-kiel.de/kics2/`.

2.1 Modeling Distributions

One key ingredient of probabilistic programming is the definition of distributions. A distribution consists of pairs of elementary events and their probability. We model probabilities as *Float* and distributions as a combination of an elementary event and the corresponding probability.

type *Probability* = *Float*
data *Dist a* = *Dist a Probability*

In a functional language like Haskell, the canonical way to define distributions uses lists. Here, we use Curry's built-in non-determinism as an alternative for lists to model distributions with more than one event-probability pair. As an example, we define a fair coin, where *True* represents heads and *False* represents tails, as follows.[5]

$coin :: Dist\ Bool$
$coin = Dist\ True\ \frac{1}{2}\ ?\ Dist\ False\ \frac{1}{2}$

In Curry the (?)-operator non-deterministically chooses between two given arguments. Non-determinism is not reflected in the type system, that is, a non-deterministic choice has type $a \to a \to a$. Such non-deterministic computations introduced by (?) describe two individual computation branches; one for the left argument und one for the right argument of (?). Printing an expression in the REPL[6] evaluates the non-deterministic computations, thus, yields one result for each branch as shown in the following examples.

$\lambda>$ *coin*	$\lambda>$ 1 ? 2
Dist True 0.5	1
Dist False 0.5	2

It is cumbersome to define distributions explicitly as in the case of *coin*. Hence, we define helper functions for constructing distributions. Given a list of events and probabilities, *enum* creates a distribution by folding these pairs non-deterministically with a helper function *member*.[7]

$member :: [a] \to a$
$member = foldr\ (?)\ failed$
$enum :: [a] \to [Probability] \to Dist\ a$
$enum\ vs\ ps = member\ (zipWith\ Dist\ vs\ ps)$

In Curry the constant *failed* is a silent failure that behaves as neutral element with respect to (?). That is, the expression *True* ? *failed* has the same semantics

[5] Here and in the following we write probabilities as fractions for readability.
[6] We visualize the interactions with the REPL using $\lambda>$ as prompt.
[7] We shorten the implementation of *enum* for presentation purposes; actually, *enum* only allows valid distributions, e.g., that the given probabilities add up to 1.0.

as *True*. Hence, the function *member* takes a list and yields a non-deterministic choice of all elements of the list.

As a short-cut, we define a function that yields a *uniform* distribution given a list of events as well as a function *certainly*, which yields a distribution with a single event of probability one.

```
uniform :: [a] → Dist a
uniform xs = let len = length xs in enum xs (repeat 1/len)
certainly :: a → Dist a
certainly x = Dist x 1.0
```

The function *repeat* yields a list that contains the given value infinitely often. Because of Curry's laziness, it is sufficient if one of the arguments of *enum* is a finite list because *zipWith* stops when one of its arguments is empty.

We can refactor the definition of *coin* using *uniform* as follows.

```
coin :: Dist Bool
coin = uniform [True, False]
```

In general, the library hides the constructor *Dist*, that is, the user has to define distributions by using the combinators provided by the library.

The library provides additional functions to combine and manipulate distributions. In order to work with dependent distributions, the operator (⋙=) applies a function that yields a distribution to each event of a given distribution and multiplies the corresponding probabilities.[8]

```
(⋙=) :: Dist a → (a → Dist b) → Dist b
d ⋙= f = let Dist x p = d
             Dist y q = f x
         in Dist y (p *. q)
```

The implementation via **let**-bindings seems a bit tedious, however, it is important that we define (⋙=) as it is. The canonical implementation performs pattern matching on the first argument but uses a **let**-binding for the result of f. That is, it is strict in the first argument but non-strict in the application of f, the second argument. We discuss the implementation in more detail later. For now, it is sufficient to note that (⋙=) yields a partial *Dist*-constructor without evaluating any of its arguments. In contrast, a definition using pattern matching or a case expression needs to evaluate its argument first, thus, is more strict.

Intuitively, we have to apply the function f to each event of the distribution d and combine the resulting distributions into a single distribution. In a Haskell implementation, we would use a list comprehension to define this function. In the Curry implementation, we model distributions as non-deterministic computations, thus, the above rule describes the behavior of the function for

[8] Due to the lack of overloading in Curry, operations on *Float* have a (floating) point suffix, e.g. (∗.), whereas operations on *Int* use the common operation names.

an arbitrary pair of the first distribution and an arbitrary pair of the second distribution, that is, the result of f.

For independent distributions we provide the function *joinWith* that combines two distributions with respect to a given function. We implement *joinWith* by means of $(\ggg\!=)$.

```
joinWith :: (a → b → c) → Dist a → Dist b → Dist c
joinWith f d1 d2 = d1 >>= λx → d2 >>= λy → certainly (f x y)
```

In a monadic setting this function is sometimes called *liftM2*. Here, we use the same nomenclature as Erwig and Kollmansberger [10].

As an example of combining multiple distinct distributions, we define a function that flips a coin n times.

```
flipCoin :: Int → Dist [Bool]
flipCoin n | n ≡ 0      = certainly []
           | otherwise = joinWith (:) coin (flipCoin (n − 1))
```

When we run the example of flipping two coins in the REPL of KiCS2, we get four events.

```
λ> flipCoin 2
Dist [True, True] 0.25
Dist [True, False] 0.25
Dist [False, True] 0.25
Dist [False, False] 0.25
```

In the example above, *coin* is a non-deterministic operation, namely, $coin = Dist\ True\ \frac{1}{2}\ ?\ Dist\ False\ \frac{1}{2}$. Applying *joinWith* to *coin* and *coin* combines all possible results of two coin tosses.

2.2 Querying Distributions

With a handful of building blocks to define distributions available, we now want to calculate total probabilities, thus, perform queries on our distributions. We provide an operator $(??) :: (a \to Bool) \to Dist\ a \to Probability$ to extract the probability of a distribution with respect to a given predicate. The operator filters events that satisfy the given predicate and computes the total probability of the remaining elementary events. It is straightforward to implement this kind of filter function on distributions in Curry.

```
filterDist :: (a → Bool) → Dist a → Dist a
filterDist p d@(Dist x _) | p x = d
```

The implementation of *filterDist* is a partial identity on the event-probability pairs. Every event that satisfies the predicate is part of the resulting distribution. The function fails for event-predicate pairs that do not satisfy the predicate. The definition is equivalent to the following version using an **if-then-else**-expression.

$$\mathit{filterDist'}\ p\ d@(\mathit{Dist}\ x\ \_) = \mathbf{if}\ p\ x\ \mathbf{then}\ d\ \mathbf{else}\ \mathit{failed}$$

Computing the total probability, i.e., summing up all remaining probabilities, is a more advanced task in the functional logic approach. Remember that we represent a distribution by chaining all event-probability pairs with (?), thus, constructing non-deterministic computations. These non-deterministic computations introduce individual branches of computations that cannot interact with each other. In order to compute the total probability of a distribution, we have to merge these distinct branches. Such a merge is possible by the encapsulation of non-deterministic computations. Similar to the *findall* construct of the logic language Prolog, in Curry we encapsulate a non-deterministic computation by using a primitive called *allValues*[9]. The function *allValues* operates on a polymorphic — and potentially non-deterministic — value and yields a multi-set of all non-deterministic values.

$$\mathit{allValues} :: a \to \{a\}$$

In order to work with encapsulated values, Curry provides the following two functions to fold and map the resulting multi-set.

$$\mathit{foldValues} :: (a \to a \to a) \to a \to \{a\} \to a$$
$$\mathit{mapValues} :: (a \to b) \to \{a\} \to \{b\}$$

We do not discuss the implementation details behind *allValues* here. It is sufficient to know that, as a library developer, we can employ this powerful function to encapsulate non-deterministic values and use these values in further computations. However, due to intransparent behavior in combination with sharing as discussed by Braßel et al. [4], a user of the library should not use *allValues* at all. In a nutshell, inner-most and outer-most evaluation strategies may cause different results when combining sharing and encapsulation.

With this encapsulation mechanism at hand, we can define the extraction operator (??) as follows.

$$\mathit{prob} :: \mathit{Dist}\ a \to \mathit{Probability}$$
$$\mathit{prob}\ (\mathit{Dist}\ \_\ p) = p$$
$$(??) :: (a \to \mathit{Bool}) \to \mathit{Dist}\ a \to \mathit{Probability}$$
$$(??)\ p = \mathit{foldValues}\ (+.)\ 0.0 \circ \mathit{allValues} \circ \mathit{prob} \circ \mathit{filterDist}\ p$$

First we filter the elementary events by some predicate and project to the probabilities only. Afterwards we encapsulate the remaining probabilities and sum them up. As an example for the use of (??), we may flip four coins and calculate the probability of at least two heads — that is, *True*.

```
λ> ((>= 2) . length . filter id) ?? flipCoin 4
0.6875
```

[9] We use an abstract view of the result of an encapsulation to emphasize that the order of encapsulated results does not matter. In practice, we can, for example, use the function $\mathit{allValues} :: a \to [a]$ defined in the library *Findall*.

3 The Details

Up to now, we have discussed a simple library for probabilistic programming that uses non-determinism to represent distributions. In this chapter we will see that we can highly benefit from Curry-like non-determinism with respect to performance when we compare PFLP's implementation with a list-based implementation. More precisely, when we query a distribution with a predicate that does not evaluate its argument completely, we can possibly prune large parts of the search space. Before we discuss the details of the combination of non-strictness and non-determinism, we discuss aspects of sharing non-deterministic choices. At last, we discuss details about the implementation of $(\ggg\!=)$ and why PFLP does not allow non-deterministic events within distributions.

3.1 Call-time Choice vs. Run-time Choice

By default Curry uses call-time choice, that is, variables denote single deterministic choices. When we bind a variable to a non-deterministic computation, one value is chosen and all occurrences of the variable denote the same deterministic choice. Often call-time choice is what you are looking for. For example, the definition of *filterDist* makes use of call-time choice.

$$
\begin{aligned}
&\mathit{filterDist} :: (a \to \mathit{Bool}) \to \mathit{Dist}\ a \to \mathit{Dist}\ a\\
&\mathit{filterDist}\ p\ d@(\mathit{Dist}\ x\ \_) \mid p\ x = d
\end{aligned}
$$

The variable d on the right-hand side denotes a single deterministic choice, namely, the one that satisfies the predicate and not the non-deterministic computation that was initially passed to *filterDist*.

Almost as often run-time choice is what you are looking for and call-time choice gets in your way; probabilistic programming is no exception. For example, let us reconsider flipping a coin n times. We parametrize the function *flipCoin* over the given distribution and define the following generalized function.

$$
\begin{aligned}
&\mathit{replicateDist} :: \mathit{Int} \to \mathit{Dist}\ a \to \mathit{Dist}\ [a]\\
&\mathit{replicateDist}\ n\ d \mid n \equiv 0 \quad\ = \mathit{certainly}\ [\,]\\
&\qquad\qquad\quad\ \mid \mathit{otherwise} = \mathit{joinWith}\ (:)\ d\ (\mathit{replicateDist}\ (n-1)\ d)
\end{aligned}
$$

When we use this function to flip a coin twice, the result is not what we intended.

$$
\begin{aligned}
&\lambda{>}\ \mathit{replicateDist}\ 2\ \mathit{coin}\\
&\mathit{Dist}\ [\mathit{True}, \mathit{True}]\ 0.25\\
&\mathit{Dist}\ [\mathit{False}, \mathit{False}]\ 0.25
\end{aligned}
$$

Because *replicateDist* shares the variable d, we only perform a choice once and replicate deterministic choices. In contrast, top-level nullary functions like *coin* are evaluated every time, thus, exhibit run-time choice, which is the reason why the previously shown *flipCoin* behaves properly.

In order to implement *replicateDist* correctly, we have to enforce run-time choice. We introduce the following type synonym and function to model and work with values with run-time choice behavior.

```
type RT a = () → a
pick :: RT a → a
pick rt = rt ()
```

We can now use the type *RT* to hide the non-determinism on the right-hand side of a function arrow. This way, *pick* explicitly triggers the evaluation of *rt*, performing a new choice for every element of the result list.

```
replicateDist :: Int → RT (Dist a) → Dist [a]
replicateDist n rt | n ≡ 0      = certainly []
                   | otherwise = joinWith (:) (pick rt) (replicateDist (n − 1) rt)
```

In order to use *replicateDist* with *coin*, we have to construct a value of type *RT* (*Dist Bool*). However, we cannot provide a function to construct a value of type *RT* that behaves as intended. Such a function would share a deterministic choice and non-deterministically yield two functions, instead of one function that yields a non-deterministic computation. The only way to construct a value of type *RT* is to explicitly use a lambda abstraction.

```
λ> replicateDist 2 (λ() → coin)
Dist [True, True] 0.25
Dist [True, False] 0.25
Dist [False, True] 0.25
Dist [False, False] 0.25
```

Instead of relying on call-time choice as default behavior, we could model *Dist* as a function and make run-time choice the default in PFLP. In this case, to get call-time choice we would have to use a special construct provided by the library — as it is the case in many probabilistic programming libraries, e.g., *mem* in Church. We have decided to go with the current modeling based on call-time-choice, because the alternative would work against the spirit of the Curry programming language.

There is a long history of discussions about the pros and cons of call-time choice and run-time choice. It is common knowledge in probabilistic programming [8] that, in order to model stochastic automata or probabilistic grammars, memoization — that is, call-time choice — has to be avoided. Similarly, Antoy [1] observes that you need run-time choice to elegantly model regular expressions in the context of functional logic programming languages. Then again, probabilistic languages need a concept like memoization in order to use a single value drawn from a distribution multiple times. If we flip a coin and have more than one dependency on its result in the remaining program, the result of that flip is not supposed to change between one occurrence and the other.

3.2 Combination of Non-strictness and Non-determinism

This section illustrates the benefits from the combination of non-strictness and non-determinism with respect to performance. More precisely, in a setting that uses Curry-like non-determinism, non-strictness can prevent non-determinism from being "spawned". Let us consider calculating the probability for throwing only sixes when throwing n dice. First we define a uniform die as follows.

```
data Side = One | Two | Three | Four | Five | Six
die :: Dist Side
die = uniform [One, Two, Three, Four, Five, Six]
```

We define the following query by means of the combinators introduced so far. The function *all* simply checks that all elements of a list satisfy a given predicate; it is defined by means of the boolean conjunction ($\wedge$).

```
allSix :: Int → Probability
allSix n = all (≡ Six) ?? replicateDist n (λ() → die)
```

The following table compares running times[10] of this query for different numbers of dice. The row labeled "Curry ND" lists the running times for an implementation that uses the operator ($\ggg=$). The row "Curry List" shows the numbers for a list-based implementation in Curry, which is a literal translation of the library by Erwig and Kollmansberger. The row labeled "Curry ND!" uses an operator ($\ggg=!$) instead, which we will discuss shortly. Finally, we compare our implementation to the original list-based implementation, which the row labeled "Haskell List" refers to. The table states the running times in milliseconds of a compiled executable for each benchmark as a mean of three runs. Cells marked with "–" take more than a minute.

# of dice	5	6	7	8	9	10	100	200	300
Curry ND	<1	<1	<1	<1	<1	<1	48	231	547
Curry List	2	13	72	419	2554	15 394	–	–	–
Curry ND!	52	409	2568	16 382	–	–	–	–	–
Haskell List	1	5	30	210	1415	6538	–	–	–

Obviously, the example above is a little contrived. While the query is exponential in both list versions, it is linear in the non-deterministic setting[11]. In order to illustrate the behavior of the example above, we consider the following application for an arbitrary distribution *dist* of type *Dist* [*Side*].

```
filterDist (all (≡ Six)) (joinWith (:) (Dist One 1/6) dist)
```

[10] All benchmarks were executed on a Linux machine with an Intel Core i7-6500U (2.50 GHz) and 8 GiB RAM running Fedora 25. We used the Glasgow Haskell Compiler (version `8.0.2`, option `-O2`) and set the search strategy in KiCS2 to depth-first.

[11] Non-determinism causes significant overhead for KiCS2, thus, "Curry ND" does not show linear development, but we measured a linear running time using PAKCS [14].

This application yields an empty distribution without evaluating the distribution *dist*. The clou here is that *joinWith* yields a *Dist* constructor without inspecting its arguments. When we demand the event of the resulting *Dist*, *joinWith* has to evaluate only its first argument to see that the predicate *all* ($\equiv$ *Six*) yields *False*. The evaluation of the expression fails without inspecting the second argument of *joinWith*. Figure 1 illustrates the evaluation in more detail.

In case of the example *allSix*, all non-deterministic branches that contain a value different from *Six* fail fast due to the non-strictness. Thus, the number of evaluation steps is linear in the number of rolled dice. Note that a similar behavior is *not* possible in a list-based implementation that implements ($\gg\!=$) with *concatMap*. In such an implementation, we have to traverse the entire distribution before we can evaluate the predicate *all* ($\equiv$ *Six*). The consequence is that the running times of "Haskell List" cannot compete with "Curry ND" when the number of dice increases.

We can only benefit from the combination of non-strictness and non-determinism if we define ($\gg\!=$) with care. Let us take a look at a strict variant of ($\gg\!=$) and discuss its consequences.

$$(\gg\!=\!!) :: \mathit{Dist}\ a \to (a \to \mathit{Dist}\ b) \to \mathit{Dist}\ b$$
$$\mathit{Dist}\ x\ p \gg\!=\! !\ f = \mathbf{case}\ f\ x\ \mathbf{of}\ \mathit{Dist}\ y\ q \to \mathit{Dist}\ y\ (p *.\ q)$$

This implementation is strict in its first argument as well as in the result of the function application. When we use ($\gg\!=\!!$) to implement the *allSix* example, we lose the benefit of Curry-like non-determinism. The row labeled "Curry ND!" shows the running times when using ($\gg\!=\!!$) instead of ($\gg\!=$). As ($\gg\!=\!!$) is strict, the function *joinWith* has to evaluate both its arguments to yield a result.

Intuitively, we expect similar running times for "Curry ND!" and "Curry List". However, this is not the case. "Curry ND!" heavily relies on non-deterministic computations, which causes significant overhead for KiCS2. We do not investigate these differences here but propose it as a direction for future research.

Obviously, turning an exponential problem into a linear one is like getting only sixes when throwing dice. In most cases we are not that lucky. For example, consider the following query for throwing n dice that are either five or six.

$$\mathit{allFiveOrSix} :: \mathit{Int} \to \mathit{Probability}$$
$$\mathit{allFiveOrSix}\ n = \mathit{all}\ (\lambda s \to s \equiv \mathit{Five} \vee s \equiv \mathit{Six})\ ??\ \mathit{replicateDist}\ n\ (\lambda() \to \mathit{die})$$

We again list running times for different numbers of dice for this query.

# of dice	5	6	7	8	9	10
Curry ND	4	7	15	34	76	163
Curry List	2	13	84	489	2869	16 989
Curry ND!	49	382	2483	15 562	–	–
Haskell List	2	5	31	219	1423	6670

As we can see from the running times, this query is exponential in all implementations. Nevertheless, the running time of the non-strict, non-deterministic

$$
\begin{array}{l}
\mathit{filterDist}\ (\mathit{all}\ (\equiv \mathit{Six}))\ (\mathit{joinWith}\ (:)\ (\mathit{Dist}\ \mathit{One}\ \frac{1}{6})\ \mathit{dist}) \\
\equiv \quad \{\ \text{Def. of } \mathit{joinWith}\ \} \\
\mathit{filterDist}\ (\mathit{all}\ (\equiv \mathit{Six})) \\
\qquad (\mathit{Dist}\ \mathit{One}\ \frac{1}{6} \ggg\!= \lambda x \to \mathit{dist} \ggg\!= \lambda xs \to \mathit{certainly}\ (x : xs)) \\
\equiv \quad \{\ \text{Def. of } (\ggg\!=)\ \text{(twice)}\ \} \\
\mathit{filterDist}\ (\mathit{all}\ (\equiv \mathit{Six})) \\
\qquad (\textbf{let}\ \mathit{Dist}\ x\ p = \mathit{Dist}\ \mathit{One}\ \frac{1}{6}; \mathit{Dist}\ xs\ q = \mathit{dist}; \mathit{Dist}\ ys\ r = \mathit{certainly}\ (x : xs) \\
\qquad\ \textbf{in}\ \mathit{Dist}\ ys\ (p *.\ (q *.\ r)) \\
\equiv \quad \{\ \text{Def. of } \mathit{filterDist}\ \} \\
\textbf{let}\ \mathit{Dist}\ x\ p = \mathit{Dist}\ \mathit{One}\ \frac{1}{6}; \mathit{Dist}\ xs\ q = \mathit{dist}; \mathit{Dist}\ ys\ r = \mathit{certainly}\ (x : xs) \\
\textbf{in if}\ \mathit{all}\ (\equiv \mathit{Six})\ ys\ \textbf{then}\ \mathit{Dist}\ ys\ (p *.\ (q *.\ r))\ \textbf{else}\ \mathit{failed} \\
\equiv \quad \{\ \text{Def. of } \mathit{certainly}\ \} \\
\textbf{let}\ \mathit{Dist}\ x\ p = \mathit{Dist}\ \mathit{One}\ \frac{1}{6}; \mathit{Dist}\ xs\ q = \mathit{dist} \\
\textbf{in if}\ \mathit{all}\ (\equiv \mathit{Six})\ (x : xs)\ \textbf{then}\ \mathit{Dist}\ (x : xs)\ (p *.\ (q *.\ 1.0))\ \textbf{else}\ \mathit{failed} \\
\equiv \quad \{\ \text{Def. of } \mathit{all}\ \} \\
\textbf{let}\ \mathit{Dist}\ x\ p = \mathit{Dist}\ \mathit{One}\ \frac{1}{6}; \mathit{Dist}\ xs\ q = \mathit{dist} \\
\textbf{in if}\ x \equiv \mathit{Six} \wedge \mathit{all}\ (\equiv \mathit{Six})\ xs\ \textbf{then}\ \mathit{Dist}\ (x : xs)\ (p *.\ (q *.\ 1.0))\ \textbf{else}\ \mathit{failed} \\
\equiv \quad \{\ \text{Def. of } (\equiv) \text{ and } (\wedge)\ \} \\
\textbf{let}\ \mathit{Dist}\ x\ p = \mathit{Dist}\ \mathit{One}\ \frac{1}{6}; \mathit{Dist}\ xs\ q = d \\
\textbf{in if}\ \mathit{False}\ \textbf{then}\ \mathit{Dist}\ (x : xs)\ (p *.\ (q *.\ 1.0))\ \textbf{else}\ \mathit{failed} \\
\equiv \quad \{\ \text{Def. of } \textbf{if} - \textbf{then} - \textbf{else}\ \} \\
\mathit{failed}
\end{array}
$$

Fig. 1. Simplified evaluation illustrating non-strict non-determinism

implementation is much better because we only have to consider two sides — six and five — while we have to consider all sides in the list implementations and the non-deterministic, strict implementation. That is, while the basis of the complexity is two in the case of the non-deterministic, non-strict implementation, it is six in all the other cases. Again, we get an overhead of a factor around 25 in the case of the strict non-determinism compared to the list implementation.

3.3 Definition of the Bind Operator

In this section we discuss our design choices concerning the implementation of the bind operator. We illustrate that we have to be careful about non-strictness, because we do not want to lose non-deterministic results. Most importantly, the final implementation ensures that users cannot misuse the library if they stick to one simple rule.

First, we revisit the definition of $(\ggg\!=)$ introduced in Section 2.

$$
\begin{array}{l}
(\ggg\!=) :: \mathit{Dist}\ a \to (a \to \mathit{Dist}\ b) \to \mathit{Dist}\ b \\
d \ggg\!= f = \textbf{let}\ \mathit{Dist}\ x\ p = d \\
\qquad\qquad\ \ \mathit{Dist}\ y\ q = f\ x \\
\qquad\qquad \textbf{in}\ \mathit{Dist}\ y\ (p *.\ q)
\end{array}
$$

We can observe two facts about this definition. First, the definition yields a *Dist*-constructor without matching any argument. Second, if neither the event nor the probability of the final distribution is evaluated, the application of the function f is not evaluated as well.

We can observe these properties with some exemplary usages of $(\ggg\!=)$. As a reference, we see that pattern matching the *Dist*-constructor of a *coin* triggers the non-determinism and yields two results.

```
λ> (λ(Dist _ _) → True) coin
True
True
```

In contrast, distributions resulting from an application of $(\ggg\!=)$ behave differently. This time, pattern matching on the *Dist*-constructor does not trigger any non-determinism.

```
λ> (λ(Dist _ _) → True) (certainly () >>= λ_ → coin)
True
λ> (λ(Dist _ _) → True) (coin >>= certainly)
True
```

We observe that the last two examples yield a single result, because the $(\ggg\!=)$-operator changes the position of the non-determinism. That is, the non-determinism does not reside at the same level as the *Dist*-constructor, but in the arguments of *Dist*. Therefore, we have to be sure to trigger all non-determinism when we compute probabilities. Not evaluating non-determinism might lead to false results when we sum up probabilities. Hence, non-strictness is a crucial property for positive pruning effects, but has to be used carefully.

Consider the following example usage of $(\ggg\!=)$, which is simply an inlined version of *joinWith* applied to the boolean conjunction $(\wedge)$.

```
λ> (λ(Dist x _) → x) (coin >>= λx → coin >>= λy → certainly (x ∧ y))
False
True
False
```

We lose one expected result from the distribution, because $(\wedge)$ is non-strict in its second argument in case the first argument is *False*. When the first *coin* evaluates to *False*, $(\ggg\!=)$ ignores the second coin and yields *False* straightaway. In this case, the non-determinism of the second *coin* is not triggered and we get only three instead of four results. The non-strictness of $(\wedge)$ has no consequences when using $(\ggg\!=!)$, because the operator evaluates both arguments and, thus, triggers the non-determinism.

As we have seen above, when using the non-strict operator $(\wedge)$, one of the results gets lost. However, when we sum up probabilities, we do not want events to get lost. For example, when we compute the total probability of a distribution, the result should always be 1.0. However, the query above has only three results and every event has a probability of 0.25, resulting in a total probability of 0.75.

Here is the good news. While events can get lost when passing non-strict functions to ($\ggg=$), probabilities never get lost. For example, consider the following application.

$\lambda>$ $(\lambda(Dist\ \_\ p) \to p)$ $(coin \ggg= \lambda x \to coin \ggg= \lambda y \to certainly\ (x \wedge y))$
0.25
0.25
0.25
0.25

Since multiplication is strict, if we demand the resulting probability, the operator ($\ggg=$) has to evaluate the *Dist*-constructor and its probability. That is, no values get lost if we evaluate the resulting probability. Fortunately, the query operation (??) calculates the total probability of the filtered distributions, thus, evaluates the probability as the following example shows.

$\lambda>$ not ?? $(coin \ggg= \lambda x \to coin \ggg= \lambda y \to certainly\ (x \wedge y))$
0.75

We calculate the probability of the event *False* and while there where only two *False* events, the total probability is still 0.75, i.e., three times 0.25.

All in all, in order to benefit from non-strictness, all operations have to use the right amount of strictness, not too much and not too little. For this reason PFLP does not provide the *Dist*-constructor nor the corresponding projection functions to the user. With this restriction, the library guarantees that no relevant probabilities get lost.

3.4 Non-deterministic Events

We assume that all events passed to library functions are deterministic, thus, do not support non-deterministic events within distributions. In order to illustrate why, we consider an example that breaks this rule here.

Curry provides free variables, that is, expressions that non-deterministically evaluate to every possible value of its type. When we revisit the definition of a die, we might be tempted to use a free variable instead of explicitly enumerating all values of type *Side*. For example, consider the following definition of a die.

$die2 :: Dist\ Side$
$die2 = Dist\ unknown\ \frac{1}{6}$

We just use a free variable — the constant *unknown* — and calculate the probability of each event ourselves. The free variable non-deterministically yields all constructors of type *Side*. Now, let us consider the following query.

$\lambda>$ $const\ True$?? $die2$
0.16666667

The result of this query is $\frac{1}{6}$ and not 1.0 as expected. This example illustrates that probabilities can get lost if we do not use the right amount of strictness. The definition of (??) first projects to the probability of *die2* and throws away all non-determinism. Therefore, we lose probabilities we would like to sum up.

As a consequence for PFLP, non-deterministic events within a distribution are not allowed. If users of the library stick to this rule, it is not possible to misuse the operations and lose non-deterministic results due to non-strictness.

4 Related and Future Work

The approach of this paper is based on the work by Erwig and Kollmansberger [10], who introduce a Haskell library that represents distributions as lists of event-probability pairs. Their library also provides a simple sampling mechanism to perform inference on distributions. Inference algorithms come into play because common examples in probabilistic programming have an exponential growth and it is not feasible to compute the whole distribution. Similarly, Ścibior et al. [19] present a more efficient implementation using a DSL in Haskell. They represent distributions as a free monad and inference algorithms as an interpretation of the monadic structure. Thanks to this interpretation, the approach is competitive to full-blown probabilistic programming languages with respect to performance. PFLP provides functions to sample from distributions as well. However, in this work we focus on modeling distributions and do not discuss any sampling mechanism. In particular, as future work we plan to investigate whether we can benefit from the improved performance as presented here in the case of sampling. Furthermore, a more detailed investigation of the performance of non-determinism in comparison to a list model is a topic for another paper.

The benefit with respect to the combination of non-strictness and non-determinism is similar to the benefit of property-based testing using Curry-like non-determinism in Haskell [18] and Curry [6]. In property-based testing, sometimes we want to generate test cases that satisfy a precondition. With Curry-like non-determinism the precondition can prune the search space early, while a list-based implementation has to generate all test cases and filter them afterwards. Both applications, probabilistic programming and property-based testing, are examples, where built-in non-determinism outperforms list-based approaches as introduced by Wadler [20]. In comparison to property-based testing, here, we observe that we can even add a kind of monadic layer on top of the non-determinism that computes additional information and still preserve the demand driven behavior. However, the additional information has to be evaluated strictly — as it is the case for probabilities, otherwise we might lose non-deterministic results.

There are other more elaborated approaches to implement a library for probabilistic programming. For example, Kiselyov and Shan [15] extend their library for probabilistic programming in OCAML with a construct for lazy evaluation to achieve similar positive effects. However, they use lazy evaluation for a concrete application based on importance sampling. Due to the combination of non-

strictness and non-determinism, we can efficiently calculate the total probability of the resulting distribution without utilizing sampling.

As future work, we see a high potential for performance improvements for the Curry compiler KiCS2. PFLP serves as a starting point for further studies of functional logic features in practical applications. For example, we would expect the running times of the strict implementation based on non-determinism to be approximately as efficient as a list-based implementation. However, as the numbers in Section 3 show, the list approach is considerably faster.

The library's design does not support the use of non-determinism in events or probabilities of a distribution. In case of deeper non-determinism, we have to be careful to trigger all non-determinism when querying a distribution as shown in Section 3. Hence, the extension of the library with an interface using non-determinism on the user's side is an idea worth studying.

Last but not least, we see an opportunity to apply ideas and solutions of the functional logic paradigm in probabilistic programming. For instance, Christiansen et al. [7] investigate free theorems for functional logic programs. As their work considers non-determinism and sharing, adapting it to probabilistic programming should be easy. As another example, Braßel [3] presents a debugger for Curry that works well with non-determinism. Hence, it should be easy to reuse these ideas in the setting of probabilistic programming as well.

5 Conclusion

We have implemented a simple library for probabilistic programming in a functional logic programming language, namely Curry. Such a library proves to be a good fit for a functional logic language, because both paradigms share similar features. While other libraries need to reimplement features specific to probabilistic programming, we solely rely on core features of functional logic languages.

The key idea of the library is to use non-determinism to model distributions. We discussed design choices as well as corresponding disadvantages and advantages of this approach. In the end, the library uses non-strict probabilistic combinators in order to avoid spawning unnecessary non-deterministic computations and, thus, benefit in terms of performance due to early pruning. However, we have observed that the combination of non-strictness and non-deterministic needs to be taken with a pinch of salt. Using combinators that are too strict leads to a loss of the aforementioned performance benefit.

Acknowledgements We are thankful for fruitful discussions with Michael Hanus as well as suggestions of Jan Bracker and the anonymous reviewers to improve the readability of this paper.

References

[1] Antoy, S.: Evaluation Strategies for Functional Logic Programming. Journal of Symbolic Computation 40(1) (2005)

[2] Antoy, S., Hanus, M.: Functional Logic Programming. Communications of the ACM 53(4) (2010)
[3] Braßel, B.: A Technique to Build Debugging Tools for Lazy Functional Logic Languages. Electronic Notes in Theoretical Computer Science 246 (2009)
[4] Braßel, B., Hanus, M., Huch, F.: Encapsulating Non-Determinism in Functional Logic Computations. Journal of Functional and Logic Programming (2004)
[5] Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A New Compiler from Curry to Haskell. In: Proc. of International Conf. on Functional and Constraint Logic Programming (2011)
[6] Christiansen, J., Fischer, S.: EasyCheck – Test Data for Free. In: Proc. of International Symposium on Functional and Logic Programming (2008)
[7] Christiansen, J., Seidel, D., Voigtländer, J.: Free Theorems for Functional Logic Programs. In: Proc. of Workshop on Programming Languages Meets Program Verification (2010)
[8] De Raedt, L., Kimmig, A.: Probabilistic Programming Concepts. arXiv:1312.4328 (preprint) (2013)
[9] De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: Proc. of International Joint Conf. on Artifical Intelligence (2007)
[10] Erwig, M., Kollmansberger, S.: Functional Pearls: Probabilistic Functional Programming in Haskell. Journal of Functional Programming 16(1) (2006)
[11] Fischer, S., Kiselyov, O., Shan, C.c.: Purely Functional Lazy Nondeterministic Programming. In: Proc. of International Conf. on Functional Programming (2009)
[12] Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a Language for Generative Models. CoRR (2012)
[13] Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic Programming. In: Proc. of the on Future of Software Engineering (2014)
[14] Hanus (ed.), M.: PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/pakcs/` (2017)
[15] Kiselyov, O., Shan, C.C.: Embedded Probabilistic Programming. In: Proc. of Working Conf. on Domain-Specific Languages (2009)
[16] McCallum, A., Schultz, K., Singh, S.: FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In: Proc. of International Conf. on Neural Information Processing Systems (2009)
[17] Paige, B., Wood, F.: A Compilation Target for Probabilistic Programming Languages. In: Proc. of International Conf. on Machine Learning (2014)
[18] Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In: Proc. of Symposium on Haskell (2008)
[19] Ścibior, A., Ghahramani, Z., Gordon, A.D.: Practical Probabilistic Programming with Monads. In: Proc. of Symposium on Haskell (2015)
[20] Wadler, P.: How to Replace Failure by a List of Successes. In: Proc. of a Conf. on Functional Programming Languages and Computer Arch. (1985)
[21] Wood, F., Meent, J.W., Mansinghka, V.: A new Approach to Probabilistic Programming Inference. In: Artificial Intelligence and Statistics (2014)