A General Importance Sampling Algorithm for Probabilistic Programs

Avi Pfeffer

TR-12-07

Computer Science Group
Harvard University
Cambridge, Massachusetts

A General Importance Sampling Algorithm for Probabilistic Programs

Avi Pfeffer
Harvard School of Engineering and Applied Sciences
avi@eecs.harvard.edu

Abstract

Highly expressive probabilistic modeling languages are capable of describing a wide variety of models. Some of these models are quite complex, so approximate inference algorithms are needed. One approach to approximate inference is importance sampling, but this can be hard to do in expressive languages because of the many deterministic relationships between concepts. This paper presents an importance sampling algorithm for the IBAL language based on the principle of using the structure of a model to infer as much as possible about a decision before making a commitment. The paper demonstrates using a musical example how easy it is to encode interesting new models in IBAL. Results show that the importance sampling algorithm is able to make useful inferences, and is far superior to a rejection sampling algorithm. The paper presents proof of concept on the musical example that the algorithm is capable of handling real applications.

1 Introduction

Recent years have seen great interest in highly expressive probabilistic modeling languages with first-order power. These languages allow the representation of a wide variety of domains, often much more elegantly and compactly than with standard propositional representations. Along with this greater expressivity, however, comes an inference challenge. It is very easy to create models for which exact inference is difficult. Therefore the search for approximate inference algorithms for these languages is important.

There are many approximate inference algorithms for graphical models. One important family of inference algorithms is sampling, of which there are three main kinds: rejection sampling (RS), known in Bayesian networks as logic sampling [Henrion, 1988], importance sampling, known in Bayesian networks as likelihood weighting [Fung and Chang, 1989; Shachter and Peot, 1989], and Markov chain Monte Carlo (MCMC) [Gilks *et al.*, 1996]. Recently progress was made on performing MCMC for the BLOG language [Milch and Russell, 2006], but designing good proposal distributions is still hard.

It is easy to develop a rejection sampling algorithm for expressive languages. Importance sampling, which is generally considered to be better than RS, is more difficult. The problem is that models in expressive languages often contain many high level concepts that are defined in terms of other concepts, and these lead to deterministic relationships, which provide a challenge for many inference algorithms, including MCMC. For importance sampling, the deterministic relationships may force the sampler to make early commitments that eventually lead to evidence having zero probability. This paper presents a general importance sampling algorithm that uses the structure of a model to infer as much as possible about a probabilistic expression before sampling it. The algorithm is presented for the IBAL language, which has the full expressive power of programming languages. IBAL provides an ideal language for studying these issues because the structure of the language provides tools for performing many interesting inferences.

The issues discussed in this paper are not specific to IBAL. Many languages for first-order probabilistic modeling use knowledge-based model construction as their inference algorithm, in which a single Bayesian network is constructed to capture the entire model, and a Bayesian network inference algorithm is run on it to answer queries. This is true, for example, for Bayesian logic programs [Kersting and de Raedt, 2000] and multi-entity Bayesian networks [Laskey and Costa, 2005]. Likelihood weighting is a candidate inference algorithm on the constructed network, but it

would probably fail because of the deterministic relationships. Ideally, one would perform some sort of reasoning about the observations to prevent this from happening. But unfortunately, once the Bayesian network is constructed, it is too late. Everything has been reduced to a node, and the structure inherent in the program has been lost. In IBAL, by contrast, inference is performed directly on the program itself, so any structure present that allows reasoning to be performed about observations can be exploited.

The paper begins by presenting the IBAL language, and presenting a musical application that shows how easy it is to create interesting new models in IBAL. It then proceeds to describe the importance sampling algorithm, beginning with simple inferences and moving gradually to more sophisticated ones. Examples are presented for each of the steps illustrating their benefits. Finally, results are presented for the musical application that show how the algorithm is able to perform on a reasonably sized application.

2 The IBAL language

A model in IBAL looks like a program in a functional programming language. Evaluating an expression in an ordinary programming language produces a value; evaluating an expression in IBAL stochastically produces a value, while making sure that certain observations are satisfied. The meaning of the IBAL program is the probability distribution over the output conditioned on the observations. IBAL is defined as follows.

A *value* is a symbol, boolean or integer constant, a function value, or a tuple $\{a_1 : x_1, ..., a_n : x_n\}$ where each x_i is a value. In any tuple τ, the notation $\tau.a$ denotes the entity associated with field a in τ. An *environment* is a mapping from variable names to values. A *pattern* is either a constant, the special pattern `_` (meaning "any"), or a tuple of patterns. A value *matches* a pattern as follows. All values match the pattern `_`. A constant value matches the same constant pattern. A tuple value matches a tuple pattern if every field in the pattern is also a field in the value, and the corresponding patterns match the corresponding values. An expression is one of the following:

c	// constant
a	// variable
$\{a_1 = \epsilon_1, ..., a_n = \epsilon_n\}$	// tuple construction
$\epsilon.a$	// component access
`if` ϵ_1 `then` ϵ_2 `else` ϵ_3	// conditional
`dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$	// stochastic choice
`let` $a = \epsilon_1$ `in` ϵ_2	// variable binding
`let` $a_0(a_1, ..., a_n) = \epsilon_1$ `in` ϵ_2	// function definition
$\epsilon_1 \otimes \epsilon_2$	// operator
$\epsilon_0(\epsilon_1, ..., \epsilon_n)$	// function application
ϵ `|=` π	// pattern matching
`observe` π `in` ϵ	// observation

ϵ denotes an expression and π a pattern. The semantics of IBAL is defined formally in [Pfeffer, To appear]. Informally and intuitively, it can be understood using a rejection sampling process. Each expression corresponds to an experiment in which the expression is sampled in an environment. To sample `let` $a = \epsilon_1$ `in` ϵ_2, the algorithm first samples ϵ_1, then samples ϵ_2 in the environment in which a is bound to the value of ϵ_1. Note that an expression form ϵ `|=` π is provided. This samples ϵ and returns *true* if the value matches π. IBAL provides two constructs not found in ordinary programming languages. The expression `dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$ corresponds to the process of choosing randomly from $1, ..., n$ according to the probabilities p_i, and then sampling the corresponding subexpression. To sample the expression `observe` π `in` ϵ, the sampler first samples ϵ to produce a value x. Then, if x matches π, x is returned, otherwise a `Reject` exception is thrown. This process defines a probability that a sample will not be rejected, which constitutes the probability of evidence, and a probability distribution over values sampled, which constitutes the conditional probability distribution over the output given the evidence.

Besides these constructs, IBAL offers syntactic sugar, some of which will be used in this paper, including `case` expressions (analogous to C's `switch`), and `uniform` for sampling from the uniform distribution. It also provides standard notation for lists, which are implemented as tuples with fields `TAG`, `CAR` and `CDR`. In particular the notation $\epsilon_1 :: \epsilon_2$ denotes the list formed from consing the result of ϵ_1 to the result of ϵ_2.

3 A Musical Example

The example in this section illustrates the IBAL language, and demonstrates how easy it is to create interesting new models in the language. An important phenomenon in music, particularly classical music, is motivic development, in which one idea is transformed and re-presented throughout a piece. A target motive is obtained from a source motive by a set of operations, including transposition, deletion, inversion and insertion. We have developed a grammar model for modeling transformations. The idea is to take a source melody and divide it hierarchically in a tree. At each node of the tree, one of a set of possible operations is applied to each subtree. At each node, the melodies at the children of the node are obtained, the appropriate operations applied to them, and the results concatenated. Part of the IBAL code for this model is

presented below.

```
let splitAt(n,l) =
  if n |= 0
  then { fst = [], snd = l }
  else
    let x = splitAt(n-1,l.CDR) in
    { fst = l.CAR :: x.fst, snd = x.snd } in
let split(l) =
  let n = uniform(length(l) + 1) in
  splitAt(n,l) in
let transpose2(note) =
  case note of {
    'e : dist [ 0.4 : 'f, 0.6 : 'fsharp ];
  etc. } in
let transform(input) =
  let maptranspose2(list) =
(*)    map(transpose2,transform(list))) in
  etc.
  let chooseOper() =
    dist [ 0.4 : identity,
           0.2 : delete,
           0.1 : maptranspose2, etc. ] in
  if input |= []
  then []
  else
    let f1 = chooseOper() in
    let f2 = chooseOper() in
    let z = split(input) in
(**)    append(f1(z.fst), f2(z.snd))
in
observe ['c, 'b, 'csharp, 'd, 'e] in
transform(['a, 'gsharp, 'a, 'c, 'gsharp, 'a])
```

The model uses list utilities such as `append`, which concatenates two lists, and `map`, which applies a function to all members of a list. In the grammar, there are three points at which a stochastic choice is made. The first is in splitting a list into two parts; the `split` function splits it at a uniformly chosen point. The second stochastic choice is in the transpose operation, of which there are six, for the different possible intervals. A transposition is a shift of a sequence of notes by an interval but there is some randomness in how it is applied to individual notes. Dor example, will transposition by a second shift a note by a minor second or a major second? Now let us examine the operation of `transform`. The third source of randomness comes in when the choice of which functions `f1` and `f2` to apply to each part of the list being transformed is made. The core operation is in the line marked `(**)`. Here, the chosen functions are applied to each half of the list and the result is concatenated. In particular, `f1` can be `maptranspose2`, defined in line `(*)`. This function transposes every element of its argument by a second, and then recursively applies `transform` to the result.

4 Basic Ideas

We assume that the reader is familiar with the likelihood weighting algorithm. What makes likelihood weighting work is that whenever a value is sampled for a variable, the sampler knows whether or not there is an observation for the variable, and knows what that observation is and which choices for the variable match the observation. We would like to apply the same intuition to IBAL. Unfortunately, however, when it comes time to sample a `dist` expression, each option could in principle be an arbitrarily complex expression, so we cannot necessarily tell whether choosing an option would violate an observation. The key principle behind our algorithm is to use the structure of a program to infer as much as possible about a `dist` expression before making a choice.

The first technique proposed for importance sampling in IBAL is a very simple one. Whenever the evaluator encounters a `dist` expression, it checks each of the options to see if it can *possibly* match the observations. It does not try to fully evaluate the option, just perform a quick and dirty check. If the check says that the option cannot possibly match the observation, the option is excluded. If the check says that the option does match the observation, or is inconclusive, the option is not excluded. When sampling the `dist` expression, only the non-excluded options are considered. The weight of the sample is then multiplied by the total probability of the non-excluded options. If none of the options can match the observation, the sample is rejected, i.e. treated as having weight 0.

The question is, what should be checked in simple checking? There is a tradeoff here. Ideally, we would fully evaluate a subexpression to determine if it could match the observation. Unfortunately that is not feasible. The whole point of sampling is to avoid having to evaluate every subexpression, and only to choose one. So the simple checking must be very cheap. We have chosen to check everything except nested `dist` expressions, `let` expressions, and function applications.

While simple checking is useful, it requires us to make observations directly about `dist` expressions. In some cases this is inconvenient and in others impossible. Our next technique, which avoids this problem, is *evidence pushing*. In evidence pushing we pass evidence from expressions to their subexpressions where appropriate. To achieve this, the sampling function now takes a third argument which is an observation on the expression. The process of passing observations to subexpressions is simple. For example, if an expression `dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$ is sampled in environment ν with observation π, we first perform simple checking to see which of the subexpressions can possibly match

π. We then choose a subexpression ϵ_i from the qualifying subexpressions according to the given probabilities. We then pass π down to ϵ_i and sample it. The reasoning is this: if it is necessarily true that the result of the `dist` expression matches π, and the `dist` choice came out the way it did, then it is necessarily true that the result of ϵ_i will match π. We are not saying that in all cases the result of ϵ_i matches π, only in cases where sampling the `dist` expression chose ϵ_i. But that is enough to push the evidence down. Similarly, for an expression `if` ϵ_1 `then` ϵ_2 `else` ϵ_3 being sampled with observation π. From the observation about the entire `if` expression we can infer nothing about the test, so ϵ_1 is sampled with observation _. But once we have discovered that the consequence is, say, ϵ_2, then ϵ_2 can be sampled with observation π. Again, we are not saying that ϵ_2 always matches π, only that when ϵ_1 comes out *true* it must match π.

Evidence pushing can produce inferences that could not be made with a knowledge-based model construction (KBMC) approach. An example is what is called type uncertainty in probabilistic relational models. An object is defined to be one of several possible classes according to some distribution. A simple program demonstrating this is

```
let f() = dist [ 0.1 : true, 0.9 : false ] in
let g() = dist [ 0.3 : true, 0.7 : false ] in
observe true in
dist [ 0.8 : f(), 0.2 : g() ]
```

In KBMC, a Bayesian network is created containing a node for objects of types `f` and `g`. A multiplexer is then used to select the appropriate type with the given distribution. In the KBMC approach, one cannot assert evidence about the underlying objects of types `f` and `g`, because that would produce a lower than correct probability of evidence. Evidence pushing solves this by first processing the `dist` expression for the result, choosing a type. If, for example, the subexpression `f()` is chosen, the observation is pushed down to the `dist` expression within `f`, and used to help make the choice there. Our algorithm with evidence pushing gets error of 0.0068 on this example, compared with 0.0364 for rejection sampling. Unless otherwise specified, in all tests programs were given 0.01 seconds to run, and results were averaged over 10,000 tests. All errors in this paper are relative errors.

A third basic technique is *conditional checking*. As stated above, we can infer nothing from the observation about an `if` expression to an observation about the test. But in fact in some cases we may be able to do that. Suppose we have an observation π about the expression `if` ϵ_1 `then` ϵ_2 `else` ϵ_3. We can use checking to see if ϵ_2 and ϵ_3 can possibly match π. Suppose ϵ_2 can match π but ϵ_3 cannot. Then we can infer the observation *true* about ϵ_1. If neither ϵ_2 nor ϵ_3 can match π we can immediately reject the sample. An example program on which conditional checking does well is

```
let f() = dist [ 0.01 : true, 0.99 : false ] in
(observe { p : 'a } in
if f()
then { p = 'a, q = dist [ 0.3 : true,
                          0.7 : false ] }
else { p = 'b, q = true }).q
```

Conditional checking finds that only the `then` clause is consistent with the observation. It infers the observation *true* on `f()`, which is used in combination with evidence pushing and simple checking to rule out the second option of the `dist` expression. On this program, the algorithm with conditional checking can get the probability of evidence exactly right, and 0.0317 error on the probability of the outcome *true*, against 0.1786 error on probability of evidence and 0.2823 error on probability of *true* outcome for the algorithm without conditional checking.

5 Delayed evaluation

Conditional checking is able to perform a "backwards" inference, from the consequences of an `if` expression to the test. There are more backwards inferences we might be able to make. Specifically, in a `let` expression, when we sample the result we might discover an observation that can be applied to the test. For example,

```
obs true in
let x = dist [ 0.1 : true, 0.9 : false ] in
x
```

From the fact that the result expression is the variable `x`, and it is observed to be *true*, we can infer the observation that the binding expression for `x` must be *true*. Similar inferences can be made from the body of a function to its arguments. In order to make these inferences, we must evaluate the result of a `let` expression before its binding, and the body of a function before its arguments.

It sounds like what we want is lazy evaluation, which is a standard technique in which evaluation is performed in precisely that order. In fact we need something different, which we call *delayed evaluation*, for several reasons. First, the purpose of lazy evaluation and delayed evaluation is quite different. In lazy evaluation, the goal is to avoid performing unnecessary computations. For us, the goal is not to avoid sampling the binding but accumulating as much evidence as possible before sampling it. Second, not sampling the binding if it is not needed would produce different semantics for the language. In the rejection sampling semantics presented in Section 2, the `let` binding is always evaluated. Thus the probability of the evidence includes the probability of observations inside

the binding. If we did not sample the binding expression when it was not needed for the result, we would not incorporate these observations into the probability of the evidence. The third reason is that the technique in the next section, evidence collection, cannot be performed with a standard implementation of lazy evaluation. With evidence collection we delay evaluating the binding variable until as much as evidence as possible has been accumulated about it. In standard lazy evaluation, a binding expression is evaluated as soon as the variable is needed, which may be too soon.

For ease of presentation, we will first present delayed evaluation without evidence collection. The basic idea is as follows. The sampling function is now allowed to return *Needed*(a, π) instead of a value and a weight. If a variable expression a is encountered with observation π, and a has not yet been assigned a value, *Needed*(a, π) is returned. Control then pops upwards until the `let` expression binding a. The binding expression for a is then sampled with observation π. Finally, the evaluation of the result expression of the `let` is resumed, with a now bound to a value.

A key point is that when the result expression of the `let` was first evaluated before the binding, a lot of work may already have been done. In particular, some decisions may have been made as to which subexpressions of `dist` expressions were chosen, and which consequences of `if` expressions were taken. These decisions cannot be made again. For this reason, when sampling an expression returns *Needed*, it also returns a *state* from which evaluation of the expression can be resumed after a has been assigned a value. A state contains a weight, representing the accumulated weight of decisions made so far in producing the state. In addition, one might expect a state to contain to an expression, but in fact it contains more than that. The same expression can have multiple associated states. For example, a `let` expression can be in one of several states, including *Let1*, the state of trying to sample the result before the binding, and *Let2*, the state of sampling the binding because it is needed for the result. States take arguments. For example, the arguments to *Let1* are the expression of the binding and the state of the result. The arguments to *Let2* are the state of the binding, the state of the result, and the observation on the binding. There is also a *Start* state that takes an expression as an argument, that corresponds to the state in which sampling the expression has not yet begun.

When an expression `let` $a = \epsilon_1$ `in` ϵ_2 is sampled, in environment ν with pattern π, it first attempts to sample ϵ_2. Otherwise, if the first sampling attempt returns *Needed*(a_1, π_1, σ_1), where σ_1 is a state, one of the following two things happens: (1) If a_1 is not equal to a, it means some free variable in the `let` expression is needed. A new state $\sigma_2 = \mathit{Let1}(\epsilon_1, \sigma_1)$ is created and *Needed*(a_1, π_1, σ_2) is returned. Once a_1 has been bound to a value, control will eventually resume with σ_2. (2) Alternatively, a_1 is equal to a, and sampling the `let` expression will attempt to sample the binding expression ϵ_1 with the observation π_1. If this returns *Needed*(a_2, π_2, σ_2), a new state $\sigma_3 = \mathit{Let2}(\sigma_2, \sigma_1, \pi_2)$ is created, and *Needed*(a_2, π_2, σ_3) is returned. Eventually a_2 is bound to a value, and sampling is resumed in σ_3. Suppose this successfully returns with a value x. a is bound to x, and sampling the result resumes from σ_1. And so on.

A similar process is used for evaluating the arguments to functions after attempting to evaluate the result. The process for other expression forms is generally simpler. For example, for `if` expressions conditional checking is first performed. This produces an observation on the test. If sampling the test returns *Needed*, an *If* state is created which stores the state of the test, the two expressions of the consequences, and the observation on the test. When control passes back to this state, it will continue sampling the test. If this succeeds one of the consequences is selected. At that point the algorithm can immediately "forget" that it was in an `if` expression, and immediately transition to the state corresponding to the consequence. The delayed evaluation process is best illustrated with an example.

```
1 let x = 'a in
2 let y = dist [ 0.2 : x, 0.8 : 'b ] in
3 observe true in
4 y |= 'a
```

The sampler begins by sampling the result of the outer `let` expression, and then the result of the inner `let` expression. Line 4 is sampled with observation *true*, which leads to `y` being needed with observation 'a. This leads to the binding of `y`, i.e. the `dist` expression in line 2, being sampled. Simple checking rules out the second possibility, so `x` is chosen and a weight of 0.2 is produced. This leads to the inner `let` expression returning *Needed*$(\texttt{x}, \texttt{'a}, \sigma_1)$, where $\sigma_1 = (\mathit{Let2}(\mathit{Start}(\texttt{x}), \sigma_2, \texttt{'a}))$, and $\sigma_2 = \mathit{Observe}(\mathit{Match}(\mathit{Start}(\texttt{y}), \texttt{'a}), \mathit{true})$. The binding of the outer `let` expression is sampled, producing `'a`, and the result is eventually *true* with weight 0.2. When run on this example, our algorithm gets the probability of evidence exactly right, while rejection sampling has 0.0203 error.

6 Evidence Collection

Consider the following program.

```
1 let x =
```

```
2   dist [ 0.01 : { p = true, q = true },
3          0.45 : { p = true, q = false },
4          0.45 : { p = false, q = true },
5          0.09 : { p = false, q = false } ] in
6 observe { r : true, s : true } in
7 { r = x.p, s = x.q }
```

Under delayed sampling, when the algorithm attempts to sample `x.p` in line 7, it will immediately return that `x` is needed, with observation $\{\texttt{p} : \textit{true}\}$. This is because it has not yet detected that `s` is `x.q`. It will then sample the `dist` expression in lines 2-5. Simple checking will rule out lines 4-5, but line 3 will still be possible. It would be nice if we could use the fact that `x.q` must also be *true* to rule out line 3 as well. This is the purpose of evidence collection.

In evidence collection, the idea is that several variables may be needed simultaneously, and the patterns on those variables may be merged from several sources. If sampling returns *Needed*, instead of a single variable and pattern it returns a list of $\langle$variable, pattern$\rangle$ pairs. When the sampling algorithm encounters a tuple expression $\{a_1 : \epsilon_1, ..., a_n : \epsilon_n\}$, it essentially evaluates all the component expressions in parallel. Each component expression ϵ_i will either return a value x_i, or *Needed*(θ_i, σ_i). If all components return values, a tuple value is returned. Otherwise, a new list θ is created which is the merging of the needs of the θ_i. A *Tuple* state σ is created in which each component a_i is associated with either x_i or σ_i, depending on whether the evaluation of ϵ_i returned a value or a state. The weight of σ is the product of the weights produced while evaluating the different ϵ_i. The algorithm returns *Needed*(θ, σ).

Now let us consider an expression `let` $a = \epsilon_1$ `in` ϵ_2. Suppose that attempting to evaluate ϵ_2 before the binding returns *Needed*(θ, σ). θ now may contain multiple variables. There are three possibilities: (1) θ contains a and no other variables. In this case the sampler immediately attempts to sample the binding expression ϵ_1, with the observation present in θ. (2) θ contains other variables and not a. In this case a *Let1* state is created and returned. (3) θ contains both a and other variables. In this case a *Let2* state is created and returned, so that when control resumes in this state the sampler will attempt to bind a.

A similar process to that used for tuple expressions is used for function arguments. When the body of a function is sampled, it may return a *Needed* list that includes many of the arguments to the function. These arguments are evaluated in parallel; the needed lists and observations for each argument are merged before passing outside.

We illustrate the operation of evidence collection on the example above. After encountering the `let` expression in lines 1-7 and the `observe` expression in lines 6-7, the tuple expression `r = x.p, s = x.q` is sampled with observation $\{\texttt{r} : \textit{true}, \texttt{s} : \textit{true}\}$. `x.p` and `x.q` are sampled in parallel. The first returns *Needed*$([\langle \texttt{x}, \{\texttt{p} : \textit{true}\}\rangle], \sigma_1)$, where $\sigma_1 =$ *Start*(`x.p`), while the second returns *Needed*$([\langle \texttt{x}, \{\texttt{q} : \textit{true}\}\rangle], \sigma_2)$, where $\sigma_2 =$ *Start*(`x.q`). The results are merged to form $\theta = [\langle \texttt{x}, \{\texttt{p} : \textit{true}, \texttt{q} : \textit{true}\}\rangle]$. The tuple expression returns *Needed*(θ, σ_3), where $\sigma_3 =$ *Tuple*$(\{\texttt{r} : \sigma_1, \texttt{s} : \sigma_2\})$. The binding of the `let` expression is eventually sampled, i.e. the `dist` expression in lines 2-5, with the observation $\{$ `p` : *true*, `q` : *true*$\}$, and the desired effect is obtained. On this example, our algorithm with evidence collection gets the probability of evidence exactly right, compared with 0.1189 error for the algorithm with delayed evaluation but without evidence collection, and 0.1590 for ordinary rejection sampling.

7 Targeted Sampling

While delayed sampling can allow many useful inferences to be made, it is still not as good as possible in some situations. The problem is that when a variable is sampled, it is sampled in its entirety. In some cases, we may sample the variable after we have collected observations for one subfield but not for another subfield. If we could delay sampling the second subfield until the observations for it have been collected, we could potentially make more inferences.

This leads to the idea of targeted sampling. An expression is sampled with a particular target, and observations relevant to that target. This allows us to sample a variable with one target early in the sampling process, avoid making commitments that are not relevant to the target, and sampling other subfields of the variable later when more observations have been collected. To make this work, the sampling algorithms now take an additional target argument t. Formally, a target is one of: $*$, meaning that everything is required; $\emptyset$, meaning that nothing is required; or $\{a_1 : t_1, ..., a_n : t_n\}$, where each t_i is a target. For any field a, $*.a$ is defined to be $*$. Also, if t is a tuple target that does not contain a field named a, $t.a$ is defined to be $\emptyset$.

The process for resuming a state *Tuple*$(\{a_1 : z_1, ..., a_n : z_n\})$, with environment ν, observation π, and target t, is as follows. Each z_i is either a value or a state. The components are sampled in parallel to produce a result y_i for each component. If z_i is a value, y_i is simply z_i. Otherwise, the algorithm checks the component $t.a_i$ of the target. If $t.a_i$ is $\emptyset$, y_i is just z_i. Otherwise, the

algorithm resumes z_i with environment ν, observation $\pi.a_i$, and target $t.a_i$. The results are merged as in Section 6.

The process of passing targets from expressions to subexpressions is simple, and is similar to the process of passing observations in evidence pushing. For example, if `if` ϵ_1 `then` ϵ_2 `else` ϵ_3 is sampled with target t, ϵ_1 is sampled with target `*` and whichever consequence is chosen is sampled with target t. In order to know what target is needed on a variable, the *Needed* list is now modified so that each variable in the list is associated with both an observation and a target. If the algorithm samples a variable expression a, with environment ν, observation π and target t, and the binding of a in ν has not been fully resolved on t, it returns $\mathit{Needed}([\langle a, \pi, t\rangle])$. Useful non-`*` targets are produced by dot expressions. If $\epsilon.a$ is sampled with target t, then ϵ is sampled with target $\{a : t\}$. Subsequently, when sampling ϵ, only the a subfield will be sampled.

The process for sampling `let` expressions is a little different from before. The difference is that before, when the binding expression was sampled, it was sampled in its entirety to produce a value, and the variable bound to it. Thus it was guaranteed that when the result was sampled again, the variable would have a value, and the binding expression would never need to be sampled again. Now, when the binding expression is sampled, it is only sampled for a particular target. This may produce a value, or it may produce a *Tuple* state in which only some components are resolved to a value. Thus it is possible, when sampling the result again, that it will be discovered that more of the variable is needed, so the binding expression must be sampled again on a broader target. The algorithm thus alternates between sampling the result and sampling the binding expression, gradually refining the binding until it contains enough information to fully sample the result. Then, after the result has produced a value, the binding expression is sampled again with target `*` (if it has not yet produced a value) to account for any remaining observations within it. Similar ideas apply to function applications. The algorithm alternates between sampling the body and sampling the arguments.

A type of model in which targeted sampling really comes into its own is a probabilistic context free grammar (PCFG).

```
 1 let append(y,z) =
 2   if y |= []
 3   then z
 4   else y.CAR :: append(y.CDR, z) in
 5 let noun() =
 6   [ dist [ 0.4 : 'flies, 0.6 : 'ants ] ] in
 7 let np() =
 8   dist [ 0.7 : noun()
 9          0.3 : append(noun(), np()) ] in
10 let x =
11   observe 'flies :: _ in
12   np() in
13 x |= _ :: 'ants :: _
```

The sampler quickly discovers that it must evaluate `np` with observation $\pi_1 = \{\texttt{TAG} : \texttt{'CONS}, \texttt{CAR} : \texttt{'flies}, \texttt{CDR} : \_\}$ and target $t_1 = \{\texttt{TAG} : *, \texttt{CAR} : *, \texttt{CDR} : \{\texttt{TAG} : *, \texttt{CAR} : *, \texttt{CDR} : \emptyset\}\}$. When the `dist` expression in lines 8-9 is sampled, the second choice `append(noun(),np())` is randomly chosen, and sampled with the same pattern and target. It is then discovered in line 2 that `y.TAG` is needed to compare it to `[]`. Thus $\mathit{Needed}([\langle \texttt{y}, \_, t_2\rangle], \sigma_1)$ is returned, where $t_2 = \{\texttt{TAG} : *\}$ and σ_1 is a state representing the match expression in line 2. This results in the sampler evaluating the arguments to `append`. In particular, `noun()` is sampled with target t_2. This results in the expression in line 6 being sampled. Note that here the `dist` expression is inside a list. This is important, because it means the sampler can tell that the `TAG` is `'CONS` before making a commitment to a choice. Since the target only calls for `TAG`, the sampler immediately returns with $\mathit{Needed}([], \sigma_2)$, where $\sigma_2 = \mathit{Tuple}(\{\texttt{TAG} : \texttt{'CONS}, \texttt{CAR} : \mathit{Start}(\texttt{dist [0.4 : 'flies, 0.6 : 'ants]}), \texttt{CDR} : \mathit{Start}(\texttt{[]})\})$. The sampler goes back to evaluating the body of `append`, finding that the `else` clause in line 4 is taken. By evidence pushing, it continues to have the pattern π_1, which was the pattern on the whole `if` expression. This is a tuple construction, so all the fields are sampled in parallel. It is discovered that both `y.CAR` and `y.CDR` are required, so the sampler evaluates those fields. In particular, it evaluates `y.CAR`, whose state is $\mathit{Start}($`dist [ 0.4 : 'flies, 0.6 : 'ants ]`$)$, with the pattern π_1.`CAR`, which is `'flies`. By simple checking the result is forced to be `'flies`, and a weight of 0.4 is obtained. And so on. We ran the different algorithms on a larger PCFG, taken from [Charniak, 1993]. All methods were given 1 second for their reasoning, and results are averaged over 100 tests. Our algorithm obtained 0.0254 error for probability of evidence, while rejection sampling obtained 0.1320. For the probability of predicting the next word in a sequence, our algorithm obtained 0.0275 error, versus 0.1355 for rejection sampling.

8 Results on Music Model

Everything comes together in the music model of Section 3. We tested it on the transformation of motives in Beethoven's early piano sonatas. The first motive in six movements was taken as the source motive. A

later motive in each movement was chosen as the destination. The destination motives were chosen so that the human ear can easily tell that they are closely related to the source motive, but several operations are required to transform the source to the destination. In each case, we asserted an observation in the program that transforming the first motive produced the second. The tasks featured a relatively low probability of evidence. To test the algorithms, we examined if they could classify each of the destination motives as coming from the correct source motive. In each test, in which the algorithms were given 30 seconds of reasoning time, correct classification was achieved if the probability of evidence for the correct transformation was greater than for the incorrect ones. The results are shown in the following table. The column labeled L is the length of each motive; P(e) is the approximate probability of the correct transformation; Rej is the percentage of tests in which rejection sampling gave the correct answer; and Imp is the percentage of tests in which the importance sampling algorithm of this paper gave the correct answer. The results are aggregated over 100 tests.

Motive	L	P(e)	Rej	Imp
1	10	5×10^{-7}	8	90
2	9	1×10^{-8}	0	100
3	6	2×10^{-7}	0	91
4	18	2×10^{-12}	0	99
5	10	1×10^{-10}	0	100
6	6	7×10^{-7}	7	86

The results are dramatic. Rejection sampling is hopeless, because it returns probability zero for almost everything. Our algorithm does very well on all of the examples. Exact inference would have been infeasible because it is exponential in the length of the source motive. While this is just six motives, it is a proof of concept both that IBAL can be used to implement interesting new models and that the methods of this paper can make it applicable to real problems. We are not claiming that the music model as it stands is a great model — much more work needs to be done. We are only claiming that it is a model of realistic complexity to test the language and algorithm.

9 Conclusion

This paper has developed an importance sampling algorithm for IBAL, a highly expressive probabilistic modeling language. It is based on the principle of using the structure of a model to try to infer as much as possible about a variable before committing to a particular value. This principle is a general one, and should apply to other first-order probabilistic modeling languages. How this is carried out will vary from language to language, depending on the available structures. IBAL provides a particularly rich set of structures, so it is a good language in which to perform this kind of reasoning. The paper laid out a number of techniques for performing this reasoning, and showed an example of how IBAL can be used to create real and interesting models, and how the algorithm can be applied to them. We believe that the principle of reasoning directly about a model rather than reducing it to a Bayesian network is important, and will have applications beyond importance sampling. In particular, it would be very useful to see if it can be used to guide MCMC algorithms, perhaps by making it possible to automatically determine proposal distributions.

References

[Charniak, 1993] E. Charniak. *Statistical Language Learning*. MIT Press, 1993.

[Fung and Chang, 1989] R. Fung and K.-C. Chang. Weighing and integrating evidence for stochastic simulation in Bayesian networks. In *Uncertainty in Artificial Intelligence*, 1989.

[Gilks *et al.*, 1996] W.R. Gilks, S. Richardson, and D.J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, 1996.

[Henrion, 1988] M. Henrion. Propagation of uncertainty by probabilistic logic sampling in Bayes' networks. In *Uncertainty in Artificial Intelligence*, 1988.

[Kersting and de Raedt, 2000] K. Kersting and L. de Raedt. Bayesian logic programs. In *Proceedings of the Work-In-Progress Track at the 10th International Conference on Inductive Logic Programming*, 2000.

[Laskey and Costa, 2005] K. B. Laskey and P. Costa. Of klingons and starships: Bayesian logic for the 23rd century. In *Uncertainty in Artificial Intelligence*, 2005.

[Milch and Russell, 2006] B. Milch and S. Russell. General purpose MCMC inference over relational structures. In *Uncertainty in Artificial Intelligence*, 2006.

[Pfeffer, To appear] A. Pfeffer. The design and implementation of IBAL: A general purpose probabilistic language. In L. Getoor and B. Taskar, editors, *Statistical Relational Learning*. MIT Press, To appear.

[Shachter and Peot, 1989] R. D. Shachter and M. A. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Uncertainty in Artificial Intelligence*, 1989.