CTPPL: A Continuous Time Probabilistic Programming Language

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

Abstract

Probabilistic programming languages allow a modeler to build probabilistic models using complex data structures with all the power of a programming language. We present CTPPL, an expressive probabilistic programming language for dynamic processes that models processes using continuous time. Time is a first class element in our language; the amount of time taken by a subprocess can be specified using the full power of the language. We show through examples that CTPPL can easily represent existing continuous time frameworks and makes it easy to represent new ones. We present semantics for CTPPL in terms of a probability measure over trajectories. We present a particle filtering algorithm for the language that works for a large and useful class of CTPPL programs.

1 Introduction

Probabilistic programming languages (e.g. IBAL [Pfeffer, 2007], BLOG [Milch, 2006], Church [Goodman *et al.*, 2008]) are an exciting development in probabilistic knowledge representation. They allow a modeler to build probabilistic models using complex data structures with the full power of programming languages. Most probabilistic programming languages that have been developed are static, but ProPL [Pfeffer, 2005] and D-BLOG [de Salvo Braz *et al.*, 2008] extend such languages to dynamic models. However, they both use discrete time.

There are a number of arguments for using continuous time models. First, most real-world processes take place in continuous time; discretization is an artificial imposition to force them into a discrete time framework. Second, when we model a process in discrete time, we have to make the discrete time increments fine enough to capture all the important events, but most of the time reasoning at such a fine time granularity is unnecessary. In the context of probabilistic programming languages, there is a third argument. When we model a process in continuous time, it becomes natural to make the time taken by a subprocess be a variable which can be specified using the full power of the language. In short, time becomes a first class element of the language. In contrast, in discrete time representations, usually the world proceeds from one time step to the next, and the time between them is fixed.

In this paper we present CTPPL (pronouced "Cat People"), a continuous time probabilistic programming language. CTPPL is similar to ProPL, but is different in four main ways: (1) values in CTPPL are discrete or continuous; (2) time is continuous; (3) probability and time are first class elements of the language; and (4) inference is conducted by sampling trajectories through a state space, rather than constructing a giant dynamic Bayesian network. We first present the syntax of the language and some examples. When combining the expressive power of probabilistic programming with continuous time, both semantics and inference become challenging. In Section 4 we present the semantics of a CTPPL program as defining a probability measure over a space of equivalence classes of trajectories through state space, where the trajectories are partly discrete and partly continuous. In Section 5 we present a particle filtering algorithm that works for a large and useful class of CTPPL programs. At the core of our algorithm is an importance sampling algorithm that guarantees that if the set of particles at one time point is consistent with the next set of observations that are received, then with positive probability the algorithm will generate a new particle with positive weight.

2 The Language

The basic unit of the CTPPL language is the expression, which describes a computation that executes stochastically in time while producing a value. While it is executing, it may produce emissions, which are observations that happen at particular points in time. Expressions can be any one of the following forms:

c	Constant
n	Variable
`if` e_1 `then` e_2 `else` e_3	Conditional
`lambda` $n.e$	Function definition
$e_0(e_1)$	Application
$e_1 \oplus e_2$	Binary operator
$\{e_1, e_2\}$	Pair construction
`left` e	Component extraction
`right` e	Component extraction
`flip` e	Discrete probabilistic choice
$q(e)$	Continuous probabilistic choice
`first` $[e_1, e_2]$	First expression

`emit` $e_1; e_2$	Emission
`delay` $e_1; e_2$	Delay expression
`now`	Now expression

Constants can be boolean, string, integer or floating point. While variables, conditionals, function definition, application, pair construction and component extraction are familiar, some explanation is needed of how they specify an execution in time. In a conditional `if` e_1 `then` e_2 `else` e_3, the test e_1 is first evaluated. After it has completed, either e_2 or e_3 is evaluated, depending on the result of e_1. In a function application $e_0(e_1)$, e_0 and e_1 are evaluated in parallel to produce values v_0 and v_1. After they have both completed, the body of v_0 is evaluated with its formal argument bound to v_1. For binary operators, the two arguments are evaluated in parallel. Once they have terminated the result is produced immediately. In a pair construction $\{e_1, e_2\}$, e_1 and e_2 are evaluated in parallel. The time of completion of the entire pair construction is the later of the completion times of e_1 and e_2.

Discrete probabilistic choice is familiar from other probabilistic programming languages such as IBAL and Church: `flip` e means evaluate e to produce a non-negative floating point value v, and then with probability v return `T`, otherwise return `F`. Note that unlike IBAL and ProPL, but like Church, the probability is defined by an expression, so is a first class element of the language. Continuous probabilistic choice is new to CTPPL. Here q is a probabilistic primitive that, given a parameter, defines a probability density function. The meaning of $q(e)$ is to first evaluate e to produce a value v for the parameter, and then generate a value from the density function defined by q given the parameter value v. The language is fully general in allowing any primitive, as long as (1) we can generate samples from it, and (2), we can compute its density at any point.

The meaning of `first` $[e_1, e_2]$ is to begin evaluating e_1 and e_2 in parallel. As soon as either of them completes with a value v, the entire first expression completes with value v. Ties are broken in favor of e_1. The meaning of `emit` $e_1; e_2$ is to evaluate e_1 to produce a value v, completing at time t. v is then emitted at time t. Finally e_2 is evaluated, beginning at time t, to produce the result of the expression.

Passage of time is specified through delay expressions. The meaning of the expression `delay` $e_1; e_2$ is to first evaluate e_1 to produce positive floating point value v, terminating at time t. Then e_2 is evaluated, beginning at time $t + v$. In other words, there is a delay of v between the time e_1 ended and the time e_2 begins. The expression e_1 is called the *delaying subexpression* of the delay expression. While delay expressions are present in ProPL, they are different in CTPPL. The two differences are that the delay time is continuous, and the delay time is specified by an arbitrary expression (in ProPL, it is specified by an integer constant). Thus the delay is a first class element of the language. The final construct, which further makes time a first class element of the language, is `now`, which always returns immediately, evaluating to the current time at the time of its evaluation.

In addition to the above bare-bones language, CTPPL provides a good deal of syntactic sugar. One of these is the expression `dist` $[e_1^p : e_1^r, ..., e_n^p : e_n^r]$, which is a generalization of discrete probabilistic choice. The meaning is to evaluate $e_1^p, ..., e_n^p$ in parallel to produce a set of non-negative floating point values, normalize the values to obtain probabilities $p_1, ..., p_n$, choose one of the e_i^r with probability p_i, and then evaluate e_i^r to produce the result. Others include let expressions, in which a variable is bound to a value to be used in a result expression; case expressions, analogous to switch expressions in C; functions of many arguments; tuples of many components; first expressions with many subexpressions; records with named fields; and lists with supporting functions. We will use syntactic sugar freely in the examples. However, we will restrict attention to the core language when defining the semantics and inference algorithm.

We impose the following restrictions on models: (1) In an emission, the emitted value cannot be floating point. (2) In a delay expression, the delaying subexpression must be continuously distributed *at the time of evaluation*, with no points of positive probability mass; for example, the delaying subexpression in `let x = uniform(1.0) in delay x; 1` is not continuously distributed, because at the time of evaluation `x` is bound to a value. (3) Emissions are not allowed to be produced while executing a delaying subexpression. (4) Delays must be positive. (5) Floating point values cannot be compared for equality, only inequality. (6) Floating point arithmetic binary operators are required to be invertible. We also assume that at any time point, with probability 1 the process will either progress to another time point or terminate. Even with these restrictions, the language allows many useful and interesting processes to be represented, and the expressive power goes far beyond that of any existing language.

3 Examples

In recent years there has been a flurry of interest in continuous time models, mostly focused on continuous time Bayesian networks (CTBNs) [Nodelman, 2007]. CTBNs are built on homogenous Markov processes. A homogenous Markov process is a finite state, continuous time process, consisting of an initial distribution P^0 and intensity matrix

$$Q = \begin{bmatrix} -q_1 & q_{12} & \cdots & q_{1n} \\ q_{21} & -q_2 & \cdots & q_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n1} & q_{n2} & \cdots & -q_n \end{bmatrix}$$

The meaning is that if the process in state i, it stays in state one for an amount of time governed by an exponential distribution with parameter q_i, then transitions to q_j with probability q_{ij}/q_i. We can write this in CTPPL using

```
x() = dist [P_1^0 : x1(), ..., P_n^0 : xn()]
x1() = delay exp(q_1);
  dist[q_12/q_1 x2(), ...,q_1n/q_1 : xn()]
...
```

Some state transitions can produce emissions.

In a CTBN, each variable has a conditional intensity matrix Q^u for every combination of values u of its parents. To capture CTBNs, we use three helper

functions that are easy to write and are not shown. `change_ith(list, i, value)` changes the `i`-th element of `list` to `value`, keeping the rest as in `list`. `extract_indices(list, indices)` returns a tuple consisting of the elements at the given indices of the input list. `first_list_i(f,list)` is a higher order function that takes a function `f`, applies it to each member of `list`, and returns the first produced result (like a `first`) expression. The function `f` takes two arguments, the element and the index of the element in `list`. In a CTPPL model of a CTBN, each variable has a process of the form:

```
x0(previous) =
 case {previous.x1,extract_indices([2,4])}
 of {1,u} ->
  delay exp(q1^u);
    dist[q12^u/q1^u:  2,..., q1n^u/q1^u:  n]
...
```

Here $[2, 4]$ are the indices of the parents of `x0`. Again, some transitions can produce emissions. A list of variable processes is passed to the `ctbn` function, which works with arbitrarily many variables. The `init()` function is a specification of a Bayesian network defining the initial state.

```
ctbn(variables) =
 let process(previous) =
  process
   (first_list_i
    (lambda (v,i) .
     change_ith(previous, i, v(previous))),
    variables)
 in process(init())
```

Some authors [Gopalratnam *et al.*, 2005; Nodelman *et al.*, 2005] have studied how to extend CTBNs to allow Erlang-Coxian and phase distributions for the delays. This can easily be done in CTPPL since delaying subexpressions can be arbitrary expressions.

CTPPL goes well beyond CTBNs in two important ways. The first is that we can define processes over arbitrarily complex data structures. The second is that time is a first class element of the language. So, for example, we could have dependent delays, as in

```
let x = uniform(2.0) in
 {delay exp(x) in ``a'',
  delay exp(x+1) in ``b''}
```

or we could have a delay dependent on the value produced by the delay expression, as in

```
let x = uniform(2.0) in
 delay exp(x) in x
```

To illustrate the power of these two ideas in combination, we develop a model of musical performance. We begin with a simple model of monophonic performance, following [Cemgil and Kappen, 2003]. The `play` function takes as arguments the current tempo, and the input, which is a list of $\{\text{time}, \text{pitch}\}$ pairs. The time element is the musical duration before which the pitch should be played. In the following code, `posnormal` denotes a truncated normal distribution, with the given mean and variance, constrained to be positive. Note that both the delay and the variance of the new tempo depend on the duration.

```
play(tempo,input) =
 if empty(input)
 then true // termination
 else
  let duration = left head(input) in
  delay posnormal({duration * tempo,
    0.01});
  emit right head(input);
  let new_tempo =
   posnormal({tempo,
     0.04 * duration * duration}) in
  play(new_tempo, tail(input))
```

In CTPPL it is easy to generalize this model to arbitrarily many voices. With multiple voices the performed pitches of the voices are interleaved. The following code uses the standard higher-order function `map_i`, which is similar to `first_list_i` above except that it maps a two-argument function over all elements of a list. `filter(non_empty, new_input)` returns `new_input` with all empty lists removed. The idea behind the code is that after each emission, it attempts, in parallel, to process each voice, generating the time of the next emission for that voice, and the transformed input after the voice has played its next note. In this input transformation, produced by mapping the `helper` function over the old input, the voice itself is replaced with its tail, because its note has been played, whereas for all the other voices the duration until the played note is subtracted from the time to the first note. Now, all this is done in parallel for each of the voices, but only the first to produce its emission is actually used. In this way, the emissions of all the voices are interleaved.

```
play(tempo, input) =
 if empty(input) then true else
 let process(l,i) =
  let duration = left head(l) in
  delay posnormal({duration * tempo,
    0.01});
  emit right head(l);
  let helper(m,j) =
   if j == i
   then tail m
   else {left head(m) - duration,
      right head(m)} ::  tail(m) }
  in
  let new_input = map_i(helper, input) in
  let new_tempo =
   posnormal({tempo,
     0.04 * duration * duration}) in
  {new_tempo, filter(non_empty, new_input)}
 in
 let { new_tempo, new_input } =
   first_list_i(process, input) in
 play(new_tempo, new_input)
```

4 Semantics

The semantics of a CTPPL program is a probability measure over a set of equivalence classes of trajectories through state space. A *state* consists of a CTPPL expression, a time stamp, and a set of marks on subexpressions of the expression. Intuitively a state is a point in the execution of a process. The expression defines the process currently being executed. The time stamp is the current time. The marks indicate which subexpressions of the expression are being processed. Since execution can occur in parallel, there may be multiple marks. Our notation for a state is a pair of an expression and a time stamp, with a bullet in front of each of the marked subexpressions. For example $(\texttt{if}\ \bullet\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, 3)$ is the state consisting of the `if` expression, with time stamp 3, with a mark on e_1.

States can transition to other states. There are three kinds of transitions: *free transitions*, which do not involve probabilities or time delays; *probabilistic transitions*, which involve probabilistic choice but no time delays; and *temporal transitions*, in which the time stamp of the state changes. Free transitions are expressed using rewrite rules. Since the time stamp does not change, we omit the time stamp part of the state in the following rules. In the following transition rules, v denotes an expression which directly specifies a value, which could be a constant, a lambda expression, or a pair of values. Also the notation w denotes an expression that does not directly specify a value. The notation $e[x/v]$ denotes the expression produced from replacing all unshadowed occurences of x in e with v. The free transition rules are:

$\bullet\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	$\rightarrow$	$\texttt{if}\ \bullet\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
$\texttt{if}\ \bullet\ \texttt{T}\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	$\rightarrow$	$\bullet e_2$
$\texttt{if}\ \bullet\ \texttt{F}\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	$\rightarrow$	$\bullet e_3$
$\bullet(e_0(e_1))$	$\rightarrow$	$(\bullet e_0)(\bullet e_1)$
$\bullet(\texttt{lambda}\ n.e)(\bullet v)$	$\rightarrow$	$\bullet e[n/v]$
$\bullet(e_1 \oplus e_2)$	$\rightarrow$	$(\bullet e_1) \oplus (\bullet e2)$
$(\bullet v_1) \oplus (\bullet v_2)$	$\rightarrow$	$\bullet(v_1 \oplus v_2)$
$\bullet\{e_1, e_2\}$	$\rightarrow$	$\{\bullet e_1, \bullet e_2\}$
$\{\bullet v_1, \bullet v_2\}$	$\rightarrow$	$\bullet\{v_1, v_2\}$
$\bullet\texttt{left}\ e$	$\rightarrow$	$\texttt{left}\ \bullet\ e$
$\texttt{left}\ \bullet\ v$	$\rightarrow$	$\bullet x$ where $v = \{x, y\}$
$\bullet\texttt{right}\ e$	$\rightarrow$	$\texttt{right}\ \bullet\ e$
$\texttt{right}\ \bullet\ v$	$\rightarrow$	$\bullet y$ where $v = \{x, y\}$
$\bullet\texttt{flip}\ e$	$\rightarrow$	$\texttt{flip}\ \bullet\ e$
$\bullet q(e)$	$\rightarrow$	$q(\bullet e)$
$\bullet\texttt{first}\ [e_1, e_2]$	$\rightarrow$	$\texttt{first}\ [\bullet e_1, \bullet e_2]$
$\texttt{first}\ [\bullet v_1, \bullet e_2]$	$\rightarrow$	$\bullet v_1$
$\texttt{first}\ [\bullet w_1, \bullet v_2]$	$\rightarrow$	$\bullet v_2$
$\texttt{delay}\ \bullet\ 0; e$	$\rightarrow$	$\bullet e$
$\bullet\texttt{emit}\ e_1; e_2$	$\rightarrow$	$\texttt{emit}\ \bullet\ e_1; e_2$
$\texttt{emit}\ \bullet\ v_1; e_2$	$\rightarrow$	$\bullet e_2$, while emitting v_1

The expression `now` also produces a free transition, but it depends on the time. The rule is $(\bullet\texttt{now}, t) \rightarrow (\bullet t, t)$.

Probabilistic transitions specify a probability distribution or probability density function over next states. When executing a probabilistic transition, the process transitions to another state with the given probability or probability density. The probabilistic transition rules are:

$$\begin{array}{lcl} \texttt{flip}\ \bullet\ v & \rightarrow & \begin{cases} \texttt{T} & \text{with probability } v \\ \texttt{F} & \text{with probability } 1 - v \end{cases} \\ q(\bullet v) & \rightarrow & x \text{ with density } q(v)(x) \end{array}$$

Temporal transitions are continuous. A temporal transition begins in a *time-terminal state*, which is a state in which no free or probabilistic transitions are possible. For example,

```
(delay (delay • 2.0; exp(3.0));
  if flip 0.5 then ``a'' else ``b'',
7.2)
```

is a time-terminal state. Note that when there are nested delays, not all the delaying subexpressions in a time-terminal state need specify a value. For example, in `delay (delay 2.0; uniform(3.0)); 1`, only the inner delay's delaying subexpression specifies a value. If a time-terminal state contains no delaying subexpressions, no temporal transition is possible. Otherwise, define the *minimum delay time* of a time-terminal state to be the minimum of the values specified by its value specifying delaying subexpressions. Let (e, t) be a time-terminal state with minimum delay time δ. The process transitions through a continuum of states (e_u, u) for $u \in [t, t + \delta]$, where e_u is the same as e except that every value specifying delaying subexpression v is replaced with $v - (u - t)$. At time $t + \delta$, one of the delaying subexpressions will be 0, so a free transition is available and the process can continue.

No transitions are allowed on expressions that specify a value. The rules for applying the transitions are as follows:

1. Let $i = 0$, and let $t_0 = 0$.
2. Execute free and probabilistic transitions at t_i that are not applied to delaying subexpressions until no more are available. Transitions are performed in a depth-first manner, but the order in which parallel subexpressions are resolved is arbitrary, except that `first` subexpressions must be resolved in the order in which they appear.
3. If there are no delaying subexpressions, the expression must specify a value, and the process terminates.
4. Otherwise, let the state be called the *pre-delay-resolution state (PDRS)* at time t_i.
5. At this point, the delaying subexpressions are resolved one by one using free and probabilistic transitions. If there are no nested delays, a delaying subexpression will resolve to specify a value. If there are nested delays, the outer delaying subexpressions will not resolve to a value, but eventually some of the nested delays will.
6. Finally, we reach a time terminal state at t_i and execute a temporal transition through $[t_i, t_{i+1}]$. We set $i \leftarrow i + 1$ and continue with step 2.

A trajectory consists of a discrete sequence of states at time $t_0 = 0$, followed by a continuum of states up to time t_1, followed by another discrete sequence of states at time t_1, followed by another continuum of states up to time t_2, and so on. A trajectory also determines a sequence of emissions, defined as a list $[t_0 : m_0, t_1 : m_1, ...]$, where m_i is the set of

emissions that happen at time t_i. The trajectory and emission sequence may or may not terminate.

We define an equivalence relation on trajectories, saying that two trajectories are equivalent if they contain the same transitions, except that parallel subexpressions may be resolved in different orders. Equivalent trajectories have the same PDRSs, time-terminal states, temporal transitions and emission sequence. The rules specified above define a probability measure over equivalence classes of trajectories. The elementary sets consist of sets of equivalence classes containing the same free transitions, the same discrete probabilistic transitions, and in which the continuous probabilistic transitions fall in an interval. The density of an equivalence class is the product of the probabilities of its discrete probabilistic transitions and the densities of its continuous probabilistic transitions. Because all members of an equivalence class have the same probabilistic transitions, the probability of an equivalence class is equal to the product of the probabilities and densities of the probabilistic transitions in any member. In the next section, we will use the term *path* to describe a contiguous sub-trajectory of a trajectory, and we will use the term "probability of a path" to denote the product of the probabilities and densities of the probabilistic transitions in the path.

This semantics is well able to handle cases where a potentially unbounded number of processes happen in parallel. For example, consider the program:

```
let f(x) =
 delay exp(1);
 first [delay exp(2); x, f(x+1)]
in f(0)
```

We will first expand `f(0)` to transition to the PDRS

```
 delay • exp(1);
 first [delay exp(2); 0, f(1)]
```

We will resolve the delaying subexpression `exp(1)`, choosing a value, say 0.7. We will then take a temporal transition to time 0.7, and then transition to `• first [delay exp(2); 0, f(1)]`. Next, we will move the mark inside the first expression, and expand `f(1)` to transition to the PDRS

```
first [delay • exp(2);0,
       delay • exp(1);
       first [delay exp(2); 1, f(2)]]
```

Let's say we sample 1.3 and 0.5 for the two delaying subexpressions. We will take a temporal transition to time 1.2 and the expression

```
first [delay • 0.8; 0,
       • first [delay exp(2); 1, f(2)]]
```

Now we will move the second mark inward and expand `f(2)` to reach another PDRS

```
first [delay • 0.8; 0,
       first [delay • exp(2); 1,
          delay • exp(1);
          first [delay exp(2); 2, f(3)]]]
```

Now we will resolve the latter two delaying subexpressions. Let's say we get 0.9 and 1.2. We will take a temporal transition to time 2.0 and transition to

```
first [• 0,
     first [delay • 0.1; 1,
          delay • 0.4;
          first [delay exp(2); 2, f(3)]]]
```

By the rules for first expressions, this resolves to 0. Even though the number of parallel processes is unbounded, in any given run the process will terminate in a finite amount of time. Each such finite run has a well-defined probability of happening.

By contrast, consider the program

```
let f(x) =
   first[ delay exp(0); x, f(x+1)]
in f(0)
```

This is a divergent program. Infinitely many parallel branches will be expanded before any delaying subexpression is resolved. Our semantics does not handle such a program, and it is disallowed.

5 Inference

Our goal is to monitor the state of the system over time based on observations. Our observations consist of a stream of emissions. At each time point $i = 0, 1, ...$, we want to estimate the probability distribution over the PDRS at time t_i given the sequence of emissions up to time t_i. We choose the PDRS because of two properties: (i) no more emissions are possible at t_i after the PDRS has been reached (because of our restriction that executing a delaying subexpression is not allowed to produce emissions); and (ii), no delaying subexpressions have been resolved, so stopping at the PDRS is making a minimal commitment about what time t_{i+1} will be. Formally, we want to estimate $P(S(t_i)|m_0, ..., m_i)$, where $S(t_i)$ denotes the PDRS at time t_i.

We will estimate the distribution at time t_i using a set of particles. As usual, we proceed recursively, beginning with an initial estimate of $P(S(0)|m_0)$, and then recursively estimating $P(S(t_i)|m_0, ..., m_i)$ by repeatedly choosing a particle s_{i-1} from $P(S(t_{i-1}))$, and sampling a particle s_i from $P(S(t_i)|S(t_{i-1}) = s_{i-1}, m_i)$.

One might think that the simplest way to do this is to use rejection sampling. We can define a sampling process using the transition rules from Section 4. We begin by choosing a PDRS s_{i-1} at random from our previous set of particles at time t_{i-1}. We then resolve the delaying subexpressions until we reach a time-terminal state with minimum delay time δ. Let t' be $t_{i-1} + \delta$. We then reason as follows:

1. If $t' < t_i$, we complete the temporal transition to time t'. We then execute some free and probabilistic transitions until we reach a PDRS at time t'. If in the process an emission is produced, we reject the sample because according to our observations there are no emissions between t_{i-1} and t_i. Otherwise, we continue the process beginning with the new PDRS.

2. If $t' = t_i$, we complete the temporal transition to time t_i, perform some more free and probabilistic transitions, and finally arrive at a PDRS s_i at time t_i. We check to see if the emissions m_i have been produced. If so, we accept the sample and add s_i to the set of samples at time t_i. If not, we reject the sample.
3. If $t' > t_i$, we can immediately reject the sample. This is because no emissions are possible in the middle of a temporal transition.

The problem with this approach is that we can only accept a sample if $t' = t_i$. Since delay times are continuously distributed, this has probability zero of happening. Essentially the problem is that rejection sampling cannot be used when conditioning on events of measure zero. Instead, we use a particle filtering approach. Ng et al. [Ng *et al.*, 2005] present a particle filtering method for continuous time, hybrid state processes. Fan and Shelton [Fan and Shelton, 2008] present importance sampling and particle filtering algorithms for CTBNs which form the basis for our approach. As in standard particle filtering, we begin with an unweighted set of samples at time t_{i-1}. We choose a sample s_{i-1} at random from this set and use importance sampling to generate a weighted particle s_i at time t_i. We then resample the weighted particles to produce a set of unweighted particles which is our estimate $P(S(t_i)|m_0, ..., m_i)$.

5.1 Importance Sampling

The importance sampling algorithm we use is actually an importance/rejection algorithm, since rejections are allowed. However, we will guarantee that if m_i has positive probability given our estimate of $P(S_{i-1}|m_0, ..., m_{i-1})$, then with positive probability a new sample s_i will be generated with positive weight. The main idea of our algorithm is that some delays are *key delays*. Suppose we begin with a PDRS (e, t). We proceed to resolve the delaying subexpressions. At the time we process the last delaying subexpression e', let the minimum delay time of all the delaying subexpressions that have already been resolved to specify a value be δ, and let $t' = t + \delta$. If $t' \leq t_i$, then we know at the time we process e' that the temporal transition will take us to a time at or before t', which is no later than t_i. So we need place no constraints on the resolution of e'. If, on the other hand, $t' > t_i$, we say that e' is a key delay. We know that if e' resolves to a value $\delta' > t_i - t'$, we will be forced to reject the sample. Therefore, when we encounter a key delay e' we use the following procedure.

1. Sample a resolution of e'.
2. If resolving e' results in one or more nested delays, let e'' be the delaying subexpression in e' that is the last to be processed. We can reason about e'' in the same way as e' and continue the sampling process.
3. Otherwise, resolving e' produces a value δ'. Let $t'' = t + \delta'$.
4. If $t'' < t_i$, complete the temporal transition to t'' and execute free and probabilistic transitions to reach a PDRS s''. If emissions are produced, reject. Otherwise, continue sampling from s''.
5. If $t'' \geq t_i$, force t'' to be equal to t_i, compensating with an importance weight w. Complete the temporal transition to t_i and execute free and probabilistic transitions to reach a PDRS s_i. If the emissions m_i are produced, add s_i with weight w to the set of weighted particles at t_i. Otherwise reject the sample.

There is a subtlety related to parallel execution. If we always resolve the delaying subexpressions in the same order, some subexpressions will never be key delays. If the observed emissions are the result of those delays completing, we will never get an accepted sample. In technical terms, the proposal distribution will not be positive everywhere the true distribution is. To get around this problem, we randomly choose the order in which delaying subexpressions are resolved.

To derive the importance weight, we reason as follows: Let s be the state just before the key delay is resolved. Let s' be the state in which the key delay is resolved to specify the value $\delta' = t_i - t$. Let s'' be the state with time stamp t_i at the end of the temporal transition from s'. The probability density of a path from s_{i-1} through s, s' and s'' to s_i can be written as $p(s_{i-1} \rightsquigarrow s)p(s \rightsquigarrow s')p(s'' \rightsquigarrow s_i)$, where $s_1 \rightsquigarrow s_2$ denotes the event the s_1 leads to s_2 through some sequence of transitions, because $p(s' \rightsquigarrow s'')$ is 1. Now consider our proposal distribution. We sample the path from s_{i-1} to s normally, and similarly the path from s'' to s_i. However, we force s to lead to s'. This happens with probability $P(\phi)$, where ϕ is the event that resolving s does not lead to nested delays and produces a value $\delta \geq t_i - t$. Thus the proposal probability density of the path is $p(s_{i-1} \rightsquigarrow s)P(\phi)p(s'' \rightsquigarrow s_i)$. Therefore the importance weight, which is the true density divided by the proposal density, is $p(s \rightsquigarrow s')/P(\phi)$. $P(\phi)$ can easily be estimated using forward sampling. It remains to specify how to estimate $p(s \rightsquigarrow s')$.

5.2 Regression

Our algorithm for estimating $p(s \rightsquigarrow s')$ works by "regressing" the target value δ through the expression of state s. Intuitively, the regression process takes an expression e and a target value δ, samples values for some subexpressions of e, and computes what target value δ' for the remaining subexpression e' would allow e to produce δ. We say that the computation regresses to e'. The choice of which subexpressions are sampled and which are regressed to is sometimes randomized.

Due to the restrictions on the language, we only need to handle some of the kinds of expression. Emissions do not need to be handled because delaying subexpressions may not produce emissions. Flip expressions also do not need to be handled because they produce a Boolean, not a floating point number, and similarly lambda expressions produce functions, not floating points. For delay expressions, meanwhile, the density is 0, because s cannot possibly lead to s', which has the same timestamp as s.

Constants, variables, and now expressions are not continuously distributed, and we require that delay times be continuously distributed at the time of evaluation. Therefore such expressions cannot appear at the top level of a delaying subexpression. However, they may appear as subexpressions of a delaying subexpression, as long as they are combined with

other subexpressions in a way that allows the whole delaying subexpression to be continuously distributed. If, in the process of regression, we encountered a constant, variable, or now expression, we know that the randomization over which subexpressions to regress to has led to this non-continuously distributed one. If this happens, we give up and try again. In some other randomization, the constant, variable or now expression will be sampled rather than regressed to, and we will be able to succeed.

The base case of the regression process is a continuous probabilistic choice `q(`e`)`. For such a choice, note that

$$\begin{aligned} p(s \rightsquigarrow s') &= p(\texttt{q(}e\texttt{)} \text{ resolves to } \delta') \\ &= \textstyle\sum_v p(e \text{ resolves to } v)(q(v)(\delta')) \end{aligned}$$

We can estimate the density by sampling a resolution for e. If it results in a nested delay, we can immediately return an estimate of 0, because s cannot lead to s'. If it results in a value v, we compute $q(v)(\delta')$. We can do this because q is a probabilistic primitive for which we can calculate densities.

Conditionals can be handled easily. To compute the density that `if` e_1 `then` e_2 `else` e_3 produces δ, we first sample e_1. If the result is `T`, we return the density that e_2 produces δ, otherwise we return the density that e_3 produces δ.

Now consider binary operators. Let e be $e_1 \oplus e_2$.

$$\begin{aligned} p(s \rightsquigarrow s') &= p(e_1 \oplus e_2 \text{ resolves to } \delta') \\ &= \textstyle\sum_{v_1} p(e_1 \text{ resolves to } v_1) \\ &\quad\; p(e_2 \text{ resolves to } \theta(v_1, \delta')) \end{aligned}$$

where $\theta(v_1, \delta')$ is the value such that $v_1 \oplus \theta(v_1, \delta') = \delta'$. (We assume that binary arithmetic operators are invertible.) We can estimate the density by sampling a resolution e_1 of v_1, and then recursively computing the density that e_2 resolves to $\theta(v_1, \delta')$. Now, this works fine for an expression such as `1.0 + uniform(2.0)`, but it does not for `uniform(2.0) + 1.0`, because after we have sampled `uniform(2.0)`, we will regress to `1.0`, which is not continuously distributed, and give up. So we randomly choose which operand to sample, and of which to compute the density. For an expression such as `uniform(2.0) + exp(1.0)`, we succeed either way. On the other hand, an expression such as `2.0 + 1.0` is impossible, because it is not continuously distributed.

To handle pairs and component extraction, the regression algorithm is given an additional argument, which is a *target*. The target is either $\star$, which means that we require the entire result of the expression, or a sequence of lefts and rights. For example the target `left.right` means that we need the right component of the left component of the value produced by the expression. When we want to compute $p($`left` e produces $\delta)$ with target τ, we recursively compute $p(e$ produces $\delta)$ with target `left.`τ.

For pairs, note that if a program is well-typed, the value of a delaying subexpression must always be a floating point number, so if such a subexpression involves a pair expression, we must only need the value of the target. This is not to say that the other component does not need to be evaluated. If the other component involves a nested delay, it will delay the whole pair. Therefore we proceed as follows: We sample the other component to see if it produces a nested delay. If it does, s cannot lead to s', so our estimate for the density is 0. Otherwise, we compute the density that the needed component produces δ'.

For first expressions, we note that

$$\begin{aligned} &p(\texttt{first } [e_1, e_2] \text{ resolves to } \delta') = \\ &\quad p(e_1 \text{ resolves to } \delta') + \\ &\quad p(e_1 \text{ results in a nested delay}) p(e_2 \text{ resolves to } \delta') \end{aligned}$$

We therefore recursively compute the density that e_1 produces δ'. We also sample e_1 to see if it results in a nested delay, and if it does we add the density that e_2 produces δ'.

Unfortunately, we run into trouble with some function applications. Our basic strategy for function applications is to sample the function and the actual argument, and then compute the density that the body, with the formal argument bound to the actual argument, resolves to δ'. This works fine in many cases, but not when the body of the function involves no continuous probabilistic choices. An example of an expression that causes problems is

```
(lambda x . if x > 3.0 then x else 2 * x)
(uniform(5.0))
```

Our algorithm is therefore incomplete. It works for a large and very useful class of programs, including all the examples of Section 3. We could have placed a restriction on the language to rule out function applications in delaying subexpressions. However, we think that would have been too draconian, ruling out some genuinely useful models on which our algorithm works fine.

We now argue that if the particles at time t_{i-1} are consistent with the emissions at time t_i, with positive probability our algorithm produces a sample with positive weight, as long as the model is one for which our density computation works. Let π be a path beginning with a particle at t_{i-1} that produces the emissions. Let t be the time point at the beginning of the temporal transition that terminated at t_i. Consider the delaying subexpression at t that resolved to $t_i - t$. We can reorder the resolution of delaying subexpressions such that this delay was the last to be resolved. Let $s = (e, t)$ be the state before this subexpression is resolved. There is an interval around all the continuous probabilistic choices in π before s is reached, such that if a sampler takes all the same discrete choices, and all the continuous choices in these intervals, and resolves delaying subexpressions in the same order, it will reach a state $s' = (e, t')$, where t' is in an interval around t, and the density that e will resolve to $t_i - t'$ is positive. Since we randomize the order in which delaying subexpressions are resolved, the sampler makes all these choices with positive probability. Since the density is positive, with positive probability all samples taken during the course of the regression will turn out the way they need to for the density estimate to be positive, so the importance weight will be positive.

5.3 Implementation

We have implemented our algorithm and tested it on a variety of examples, including some for which we can determine the correct probabilities analytically, to demonstrate its correct-

ness. For example, given the model

```
let f(b) =
 delay uniform (if b then 2.0 else 3.0);
 emit ``foo'';
 delay uniform 1.0;
 emit dist [ 0.8 :  b, 0.2 :  !b ];
 delay exp 1.0;
 b
in f(flip 0.4)
```

and the evidence $[1.0 : \text{“foo”}, 1.5 : \texttt{T}]$, the system generates $10,000$ particles in 2.5 seconds on a 2GHz single-core Windows machine. It predicts that the probability the output will be `T` is 0.8053; the correct probability is 0.8. We have also run it on the music example of Section 3. With 4 voices and 20 notes per voice, the system filters the observations using 100 particles in 34 seconds.

6 Discussion

We have presented a powerful, expressive language for representing probabilistic processes in continuous time. We have presented a semantics for the language, and an inference algorithm that works for a large and useful class of models.

A natural question to ask, however, is whether this language is needed at all. After all, probabilistic programming languages can define models over arbitrary data structures. Perhaps we could write a program in IBAL or Church that would define a probability distribution over trajectories and emission sequences, which would consist of values of variables and timestamps. We could then run the general-purpose inference algorithm of those languages on our program.

While such an approach might be possible some time in the future, there are several reasons why it is a good idea to develop a special-purpose continuous time language. One basic reason is that temporal models are so important that they deserve a language of their own. It would be much more cumbersome to develop them in a more general language from scratch. Ideally, one might hope to write a library in the more general language that would make developing temporal models straightforward, but that has not been done yet. In fact, CTPPL may serve as a guideline for designing such a library.

Another reason to develop a special-purpose temporal language is that time is special. The task of continually monitoring the state of a process in an online manner is different from that of reasoning about an entire model, therefore it requires different algorithms. In particular, we use particle filtering whereas Church uses MCMC.

A third, fundamental, reason is that IBAL and Church do not allow continuous variables, and it is non-trivial to extend them to do so. Note that CTPPL does not solve the problem of continuous variables in a general way. Floating point valued emissions are not allowed. The only place in which CTPPL fully handles continuous variables is in the time element. CTPPL is able to do this because of the special structure of time. Time always moves linearly forward, and delays are always positive. Our algorithm relies crucially on this structure. Nevertheless, we believe our regression algorithm to contain the seeds that will allow a language like Church to fully incorporate continuous variables. Thus, even though it develops a special-purpose language, the ideas in this paper may be useful in general.

Acknowledgements

We wish to thank Christian Shelton for fruitful discussions. Many thanks also to the anonymous reviewers for their useful suggestions. This work was supported by DARPA under the CALO program, through a subcontract from SRI International.

References

[Cemgil and Kappen, 2003] A. T. Cemgil and B. Kappen. Monte Carlo method for tempo tracking and rhythm quantization. *Journal of Artificial Intelligence Research*, 18:45–81, 2003.

[de Salvo Braz *et al.*, 2008] R. de Salvo Braz, N. Arora, E. Sudderth, and S. Russell. Open-universe state estimation with D-BLOG. Poster presented at NIPS 2008 workshop on probabilistic programming, 2008.

[Fan and Shelton, 2008] Y. Fan and C. R. Shelton. Sampling for approximate inference in continuous time bayesian networks. In *International Symposium on Artificial Intelligence and Mathematics*, 2008.

[Goodman *et al.*, 2008] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[Gopalratnam *et al.*, 2005] K. Gopalratnam, H. Kautz, and D. S. Weld. Extending continuous time Bayesian networks. In *National Conference on Artificial Intelligence (AAAI)*, 2005.

[Milch, 2006] B. Milch. *Probabilistic Models with Unknown Objects*. PhD thesis, Computer Science Division, University of California, Berkeley, 2006.

[Ng *et al.*, 2005] B. Ng, A. Pfeffer, and R. Dearden. Continuous time particle filtering. In *International Joint Conference on Artificial Intelligence*, 2005.

[Nodelman *et al.*, 2005] U. Nodelman, C. R. Shelton, and D. Koller. Expectation maximization and complex duration distributions for continuous time Bayesian networks. In *Uncertainty in Artificial Intelligence*, 2005.

[Nodelman, 2007] U. Nodelman. *Continuous Time Bayesian Networks*. PhD thesis, Stanford University, 2007.

[Pfeffer, 2005] A. Pfeffer. Functional specification of probabilistic process models. In *National Conference on Artificial Intelligence (AAAI)*, 2005.

[Pfeffer, 2007] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.