Composition of queries in probabilistic programming languages
Supervised by Prof. Frank Wood

Maximilian Igl

June 2016
# Contents

1. **Introduction** .................................................. 3  
   1.1. Inference in Prob. Programming .............................. 3  
   1.2. CPS and Thunks ............................................. 4  
   1.3. Anglican Inference Backend ................................ 4  
   1.4. Compositionality of queries ............................... 5  

2. **Composition of models** ........................................ 6  
   2.1. Code structuring without inference nesting ............... 6  
   2.2. Motivation .................................................. 6  
   2.3. Design Considerations ...................................... 7  
   2.4. Implementation ............................................. 9  
   2.5. Example ..................................................... 10  
   2.6. On the current implementation of conditional ............ 13  

3. **Composition of inference** ..................................... 15  
   3.1. Nested inference algorithms in the literature .......... 15  
   3.2. Nesting of inference algorithms in PPLs .................. 19  
      3.2.1. PMMH, SMC2 and Nested SMC ......................... 19  
      3.2.2. Implementation Strategy ............................. 21  

4. **Conclusion** ..................................................... 22  

A. **Backend code for no-op queries** .......................... 23  

B. **Code for examples** ........................................... 26
1. Introduction

This report is meant to be a starting point for the discussion about nesting queries and inference algorithms in probabilistic programming languages, focussing on Anglican. To do so, and to lend some structure to the large amount of possible implementations, it focuses on the implementation side, highlighting which possible solutions are feasible from this perspective and where there exist problems.

We first propose a baseline ‘no-op’ implementation of nesting where the nested query is reduced into the enclosing one. This enables the composition of queries without composition of inference, allowing more expressive and readable code. For example, it simplifies conditioning queries on other queries (i.e. models).

Second, we look at nested inference algorithms (opposed to just nested queries). We start by giving an overview over some of the proposed algorithms in the literature and argue that the goal of nesting queries in probabilistic programming languages should not be to reproduce those algorithms. This is mainly due to their complexity which is created by interactions between the nested and outer algorithms.

Lastly, we propose an extension to the implementation of the ‘no-op’ case which would allow us to nest inference algorithms without interaction. The main difference to the existing implementation of conditional is the fact that weighted particles are returned and that one could introduce additional arguments, for example for automatic burn in.

In the rest of this introductory section, we will give a very brief summary of some necessary background knowledge about Anglican and it’s inference algorithms.

1.1. Inference in Prob. Programming

Since for full turing complete prob. programming languages like Anglican, inference must be as general as possible, the current state of the art inference backend relies on Monte Carlo simulations, for example particle filters or Particle Markov Chain Monte Carlo.

The resulting probability distribution of the latent variables of interest is then approximated by a histogram over all (weighted) particles.

**Monte Carlo Methods** This serves as a very brief introduction to Monte Caro methods, in particular Sequential Monte Carlo (SMC).

The usual goal is to estimate a target density $\pi(x_{1:t})$ where $x_{1:t}$, $t \in [1\ldots T]$ is increasing in dimension over time. We can write

$$\pi(x_{1:t}) = \frac{\gamma_t(x_{1:t})}{Z_t}$$
where $\gamma_t$ is only known pointwise and $Z_t$ is unknown (as it often involves a very high dimensional non-analytically solvable integral). Our goal is to approximate this target distribution by particles $i \in [1 \ldots N]$:

$$\hat{\pi}_t(x_{1:t}) = \frac{1}{N} \sum_{i=1}^{N} \delta_{X_{1:t}^i}(x_{1:t})$$

The basic idea (Importance Sampling) is to draw from a proposal density $q_t(x_{1:t})$ s.t.

$q_t \geq \pi_t$ and to account for the 'difference' between $q_t$ and $\pi_t$ by a weight

$$w_t(x_{1:t}) = \frac{\gamma_t(x_{1:t})}{q_t(x_{1:t})}$$

This is normalized by the sum over all weights to give $W_i^t$. The expectation of any function under the target distribution is then approximated by a weighted sum of the function at the particle values.

One arising problem is that after few steps the variance of the weights will be large, i.e. most of the weight will be carried by very few particles. To resolve this problem, one introduces resampling steps were 'unimportant' particles are replaced by more meaningful ones. This reduces the variance but introduces a new problem: Particle degeneracy. After many time steps the current set of particles will be derived by only very few (or even only one) particle at time $t = 0$.

Possibilities to alleviate this problem lead to advanced methods like rejuvenation \cite{1} or look-ahead \cite{2}.

\subsection{1.2. CPS and Thunks}

Anglican’s inference backend relies on the probabilistic program being transformed into the \textit{Continuation Passing Style} (CPS). This is done by the \texttt{query} macro. In CPS the result of each computation is passed to a continuation function. Consequently the program flow can easily be resumed after interruptions by the backend to deal with inference-related tasks, in particular when \texttt{sample} or \texttt{observe} are called.

To prevent stack overflow, continuations can be wrapped in \textit{thunks}. Those are anonymous functions which don’t take arguments. Their only purpose is to call a function \textit{with} arguments. Those thunks can be repeatedly called by a \texttt{trampoline} (provided another thunk is returned), thereby preventing an increasing stack.

\subsection{1.3. Anglican Inference Backend}

One can imagine the structure of the anglican backend as having several layers:
1. **Inference Backend** *(infer function):* This is the layer called by *doquery*. Particle interactions (e.g. resampling) have to be performed in this layer as all other layers are per particle. This and the next layer are the only parts which are specific to each inference algorithm.

2. **Inference Backend** *(checkpoint functions):* Checkpoints are called when values are sampled or observed, as well as when the program finishes.

3. **Execution function** *(exec):* (Re)-starts the program execution through calls to the trampoline. Relays sample and observe interruptions to the inference backend.

4. **trampoline**: Executes the CPS-transformed anglican program by repeatedly calling returned thunks. sample and observe interruptions are returned as records who’s type is used to relay them to the correct checkpoint in the inference backend. The same is done when the anglican program has ended, returning a result type.

5. Anglican program: CPS transformed version of the program

Note that layers after the exec layer don’t have any information about which inference algorithm is being run.

Apart from those layers, there is also the state, a hash-map which is passed along with each function call. The state contains the current weight of the particle, the predicted values, the possibility to store values (for XRPs), memoized function calls and potential additional fields for algorithm specific requirements.

### 1.4. Compositionality of queries

Queries in Anglican are stochastic programs, which support soft constraints and describe the generative model of the joint probability distribution over a subset of it’s variables. They can take input arguments, making them to definitions of conditional distributions.

In both cases, whether we see them as generative models or as computer programs, it make sense to design them out of modular parts. For generative models, this allows us to specify different inference algorithms for certain sub-parts and convey additional information about the structure of the model, for example its conditional independence characteristics. For computer programs, it allows us to better structure our code as well as to combine and iterate through different models faster. It also makes it easier to write more complex models who’s parts can be exchanged at runtime.

However, composing models in a nested manner, possibly even with different inference algorithms, requires us to examine the resulting composition from different viewpoints: Mathematical correctness, language semantics and implementation possibilities, particularly in Anglican. The last point will be the main interest of this report.

This report is only a starting point for the necessary considerations and still very much a work in progress. First, in section 2, we think about a possible implementation of
nesting queries within Anglican in the case that there is only one inference algorithm run over both outer and nested queries. We call this the ’pass through’ or ’no-op’ case.

In section 3, we’ll first review some inference algorithms in the literature which compose different MC algorithms in order to achieve better performance in complex models. This is followed up with a short discussion why implementing those interacting algorithms is currently difficult in Anglican. Lastly, we will discuss a possible backend implementation for nesting inference algorithms in a non-interacting manner.

2. Composition of models

2.1. Code structuring without inference nesting

In this section we discuss how a pass through (or no-op) nesting of queries could be implemented in Anglican.

2.2. Motivation

The structuring of code in Anglican can already be done to some extend using the defm macro which allows the definition of functions which can use the Anglican language, in particular predict, observe and sample.

However, functions defined by defm have several characteristics which make them not ideal as representation of sub-queries.

'Stochastic' referential transparency Since they can be defined inside the calling query, they do not guarantee 'stochastic' referential transparency. Inside their function body, they can reference variables in the scope of the outer query. If we see queries as representations of (conditional) distributions, they should only depend on their input arguments. Unfortunately, this is also not the case for queries. We can define, condition on and sample from a query all inside another query (see listing 1).

Semantics If we use functions (defined using defm) for nested code blocks, we cannot easily switch between different nested inference algorithms (using (conditional)) and no-op nesting. While we need to sample from a conditioned query to get a hash-map of results, functions will return some value directly.

In summary The idea of allowing no-op nesting of queries is that it

1. allows for easily exchanging subqueries and distributions
Listing 1: Example that 'stochastic' referential transparency is not guaranteed for queries in Anglican. It also shows that we can define and condition on queries inside other queries.

2. allows for easily changing between nested inference and no-op reductions. This should be particularly helpful for performance comparisons

3. allows for easily composing existing queries into larger models, possibly even at runtime

2.3. Design Considerations

The goal is to take a query object, i.e. an Anglican program, and nest it into an outer query in a way that is 'invisible' to the inference algorithm ('no-op' or 'pass through').

Following are some desired properties:

- To easily switch between the no-op nesting and a nested inference algorithm, the syntax should be `(sample ((conditional subquery :no-op) args))`. This way, it could for example easily be changed into `(sample ((conditional subquery :smc) args))`.

- The object resulting from `(conditional subquery :no-op)` should behave like a distribution generator, i.e. like for example `normal` or `discrete`. It also should be callable inside and outside of a query object (see discussion below). Note that actually calling `sample` on that object outside of a query does not make sense (as no inference algorithm is specified) and doesn’t need to be implemented.

- The nesting should be independent from the inference algorithm, i.e. no change of code should be needed inside the checkpoints.

- `observe` statements inside the subquery should influence the weight of the (outer) trace. In other words: Sampling from such a subquery should result in weighted particles.

---

1The discussion about the ideal scope of the Anglican language is not touched upon in this report.
• The return value of sample should be a map which contains the values which were predicted inside the nested query. Those values should not be automatically added to the predicted values of the outer query.

• Nesting should be possible to arbitrary depth.

Note that if we want to make queries decomposable and easily interchangeable, it should be possible to pass distribution generators as arguments to queries. This is interesting, as we can then condition on models, not only on values or distributions.

In the current implementation of conditional that doesn’t work (if we use it outside of a query), see listing 2 which does not work.

(def subq (query [] (predict :b (sample (normal 0 1)))))
(def subq-cond (conditional subq :smc))
(defquery outerq [subq-cond]
  (predict :b (:b (sample (subq-cond)))))
(take 2 (doquery :smc outerq [subq-cond]))

; => ClassCastException anglican.emit$conditional$fn$reify__2028 cannot be cast to clojure.lang.Associative; clojure.lang.RT.assoc (RT.java:778)
; => $

Listing 2: Trying to pass a distribution generator created by the ‘old’ conditional results in an error message. The no-op implementation of conditional handles this situation correctly, albeit using a workaround (see code in appendix A).

The new (no-op) implementation can handle this situation, albeit not very elegantly (see code in appendix A). The difficulty is that the generator object returned by conditional needs to be wrapped as primitive procedure, which usually only happens if we pass it to with-primitive-procedures or if we create it inside a query. Otherwise, we have no possibility to recognize at compile time that a symbol represents a generator. The workaround for the no-op implementation is that the generator object itself counts the number of arguments passed to it. If it receives two more than expected (presumably the state and continuation), it assumes to be in a CPS environment and behaves accordingly.

This is not ideal, as CPS transforming itself should not be the responsibility of the generator object and it can also lead to misleading error messages. As far as I understand the problem, however, this can only be resolved by expanding the scope of the Anglican language, a discussion which is outside the scope of the report.
2.4. Implementation

There are several issues which restrict the way the nesting of queries can be implemented.
First of all, we cannot just inline the source code of the nested query, since this would
not preserve the scope in which the query was originally defined. Consequently, we must
use the already CPS transformed function object which is returned by the query macro.
This also means that we cannot replace the calls to predict inside the nested query.
Furthermore, we also cannot change the result type the nested query returns after it
finishes which would stop the outer inference algorithm if passed through to it. This
makes it necessary to intercept the return values from the CPS functions.
Second, we must change the source code of the existing anglican library: Although we
could introduce an additional layer above the current trampoline layer (see section 1.3)
to execute the nested query and filter predict and result return values, this would
not be viable. The code inside the nested query would be needed to be executed when
sample is called on it. However, a sample statement in a query is actually converted
to the constructor ->sample during CPS transformation. The actual call to the sample
method is done in the checkpoint in the inference backend. However, from there, we
cannot run the query. We need to run it above the exec layer because we have no
information about which inference algorithm to call so we rely on the exec function to
relay it to the correct one.
Lastly note that we cannot recognize distribution constructors during CPS transforma-
tion as it is just a compiled function call. Since we are only rearranging symbols during
the CPS transformation, we also don’t have access to the metadata (which in any case
would only be known during runtime). This then also cannot be used to recognize them.
The consequence of this is the mentioned un-elegant solution to make a distribution
constructor callable within and outside of a query (namely counting its arguments).
As we must implement the nesting in the existing anglican code but not in the check-
points, we can only implement it in the exec function. The implementation idea is based
on a stack which is created in the state variable:

- **conditional** creates a constructor function which can be called like e.g. normal
  or discrete.
- Upon being called, it returns not a distribution, but a subquery type.
- In exec, when sample is called, it is checked whether the passed distribution is a
  subquery type (instead of being a distribution). If so, the call is not relayed to
  the inference checkpoint but handled in exec itself.
- This is done by pushing the current environment (predicts, store and mem), and
  also the current continuation, to a stack inside the state variable and emptying

---

2Innermost function is at the very top of the stack
their fields. The new continuation is the entrance point to the subquery.

- Inference runs as normal until a result type is returned. If so, exec checks whether there are environments on the stack (i.e. if we are returning from a subquery). If so, the stack is popped and the old environment restored. The predicted values before restoration are passed to the continuation which is also taken from the stack.

The complete code necessary for the implementation is given in appendix A.

2.5. Example

As an example let’s look at a SSM with a static parameter:

\[
\begin{align*}
\theta &\sim \mathcal{N}(\theta, 0, 10) \\
d &\sim \mathcal{N}(d, 0, 10) \\
x_0 &\sim \mathcal{N}(x_0, 0, 1) \\
x_t &\sim \mathcal{N}(x_t, x_{t-1} + d, \theta) \\
y_t &\sim \mathcal{N}(y_t, x_t, 1)
\end{align*}
\]

with data

\[
y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
\]

The anglican code is the following, adapted from Frank Wood’s nested HMM example.

```lisp
(defdist mu0 [] []
  (sample [this] (sample (normal 0 1)))
  (observe [this value] (observe (normal 0 1) value)))

(defdist f [x-prev theta] []
  (sample [this] (sample (normal x-prev theta)))
  (observe [this value] (observe (normal x-prev theta) value)))

(defdist g [x-cur] []
  (sample [this] (sample (normal x-cur 1)))
  (observe [this value] (observe (normal x-cur 1) value)))

(def ys '(0 1 2 3 4 5 6 7 8 9 10))

(with-primitive-procedures [f g]
  (defquery one-step-query [x-t-minus-1 y theta d]
    \[\begin{align*}
\theta &\sim \mathcal{N}(\theta, 0, 10) \\
d &\sim \mathcal{N}(d, 0, 10) \\
x_0 &\sim \mathcal{N}(x_0, 0, 1) \\
x_t &\sim \mathcal{N}(x_t, x_{t-1} + d, \theta) \\
y_t &\sim \mathcal{N}(y_t, x_t, 1)
\end{align*}\]

\footnote{There is more consideration to be done with regards to the question how the store and mem fields in the state variable should be handled. Also, this should be expanded to include inference algorithm specific fields.}

10
\[
\text{(let \[x-t (sample (f (+ x-t-minus-1 d) theta))\])}
\text{(observe (g x-t) y)}
\text{(predict :x x-t)))}
\]

\[
\text{(def one-step-posterior-nop (conditional one-step-query :no-op []))}
\]

\[
\text{(with-primitive-procedures [mu0 g]}
\text{(defquery outerq []}
\text{(let \[theta (abs (sample (normal 0 10)))}
\text{d (sample (normal 0 10))}
\text{x0 (sample (mu0))]
\text{(predict :theta theta)}
\text{(predict :d d)}
\text{(observe (g x0) (first ys))}
\text{(loop [ys (rest ys)}
\text{xs [x0]}
\text{(if (empty? ys)}
\text{(predict :xs xs)}
\text{(let \[x
\text{(:x (sample}
\text{(one-step-posterior-nop}
\text{(last xs) (first ys) theta d))))\])}
\text{(recur}
\text{(rest ys)}
\text{(conj xs x))))})})
\]

To check the validity, I compared against the results from a query which was not written as nested (i.e. 'flat'). I also compared the results against a nested query which was evaluated using the 'old' implementation of conditional, using a 10 particle PIMH inference as nested inference. See appendix B for the code of both queries.

The parameters used are given in table 1, the results are given in table 2

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of samples for reference query</td>
<td>200000</td>
</tr>
<tr>
<td>Number of samples for tested queries</td>
<td>20000</td>
</tr>
<tr>
<td>Number of samples for nested inference (old conditional)</td>
<td>10</td>
</tr>
<tr>
<td>Nested inference algorithm for old conditional</td>
<td>PIMH</td>
</tr>
<tr>
<td>Number of particles</td>
<td>100</td>
</tr>
<tr>
<td>Burn-in</td>
<td>First 50%</td>
</tr>
</tbody>
</table>

Table 1: Parameters used for the example runs. For burn in, the first 50% of all samples was discarded

The results of the accuracy and performance comparison are given in table 3. The 'flat' program was run twice, once with 10 times as many samples as a reference basis. 'Flat'
Table 2: Typical results of the reference run for particle gibbs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Inference</th>
<th>δd</th>
<th>δθ</th>
<th>δ⃗ x</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>flat</td>
<td>:pgibbs</td>
<td>0.09</td>
<td>0.06</td>
<td>0.86</td>
<td>6 sec</td>
</tr>
<tr>
<td>flat</td>
<td>:pimh</td>
<td>0.11</td>
<td>0.05</td>
<td>0.80</td>
<td>6 sec</td>
</tr>
<tr>
<td>flat</td>
<td>:lmh</td>
<td>0.08</td>
<td>0.22</td>
<td>0.76</td>
<td>4 sec</td>
</tr>
<tr>
<td>flat</td>
<td>:smc</td>
<td>0.05</td>
<td>0.27</td>
<td>0.25</td>
<td>6 sec</td>
</tr>
<tr>
<td>new</td>
<td>:pgibbs</td>
<td>0.02</td>
<td>0.09</td>
<td>0.11</td>
<td>8 sec</td>
</tr>
<tr>
<td>new</td>
<td>:pimh</td>
<td>0.05</td>
<td>0.01</td>
<td>0.44</td>
<td>8 sec</td>
</tr>
<tr>
<td>new</td>
<td>:lmh</td>
<td>0.07</td>
<td>0.46</td>
<td>0.44</td>
<td>8 sec</td>
</tr>
<tr>
<td>new</td>
<td>:smc</td>
<td>0.06</td>
<td>0.18</td>
<td>0.21</td>
<td>8 sec</td>
</tr>
<tr>
<td>old</td>
<td>:pgibbs</td>
<td>0.82</td>
<td>7.61</td>
<td>7.23</td>
<td>118 sec</td>
</tr>
<tr>
<td>old</td>
<td>:pimh</td>
<td>0.84</td>
<td>7.49</td>
<td>7.12</td>
<td>120 sec</td>
</tr>
<tr>
<td>old</td>
<td>:lmh</td>
<td>0.91</td>
<td>7.45</td>
<td>8.57</td>
<td>94 sec</td>
</tr>
<tr>
<td>old</td>
<td>:smc</td>
<td>1.03</td>
<td>7.58</td>
<td>7.56</td>
<td>118 sec</td>
</tr>
</tbody>
</table>

Table 3: Comparison between different nesting possibilities. The program column refers to whether the query was written in a non-nested way (‘flat’), whether a nested inference algorithm was used (‘old’ conditional) or whether no-op nesting was used (‘new’ conditional). The flat query was actually run twice, the first time being for the reference result with 10 times as many particles. Standard deviations are not included due to time constraints. For an explanation of the ‘old’ results, see section 2.6.

and ‘new’ should give the same accuracy (which they roughly seem to be doing), as the inference of both should be the same due to the no-op reduction. There is some time costs associated with the no-op reduction, namely about an additional $1/3$ of the runtime. However, due to the shortness of the actual query that should be close to the worst case behavior. The time cost, although not profiled in detail, is most likely stemming from the additional checks in exec and in the generator function.

The accuracy of the ‘old’ conditional is discussed in the next section. The much slower performance is due to the ‘explosion’ of particles: For each particle in the outer query, the nested query runs (in this case) 10 particles, even if only one sample is needed.
2.6. On the current implementation of conditional

As can be seen in table 2, the query using nested inference does not give the correct answer. This is to be expected, as in this case sample gives the first sample of a Markov Chain (we ran PIMH as nested algorithm). This would require some burn in, which we neglected. The problem would be similar in the case of e.g. SMC as conditional internally runs some kind of MH algorithm to un-weight the samples returned by SMC.

Including burn in into the comparison is left for future work. Furthermore, I would expect that ‘plainly’ nesting inference algorithms can only give improved performance in some cases, e.g. when some parts of the model are much better suited for certain algorithms than others or when achieving a good accuracy is much harder in one part of the model than in another. This argument especially holds, when sub-parts of the model can be solved by exact inference through e.g. expectation propagation.

There are also some peculiarities with the ‘old’ implementation of conditional with respect to it’s semantics:

- The necessity to include the return value (the distribution generator) in the call to with-primitive-procedures makes it impossible to pass it as an argument to a query, limiting the flexibility of the language
- conditional can be used in two very different ways, which exhibits a lot of flexibility, but is somewhat unexpected/surprising:
  1. We can use conditional like in the example above to sample one value in a query. This leads to the expected ‘explosion’ in particle numbers.
  2. We can also use the conditional distribution outside of a query (see listing 3), provided the conditioned query is written accordingly. This does not lead to higher particle numbers.

If we use conditional (or rather the returned query generator) outside of a query (see listing 3), we will get better results. However, we won’t have the expected ‘explosion’ of particles as all particles in the next step share the same distribution from the previous step. The resulting overall inference is more complicated: Although we still have no burn-in, the problem is reduced by the fact that we will draw multiple samples from the same distribution, so it is enough to have burn in for the outermost query as it will ‘propagate through’ to the others (although it is unclear how fast so). There is also a one particle burn in for each nested query as equalize consumes two samples to produce the first one (see output in listing 3).

What’s interesting is that using conditional outside and inside of a query lead to different nesting structures. The innermost query in the one case is the outermost in the other.
(defdist mu0 [] []
  (sample [this] {:x [1]})
  (observe [this value] (observe (normal 0 1) value)))

(defdist f [x-prev] []
  (sample [this] (sample (normal x-prev 1)))
  (observe [this value] (observe (normal x-prev 1) value)))

(defdist g [x-cur] []
  (sample [this] (sample (normal x-cur 1)))
  (observe [this value] (observe (normal x-cur 1) value)))

(def ys '(0 1 2 3))
(with-primitive-procedures [f g]
  (defquery one-step-query [x-t-minus-1-d y]
    (let [x-all (:x (sample x-t-minus-1-d))
          x-t-minus-1 (last x-all)
          x-t (sample (f x-t-minus-1))
      (observe (g x-t) y)
      (print y)
      (predict :x (conj x-all x-t)))))

(def one-step-posterior
  (conditional one-step-query
    :smc
    :number-of-particles 2))

(def posterior
  (reduce one-step-posterior (mu0) ys))
;; => 00000110011222223300111223330011223

(sample posterior)
;; => 00112233
;; => {:x [1 -0.5158104563909478 0.8543609866054718 2.3324771427460487 1.86093969323759579759577]}

Listing 3: Alternative possibility of how conditional can be used. Adapted from original code by Frank Wood. Note that there is no 'explosion' of particle numbers as each (same) distribution is used by all particles in the consecutive distribution. To see that, notice the print statement in one-step-query and the output after the reduction step. There is also a one-particle burn in step in equalize (which explains the samples during the reduction step.)
3. Composition of inference

In the previous section we implemented a no-op nesting of queries in order to be able to structure our code and make Anglican more flexible. However, it can be beneficial to actually nest inference algorithms. This section discusses how this might be possible in Anglican. To start, it gives an overview over some of the literature of nested queries.

3.1. Nested inference algorithms in the literature

Iterated Batch Importance Sampling \cite{3} Iterated Batch Importance Sampling (IBIS) targets the situation of a static distribution $\pi(\theta|y_1, \ldots, y_N)$ depending on several data points $\{y_1, \ldots, y_N\}$. The algorithm is useful primarily in situations where observations are independent or Markov.

The intuitive idea is to use MCMC to estimate $\theta$. However, instead of using all $N$ data points between each MCMC move, only a certain number $p$ additional data points $y_i$ are evaluated. Those lead to an importance weight $w_{n,p}(\theta_j)$ where $j = 1 \ldots N$ and $n$ is the total number of data points processed previously. This requires that the incremental likelihood $p(y_t|y_{t-1}, \theta)$ can be computed.

Based on those $w_{n,p}(\theta_j)$, a resample step is performed, followed by the next MCMC move (rejuvenation \cite{1}) and further processing of $p$ additional data points until all $N$ are incorporated into the target distribution.

Particle Markov chain Monte Carlo \cite{4} The idea of PMCMC is to use SMC to build a high dimensional proposal distribution for MCMC. This is especially helpful for e.g. SSMs which have a complex dependence structure. PMCMC is already implemented in anglican in several different flavours. In the following we will discuss Particle marginal Metropolis-Hastings (PMMH) and Particle Gibbs (PG). Particle independent Metropolis Hastings is not discussed as it is less relevant for the question of nested queries.

The flavours of PMCMC are differentiated in terms of the underlying MCMC algorithm used. If we assume a SSM with static parameter $\theta$ and latent variables $(x_t)$, PMMH would sample $\theta$ from a kernel $q(\theta'|\theta)$ whereas PG would sample from $p(\theta|y_{1:T}, x_{1:T})$.

For PMMH that means that the acceptance probability is influenced by the likelihood $p_\theta(y_{1:T})$ which is estimated by an SMC sweep. For PG the situation is slightly more complicated as we need to update each $x_t$ conditional on all other variables. As shown by \cite{4}, this can be accomplished by an conditional SMC update: For each SMC sweep after the first one, the retained particle from the previous sweep is inserted into the particle set and kept fixed (this requirement can be slightly loosend by Particle Gibbs with Ancestor Sampling \cite{5}). In both cases only one particle $x_{1:T}$ is retrained from each sweep (although this can be expanded).
Whereas the example of an SSM with static parameters is useful for the question of nested queries, Particle Gibbs is interesting for probabilistic programming languages in the much more general case, as it allows changes to more than one variable at a time [6].

**SMC^2 [7]** SMC^2 targets a similar problem as PMCMC, namely a State Space Model with static parameters. To do so, it extends IBIS, which is not applicable in this case: To compute the weights, the likelihood \( p(y_t|y_{1:t-1}, \theta) \) (or \( p(y_{t:t+p}|y_{1:t-1}, \theta) \)), is needed but not computable. This is due to the latent variables \( (x_t) \) which need to be integrated out.

SMC^2 extends IBIS by running a nested SMC algorithm for each particle \( m = 1 \ldots N^\theta \) representing one \( \theta^m \). This nested SMC algorithm is used to estimate the needed likelihood \( p(y_t|y_{1:t-1}, \theta^m) \). When a degeneracy criterion (usually based on ESS) if fulfilled for the outer SMC, a new set of particles for \( \theta^m \) are sampled based on a MH kernel. For each new \( \tilde{\theta} \), a new nested PF must be run until \( t \) to produce a new estimate for \( \tilde{Z}_t \).

**Auxiliary Particle Filter [8]** This is not really a nested inference algorithm. It is discussed as it is the basis for Nested Sequential Monte Carlo.

Auxiliary Particle Filtering (APF) is a form of 'look ahead'. Assume the case of a markovian state space model with latent variables \( x_t \sim f(\cdot|x_{t-1}) \) and observed variables \( y_t \sim g(\cdot|x_t) \). The idea of APF is to already incorporate information about \( y_{t+1} \) into the resampling step at time \( t \). This can help to prevent trace degeneracy, if the filtering distribution \( p(x_t|y_{1:t}) \) is 'very' different from \( p(x_t|y_{1:t+1}) \). It does so by *not* throwing away samples (in the resampling step) which have lower weight under \( p(x_t|y_{1:t}) \), but higher weight under \( p(x_t|y_{1:t+1}) \).

To do so, we sample the ancestor \( k \) together with the new \( x_{t+1} \) from a joint density:

\[
f(x_{t+1}, k|y_{1:t+1}) \sim g(y_{t+1}|x_{t+1})f(x_{t+1}|x_t)^k \pi_t^k
\]

and then discard \( k \) to produce a sample from the *empirical filtering density*. \( \pi^k \) is the probability mass of particle \( k \).

We are free to choose the proposal \( q(x_{t+1}, k|y_{t+1}) \), we only need to correct for it when computing the weights. The trick is to choose \( q \) in a way that we can *first* sample \( k \) based on \( k \sim f(y_{t+1}|\mu_k^t)\pi_t^k \) where \( \mu_k^t \) is some 'likely' value associated with particle \( k \), e.g. the mode at \( t + 1 \) or a draw. This replaces the resampling step at time \( t \) and allows us to base the samples at time \( t + 1 \) on particles which were carrying a high weight at time \( t \) when incorporating information \( y_{t+1} \).

**Nested Sequential Monte Carlo [9]** Similarly to the APF, Nested Sequential Monte Carlo (NSMC) mimics the *fully adapted* proposal distribution. NSMC is especially useful in cases where \( x_t \) are high dimensional and have an inner independence structure as this
would lead to fast sample impoverishment for a regular SMC. Observations $y_{t+1}$ are taken into account at time $t$ by resampling the outer SMC according to the marginal likelihood $Z_{q_{t+1}} = p(y_{t+1}|x_t)$ which is estimated by the inner SMC. Consequently, there is one inner SMC for each particle in the outer PF. The weights are random, but samples are properly weighted.

**Comparison table** The above mentioned algorithms are summarized in table [I].
<table>
<thead>
<tr>
<th>Inference Algorithm</th>
<th>Outer Alg</th>
<th>Inner Alg</th>
<th>Passed in</th>
<th>Passed out</th>
<th>Implementation as nested queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBIS</td>
<td>As there is no static parameter, there can’t be nested queries.</td>
<td>SMC with intertwined MH rejuvenation steps.</td>
<td>No, as there are not static parameters. Recursive nesting is not considered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMCMC (PMMH)</td>
<td>MH</td>
<td>SMC</td>
<td>Static params</td>
<td>Evidence $p_\theta(y_{1:T})$; one sampled trace</td>
<td>Possible if nested SMC can influence acceptance ratio</td>
</tr>
<tr>
<td>PMCMC (PGibbs)</td>
<td>Gibbs (MCMC)</td>
<td>Conditional SMC</td>
<td>Static params, last retained trace</td>
<td>Evidence $p_\theta(y_{1:T})$; one sampled trace</td>
<td>Already implemented as inference algorithm. Could be implemented as nested queries.</td>
</tr>
<tr>
<td>SMC2</td>
<td>IBIS</td>
<td>SMC</td>
<td>Like IBIS: Intertwined MH with two SMC particle filters. Implementation in Anglican as nested queries is difficult, since after each sampling step of the inner query the outer weights must be updated as well, possibly triggering MH resampling.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auxiliary Particle Filter (APF)</td>
<td>Not a nested inference algorithm. Basis for Nested SMC.</td>
<td></td>
<td>Best implemented as general inference algorithm. Resampling step needs to be modified, the program would need to be run till the next observe statement before resampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested SMC</td>
<td>SMC (APF)</td>
<td>SMC</td>
<td>$x_t$ if needed</td>
<td>Marginal likelihood $Z_{q+1}$, one sampled trace</td>
<td>Difficult since interaction is required (see discussion below).</td>
</tr>
</tbody>
</table>

Table 4: Selection of nested inference algorithms in the literature. Note that both IBIS and APF are not nested inference algorithms but included as they form the basis for SMC2 and NSMC respectively.
3.2. Nesting of inference algorithms in PPLs

In the following we will discuss possibilities of how nesting of inference algorithms can be done in Anglican. First we will discuss the interacting algorithms presented in table 4. Afterwards, we will discuss the non-interacting version if we just want to run different inference algorithms over different parts of the program, for example if we can use exact inference on a subquery.

3.2.1. PMMH, SMC2 and Nested SMC

I will argue in the following why I don’t think it makes sense to develop a general way in which we can easily plug inference algorithms together to build algorithms like PMMH, SMC2 or NSMC. This is based on four arguments:

1. There is only a very limited amount of such algorithms in the literature and it might be less work to just implement those as overall algorithms. The question then becomes how to differentiate semantically between nested queries of different types: Non-interacting (with independent nested inference algorithms) and interacting (which are all evaluated by one inference algorithm).

2. The nested algorithms in those cases interact with each other in non-trivial ways and it is not clear that we can do so for arbitrary combinations in a mathematical correct way.

3. As their interactions are non-trivial, this would require a lot of configuration on the user’s part (and possibly leaves a lot of room to ‘shoot oneself in the foot’).

4. As argued further below, implementing interaction between nested algorithms would require a big workaround in the current implementation of Anglican if we want to nest them using some kind of keyword like conditional.

To make those four point concrete, I will discuss what would need to be done in order to implement PMMH, SMC2 and NSMC.

SMC2 There are several important points for SMC2:

- There is an outer query with $N^O$ particles and an inner query with $N^I$ particles for each outer particle. So $N^O \cdot N^I$ total particles.

- observe statements in the outer query affect the weight of the outer query directly. observe statements of the inner query affect the inner particle directly and the outer query through the average inner particle weight (i.e. their marginal likelihood).

\[\text{Disclaimer: As I did not have time to actually implement those algorithms, it is most likely I overlooked key parts or problems in their implementation.}\]
• After each `observe` statement, all weights (outer and inner) are updated and it is checked whether some degeneracy criterion (e.g. ESS) is fulfilled for the outer query. If yes, there needs to be some mechanism for the creation of new particles. In the original paper this is done by an MH step with a kernel centered around an existing (resampled) particle (each outer particle only defines one parameter $\theta$ which makes this feasible). It should be possible to replace the MH step by an LMH step. However, it is an open question whether the resulting algorithm still gives performance gains (see discussion below).

• After a new particle is created, it needs to be accepted or rejected. This is done by running the inner SMC until the current time $t$. To do so we need to keep track of how many `observe` statements we already encountered in the existing particles.

The complexity of creating a new particle is linear in $t$. For $t \gg 1$, SMC2 relies on the fact that most outer parameters $\theta$ should already be fairly accurate, so MH updates need to happen less often. This assumes that the MH kernel gives good proposals (so that re-running the inner SMC is less likely to be just thrown away). If we use LMH, this second characteristic is no longer given and it is therefore not clear that SMC2 can be successfully used for Anglican.

**PMMH** PMMH is quite similar to SMC2. While an inner SMC algorithm is used to estimate the marginal likelihood $p_\theta(x_{1:T}|y_{1:T})$, updates to the outer query (which samples $\theta$) are done by Metropolis-Hastings with acceptance ratio

$$
\frac{p_{\theta'}(y_{1:T})p(\theta | \theta')q(\theta | \theta')}{p_{\theta}(y_{1:T})p(\theta')q(\theta' | \theta)}
$$

The difference to SMC2 is that there is no outer particle ensemble with possible re-sampling steps and the inner query is always run to full length (incorporating all $y_{1:T}$) instead of using the IBIS style successive incorporation of new data.

Consequently, the discussion of the possibility of it’s implementation in Anglican is analogue.

**Nested SMC** Implementing Nested SMC in Anglican should be possible in principle, however, doing so using conditional does not work and would require either a major redesign of the inference backend or some very un-elegant workarounds. The reason holds for all nested inference algorithms which are interacting and lies in the structure of the current Anglican (see section 1.3).

If there is any interaction between particles in the outer algorithm, this needs to be handled in the `infer` layer as the following layer on the stack (`exec`) is per particle.

\footnotetext{5}{(although some more thought should be put into this)}
Additionally, as discussed above, we need to 'hook in' the nested algorithm in the exec layer. It follows, that this layer would need to re-route the control either to the outer or inner infer, depending on the situation. This would certainly be possible, but would require a lot of checks, impacting the performance noticeably.

While it is relatively easy to see the interaction in the case of SMC2, it is slightly more subtle for NSMC. Here, it comes from the fact that the nested query needs to call observe(\(Z_{t,t+1}\)) of the outer one, then hand control back to it to trigger a resampling step based on the new observation and only then sample one trace as return value of the nesting sample statement. If we do not hand back control to the outer algorithm in between, all traces with the same ancestor will have the same new value \(x_{t+1}\).

### 3.2.2. Implementation Strategy

Having given an argument why implementing interacting nested inference algorithms in a modular way is hard in the current implementation of Anglican, I would like to outline a brief sketch of how non-interacting nesting can be implemented in a cohesive way. I will focus on nesting on particle level (i.e. conditioning a query inside another query) and neglect nesting on a distribution level (see listing 3).

This is a slight extension to the solution presented previously for the no-op nesting. Also note that this is not actually implemented yet and as the following is only a sketch it will likely need some more adoption.

The idea is to keep the structure of exec from section 2.4 but create two additional defmultis, init and exit, which will handle the stack (this time also including the log-weights) and call the nested inference algorithm. On exit, observe of the outer inference algorithm will be called to register the log-weight of the nested query (i.e. its marginal likelihood) and also (possibly) trigger resampling or other necessary computations. For all current implementations of observe, this shouldn’t cause any unwanted side effects. Alternatively, one could have called add-log-weight directly. However, this would lead to unwanted consequences. Take, as example, the program from section 2.5. Here, as all observe statements are in the nested query, running SMC on the outer query and calling add-log-weight instead of observe would never trigger a resampling step in the outer query.

If we assume that the nested inference algorithm is specified when calling conditional (like it is now), that means we have to extend the record subquery by two additional fields: One for the nested inference algorithm and one for the outer algorithm. The latter information is passed by exec and necessary to know which observe checkpoint to call.

Having this setup would also allow the specification of new options for the nested inference algorithm, like e.g. :burn-in in conditional. This would make it possible to arbitrarily interchange nested inference algorithms without having to take care of e.g.
the necessity for burn in inside the actual query.

The record subquery could also implement the distribution interface to allow sampling outside of queries.

4. Conclusion

This report is meant to be a starting point for the discussion about nesting queries and inference algorithms in probabilistic programming languages, focussing on Anglican. To do so, and to lend some structure to the large amount of possible implementations, it focused on the implementation side, highlighting which possible solutions are feasible from this perspective and where there are problems.

As a second step, it will be important to make sure the here identified, possible solutions are also viable from a mathematical point of view, as nesting queries leads to random weights which will be used both in weight-based algorithms (like IS) and rejection based algorithms (like LMH).

Furthermore, this report focused on finding solutions which leave the basic backend structure of Anglican in place while adding on new functionality. There are several reasons why one might want to think about more profound changes:

• The here proposed implementation does not transform conditioned queries into distribution object, but rather into subquery objects. This makes performance impacting checks necessary and creates additional complexity in the back end code. However, one could argue this to be justified by the fact that distribution objects should return unweighted samples whereas subquery objects need not to.

• Apart from nesting inference algorithms using conditional, we would like to have a way of telling the inference backend that a query is considered nested, but without it having it’s own algorithm. This would open up the possibility to implement interacting algorithms like SMC\textsuperscript{2} or Nested SMC.

• The here proposed solution uses a workaround to make it possible to pass sub-queries as arguments to other queries. Expanding the scope of the Anglican language to include code outside of query or defm definitions could eliminate the need for this.

• The runtime of the here proposed solution could also be slightly improved by moving the tests performed in exec into the checkpoints, possibly having a two stage sample and result checkpoint: The first stage performs the tests and would then call the actual checkpoint of the inference algorithm.

\footnote{Note the discussion in the previous section of whether this make sense in a probabilistic programming context as it requires a well chosen kernel to be efficient}
## A. Backend code for no-op queries

Code in anglican.trap:

```clojure
(defrecord subquery [query value])
```

Code from conditional in anglican.emit:

```clojure
(defn conditional
  "accepts an Anglican query and returns
  a conditional distribution defined by the query"
  [query & options]

;; Algorithm parameters are an optional sequence of the form
;; [inference-algorithm & algorithm-options]
;; with importance sampling as the default inference algorithm.
(let [[algorithm & options] options
      algorithm (or algorithm :importance)]

;; Conditional distribution is a function which, when applied
;; to the values (argument of query), returns a distribution
;; object.

(if (= algorithm :no-op)

;; Wrap the subquery into the subquery type.
;; This allows exec to recognise a nested query and
;; handle it.
;; The function recognises based on the number of
;; arguments whether it is called inside a CPS transformed
;; query or outside of it and behaves accordingly.
(let [args (:source (meta query))
      [params source]
      (if (or (symbol? (second args))
              (vector? (second args)))
        [(second args) (nthrest args 2)]
        ['params args])
      nParams (count params)]
  (with-meta
    (fn [& value]
      (cond
        ;; Outside a query
        (= (count value) nParams)
        (->subquery query value))
    [algorithm & options] options))
```

```clojure
23
```
;; Inside a query (CPS code)
(= (count value) (+ nParams 2))
(let [[cont state & value] value]
  (fn [] (cont (->subquery query value) state)))

:else
(throw (Exception. "Wrong number of args for nested query"))

{:source (:source (meta query))
 :inference :no-op})

(fn [& value]
  
  ;;

  ;; Existing implementation of conditional
)

Code from anglican.state:

; Should I add stack to the initial-state? Don’t need to.
(defn push-stack
  [state cont]
  -> state
  (update-in ‘[:stack]
    conj
    {:predicts (:predicts state)
     :store (:store state)
     :cont cont})
  (assoc :predicts [])
  (assoc ::store nil))

(defn peek-stack
  [state]
  (peek (::stack state))

(defn pop-stack
  [state]
  (let [env (peek-stack state)]
    -> state
    (assoc :predicts (:predicts env))
    (assoc ::store (:store env))

24
(assoc ::stack (pop (:stack state))))

Code from `exec` in `anglican.inference`:

```
(defn exec
  "executes the program, calling checkpoint handlers
  at the checkpoints and stopping when the handler
  returns a non-callable value. Also deals with nested queries
  by pushing the current environment on a stack (when entering)
  or retrieving from it (when exiting)"
  [algorithm prog value state]
  (loop [step (trampoline prog value state)]
    (cond
      ;; If the distribution to sample from is a subquery,
      ;; do not pass it to the checkpoint, but instead push
      ;; current env to stack and call subquery
      (and (instance? anglican.trap.sample step)
           (instance? anglican.trap.subquery (:dist step)))
        (let [state (push-stack (:state step) (:cont step))
               value (into [] (get-in step [:dist :value]))]
          (recur (trampoline
                   (get-in step [:dist :query])
                   value
                   state)))
      ;; When encountering a 'result' type
      ;; (indicating the end of a query), check whether
      ;; stack contains envs. If so, restore it and pass
      ;; the predicted values of the subquery to the
      ;; continuation of the outer query
      (and (instance? anglican.trap.result step)
           (not (nil? (peek-stack (:state step))))
           (let [predicts (get-predicts (:state step))
                 cont (:cont (peek-stack (:state step)))
                 state (pop-stack (:state step))]
            (recur (trampoline
                     cont
                     predicts
                     state)))
      ;; Otherwise, the interruption should be handled by
      ;; the backend inference checkpoints
      :else
```

25
(let [next (checkpoint algorithm step)]
  (if (fn? next)
    (recur (trampoline next))
    next))))

B. Code for examples

This is the code for the examples of section 2.5.

For the 'flattend' case which was used as reference:

(with-primitive-procedures [mu0 g f]
  (defquery flatq []
    (let [theta (abs (sample (normal 0 10)))
          d (sample (normal 0 10))
          x0 (sample (mu0))]
      (predict :theta theta)
      (predict :d d)
      (observe (g x0) (first ys))
      (loop [ys (rest ys)
             xs [x0]]
          (if (empty? ys)
              (predict :xs xs)
              (let [x (sample (f (+ (last xs) d) theta))]
                 (observe (g x) (first ys))
                 (recur
                  (rest ys)
                  (conj xs x))))))))

And for the case of the 'old' conditional which runs one inference within another. Note that with-primitive-procedures must be used to import one-step-posterior.

(def one-step-posterior
  (conditional one-step-query
               :pimh
               :number-of-particles 10))

(with-primitive-procedures
  [mu0 g one-step-posterior]
  (defquery oldq []
    (let [theta (abs (sample (normal 0 10)))
          d (sample (normal 0 10))
          x0 (sample (mu0))]
      26)
(predict :theta theta)
(predict :d d)
(observe (g x0) (first ys))
(loop [ys (rest ys)
   xs [x0]]
  (if (empty? ys)
    (predict :xs xs)
    (let [x
       (:x (sample
            (one-step-posterior
             (last xs) (first ys) theta d)))]
      (recur
       (rest ys)
       (conj xs x))))))
References


