Design and Implementation of Probabilistic Programming Language Anglican

David Tolpin
University of Oxford
dtolpin@robots.ox.ac.uk

Jan-Willem van de Meent
Northeastern University
j.vandemeent@northeastern.edu

Hongseok Yang
University of Oxford
hongseok.yang@cs.ox.ac.uk

Frank Wood
University of Oxford
fwood@robots.ox.ac.uk

ABSTRACT

Anglican is a probabilistic programming system designed to interoperate with Clojure and other JVM languages. We introduce the programming language Anglican, outline our design choices, and discuss in depth the implementation of the Anglican language and runtime, including macro-based compilation, extended CPS-based evaluation model, and functional representations for probabilistic paradigms, such as a distribution, a random process, and an inference algorithm.

We show that a probabilistic functional language can be implemented efficiently and integrated tightly with a conventional functional language with only moderate computational overhead. We also demonstrate how advanced probabilistic modelling concepts are mapped naturally to the functional foundation.

CCS CONCEPTS

• **Software and its engineering** → *Functional languages*; *Specialized application languages*;

ACM Reference format:
David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of Implementation and Application of Functional Programming Languages, Leuven, Belgium, August 2016 (IFL 2016),* 12 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064910

1 INTRODUCTION

For data science practitioners, statistical inference is typically just one step in a more elaborate analysis workflow. The first stage of this work involves data acquisition, preprocessing and cleaning. This is often followed by several iterations of exploratory model design and testing of inference algorithms. Once a sufficiently robust statistical model and a corresponding inference algorithm have been identified, analysis results must be post-processed, visualized, and in some cases integrated into a wider production system.

Probabilistic programming systems [6, 7, 9, 31] represent generative models as programs written in a specialized language that provides syntax for the definition and conditioning of random variables. The code for such models is generally concise, modular, and easy to modify or extend. Typically inference can be performed for any probabilistic program using one or more generic inference techniques provided by the system back end, such as Metropolis-Hastings [9, 29, 33], Hamiltonian Monte Carlo [23], expectation propagation [11], and extensions of Sequential Monte Carlo [15, 28, 31] methods.

While probabilistic programming systems shorten the iteration cycle in exploratory model design, they typically lack basic functionality needed for data I/O, pre-processing, and analysis and visualization of inference results. In this paper, we describe the implementation of Anglican [26, 32], a probabilistic programming language that tightly integrates with Clojure [8], a general-purpose programming language that runs on the Java Virtual Machine (JVM). Both languages share a common syntax, and can be invoked from each other. This allows Anglican programs to make use of a rich set of libraries written in both Clojure and Java. Conversely, Anglican allows intuitive and compact specification of models for which inference may be performed as part of a larger Clojure project.

There are several ways to build a new language on top of an existing one; Lisp is famous for the macro facility that allows to extend the language almost without restriction — by writing *macros*, one adds new constructs to the existing language. There are several uses of macros — one is to extend the language *syntax*, for example, by adding new control structures; another is to keep the existing syntax but alter the *operational semantics* — the way programs are executed and compute their outputs.

Anglican is implemented in just this way — a macro facility provided by Clojure, a Lisp dialect, is used both to extend Clojure with constructs that delimit probabilistic code, and to alter the operational semantics of Clojure expressions inside probabilistic code fragments. Anglican claims its right to count as a separate language because of the ubiquitous probabilistic execution semantics rather than a different syntax, which is actually an advantage rather than a drawback — Clojure programmers only need to know how to specify the boundaries of Anglican programs, but can use familiar Clojure syntax to write probabilistic code.

Inference algorithms execute Anglican programs while trying to answer probabilistic queries on those programs. This execution is significantly different from the one described in the standard operational semantics. These algorithms typically run Anglican programs multiple times, often hundreds of thousands or even millions of times for a single inference task. The algorithms may make random choices that do not correspond to any statements in the program, and decide which parts of the program code are executed and how often. Some inference algorithms re-run the program multiple times partially, from a certain point on, while reusing random choices made in the previous runs as much as possible. A good high-level picture is that each inference algorithm specifies a virtual machine that executes Anglican programs according to a non-standard (usually probabilistic) operational semantics. Supporting a wide range of inference algorithms and their unusual semantics the main reason that we developed Anglican as a language rather than as a library.

Comparisons of Anglican with other implementations of probabilistic programming languages [21][16, pp. 32–33] demonstrate that Anglican achieves state-of-the-art computational efficiency without sacrificing expressiveness. Anglican language syntax, compilation, invocation, and runtime support of Anglican queries are discussed in detail in further sections.

Contributions

This paper brings the following major contributions:

- Design and implementation of a probabilistic programming language Anglican involving tight bilateral integration with a general-purpose programming language.
- Techniques for efficient and compact implementation of inference algorithms, such as representation of inference results as a lazy sequence of samples and a novel scheme for addressing of checkpoints.
- Anglican's novel representation of random processes from statistics and machine learning, such as beta-Bernoulli process. The representation is *stateless* and seamlessly integrates into pure functional computation. This contrasts with typical *stateful* implementations of these processes in other probabilistic programming languages.

2 RELATED WORK

Efficient implementation of expressive probabilistic programming languages has recently been an active area of research [5–7, 9–11, 18, 23, 31]. There is often a compromise between the expressiveness of the language and the efficiency of inference. Some languages emphasize expressiveness [6, 7, 9, 10]; others restrict the class of models which can be expressed by the language [11, 13, 23] to facilitate application of efficient inference algorithms. Anglican allows unrestricted specification of probabilistic models in the spirit of Church [6], while still supporting efficient scalable inference [15, 20, 31].

Probabilistic programming languages are implemented as interpreters [6, 9], embedded languages [5, 18, 21], and compiled languages [10, 23], and also through source-to-source transformation with augmentation [7, 29]. Each of these methods emphasizes different priorities in language design, such as computational efficiency, integration with an existing development environment, ease of implementation, or efficiency of inference. Anglican is implemented using a combination of embedding and source-to-source transformational compilation to combine advantages of both approaches. CPS transformation, employed by Anglican compiler, is also used in WebPPL [7] and facilitates clean separation between probabilistic programs and inference algorithms and diversity of applicable inference algorithms.

Probabilistic programming languages are often implemented as pure functional languages [6, 9, 21] in the sense that they do not allow mutable states. This is because reasoning about probabilistic programs and distributions defined by the programs, as well as implementation of certain inference algorithms [28], is easier in such pure functional setting. Anglican is also implemented as a language without mutable states. However, there are also languages which follow imperative paradigm and support mutable state directly [5, 23], or through probabilistic primitives [6, 9]. Anglican introduces a functional alternative to stateful random primitives in the form of random processes.

3 DESIGN OUTLINE

An Anglican program, or *query*, is compiled into a Clojure function. When inference is performed with a provided algorithm, this produces a sequence of samples. Anglican shares a common syntax with Clojure; Clojure functions can be called from Anglican code and vice versa. A simple program in Anglican can look like the following code:

```
(defquery model "model selection" data
  (let [;;; Guess a distribution.
        dist (sample (categorical
                       [[normal 0.5]
                        [gamma 0.5]]))
        a (sample (gamma 1 1))
        b (sample (gamma 1 1))
        d (dist a b)]
    ;;; Observe samples from the distribution.
    (loop [observations data]
      (when (not-empty observations)
        ;; Retrieve the first observation as ‘o’
        (let [o (first observations)]
          ;; Observe ’o’ from the guessed
          ;; distribution ’d’.
          (observe d o))
        ;; Proceed to the next iteration with
        ;; the rest of observations.
        (recur (rest observations))))
    ;;; Return the distribution and parameters.
    [d a b]))
```

The query builds a model for the input `data`, a sequence of data points. It defines a probability distribution on three variables, $\mathtt{d} \in \{\mathtt{normal}, \mathtt{gamma}\}$ for a distribution type, and `a` and `b` for positive parameters for the type. Concretely, using the `sample` forms, the query first defines a so called prior distribution on these three variables, and then it adjusts this prior distribution based on observations in `data` using the `observe` form. Samples from this conditioned distribution (also called posterior distribution) can be obtained by running the query under one of Anglican's inference algorithms.

Clojure (and Anglican) programs run on the JVM and are able to make use of a wide range of Java libraries for data processing, networking, presentation, and imaging. Conversely, Anglican queries can be called from Java and other JVM languages. Programs involving Anglican queries can be deployed as JVM *jars*, and run without modification on any platform for which the JVM is available.

A probabilistic program, or query, mostly runs deterministic code. Aside from the special forms `sample` and `observe`, which are probabilistic in nature, Anglican can be implemented as a regular programming language. At `sample` and `observe` forms, normal deterministic execution is interrupted, and Anglican programs must allow the inference algorithm to step in, recording information and affecting control flow. We refer these points in the execution as checkpoints. Handling of checkpoints can be implemented through coroutines/cooperative multitasking, and parallel execution/preemptive multitasking, as well as through explicit maintenance of program continuations. Anglican follows the latter option.

Internally, an Anglican query is represented by a computation in *continuation passing style* (CPS) [1]. The Anglican 'compiler' accepts a Clojure subset and transforms it into a variant of CPS representation, which allows inference algorithms to intervene in the execution flow at probabilistic checkpoints [1]. The available inference algorithms include the Particle Cascade [15], Lightweight Metropolis-Hastings [29], Iterative Conditional Sequential Monte-Carlo (Particle Gibbs) [31], and others. Inference on Anglican queries generates a lazy sequence of samples, which can be processed asynchronously in Clojure for analysis, integration, and decision making.

Anglican is intended to co-exist with Clojure and be a part of the source of a Clojure program. To facilitate this, Anglican programs, or queries, are wrapped by macros which call the CPS transformations and define Clojure values suitable for passing as arguments to inference algorithms (`defquery`, `query`). In addition to defining entire queries, Anglican promotes modularization of probabilistic models through the definitions of *probabilistic functions* using `defm` (Anglican counterpart of Clojure `defn`). Probabilistic functions are written in Anglican, may include probabilistic forms `sample` and `observe`, and can be seamlessly called from inside Anglican queries, just like functions locally defined within the same query.

[1] [7] also describe a CPS-based implementation of a probabilistic programming language.

The operational semantics of an Anglican query are different from those of Clojure code, and are determined by an inference algorithm. Thus, Anglican queries must be called through these inference algorithms, rather rather than 'directly'. For this purpose, Anglican declares the (ad-hoc) polymorphic function `infer` using Clojure's multimethod mechanism. This function accepts and runs an Anglican query, and returns a lazy sequence of weighted samples from the distribution defined by the query. Providing an implementation of this function is a responsibility of the inference algorithm, which should also override the polymorphic function `checkpoint` (defined as a multimethod) so as to handle `sample` and `observe` in an algorithm-specific manner and to construct an appropriate result on the termination of a probabilistic program.

Finally, Anglican queries use 'primitive', or commonly known and used, distributions, to draw random samples and condition observations. Many primitive distributions are provided by the runtime, and an additional distribution can be defined by the user by implementing a particular set of functions for the distribution (via Clojure's protocol mechanism). The `defdist` macro provides a convenient syntax for defining primitive distributions.

4 PROBABILITY OF A PROGRAM EXECUTION

Anglican programs define probability distributions over sequences of values, implicitly by means of program execution. A good way to understand this is to imagine the following interpreter of Anglican programs. Starting from a fixed initial state, the interpreter runs the deterministic parts of a program according to the standard semantics, executes the `sample` form by generating a random sample, and treats the `observe` form by skip. More importantly, the interpreter keeps a log that records information about all the `sample` and `observe` forms encountered during execution. The information recorded for `sample` is a triple (F, x, α) of (i) a primitive probability distribution F, such as the standard normal, for which we have the probability density p_F; (ii) a value x sampled from the distribution F; and (iii) an address α that uniquely and systematically identifies the random choice made. The information recorded for `observe` is a pair (G, y) where G is a primitive probability distribution as F from above and y is an observed value. Thus, a log is a sequence of triples (F, x, α) and pairs (G, y).

One important property of logs is that they are determined by their projections to triples: when two logs project to the same sequence of triples, they must be the same. This is because the triples in a log contain all the information about random choices made during execution and all the non-sample forms in Anglican programs are deterministic. We define a *trace* $\boldsymbol{x}$ to be a sequence of triples (F, x, α), and say that $\boldsymbol{x}$ is feasible if the trace is precisely the triple part of the log of some execution. Such a feasible trace uniquely determines the rest of the execution and its log. In particular, it decides the pair part of the log, namely, a sequence of (G_j, y_j) for observed values. We call this sequence *image of* $\boldsymbol{x}$ and denote it by $\boldsymbol{y}$.

A (almost-surely terminating) probabilistic program defines a probability distribution over finite feasible traces $\boldsymbol{x}$ with probability density $\pi(\boldsymbol{x}) := \gamma(\boldsymbol{x})/Z$ where

$$\gamma(\boldsymbol{x}) := \prod_{i=1}^{|\boldsymbol{x}|} p_{F_i}(x_i) \prod_{j=1}^{|\boldsymbol{y}|} p_{G_j}(y_j) \quad \text{for the image } \boldsymbol{y} \text{ of } \boldsymbol{x}, \quad (1)$$

and Z is the normalization constant $Z := \int \gamma(\boldsymbol{x}) d(\boldsymbol{x})$. The integral and the density use the default measure obtained by the standard extension of a σ-finite measure on finite traces where the σ-finite measure itself is defined by a countable sum of Lebesgue and counting measures.

5 LANGUAGE

5.1 Syntax

The Anglican language is a subset of Clojure[2]. Anglican queries are defined within defquery. The value of the last expression in the query body is the result of the query.

```
(defquery name doc-string? param? expr*)
```

Anglican functions outside of a query are defined using defm with the same syntax as Clojure defn, however name is bound to an Anglican function.

```
(defm name doc-string? [param*] expr*)
```

Within the body of defquery and defm, the following special forms are supported, with syntax and semantics matching those of Clojure:

```
(if pred then else?)
(when pred expr*)
(cond clause*)
(case expr clause* default?)
(let [binding*] expr*)
(and expr*)
(or expr*)
(fn name? [param*] expr*)
(loop [binding*] expr*)
(recur expr*)
```

In defquery parameter lists, let bindings, and fn argument lists, Clojure vector destructuring is supported. Literals for atomic types and compound literals for vectors, hash maps, and sets are supported just like in Clojure.

Clojure provides special forms loop and recur for writing tail-recursive programs. Anglican programs are CPS-converted and do not use the stack; recursive calls in Anglican cannot lead to stack overflow. However, loop and recur are provided in Anglican for convenience as a way to express loops. Unlike in Clojure though, recur is supported only inside a loop.

[2]It would be possible to support almost full Clojure by expanding all macros in the Anglican source code. However, in Clojure, unlike in Scheme [22] or Common Lisp [19], the result of macro-expansion of derived special forms is not well specified and implementation specific.

5.2 Core Library

All of Clojure's core library except for higher-order functions (functions that accept other functions as arguments) is available in Anglican. Higher-order functions cannot be reused from Clojure, as they have to be re-implemented to accept functional arguments in CPS form. The following higher-order functions are implemented: map, reduce, filter, some, repeatedly, comp, partial.

5.3 Special Forms

In addition to re-implementing a subset of Clojure, Anglican provides special forms for probabilistic inference:

```
(sample address? distribution)
(observe address? distribution value)
(mem function)
(store tag value)
(retrieve tag)
```

A good way to understand the sample and observe forms is to consider the imaginary interpreter in Section 4. Under this interpreter, the sample form draws a value x from its parameter distribution F, and generates a unique address α for this random draw. Then, it appends (F, x, α), the record of this random draw, to the log of the current execution. Finally, the form returns x. The computation of the observe form is simpler. It just appends its two parameters, distribution G and observed value v, to the log of the current execution, and returns nil.

What we have just described is idealised semantics of sample and observe. The actual computation involved in handling sample and observe depends on the inference algorithm. Different inference algorithms may treat sample and observe differently as long as they compute the same distribution on traces (i.e. sequences of sampled values), namely, the one in Equation 1. Formal semantics for an idealized version of Anglican or other higher-order probabilistic programming languages are introduced in [4, 24]; in general though, specifying semantics of higher-order probabilistic programming languages is an open problem.

sample and observe may appear anywhere in the code of an Anglican program. To support these forms, the inference engine must be able to intervene into execution of the program, perform computations related to inference, and control further execution of the program. This tight and complicated interaction between the program and the inference engine necessitates implementing Anglican as a language rather than a library.

The remaining Anglican special forms support carrying values in the hidden program state. mem implements stochastic function memoization (Section 7.3). store stores a value for a tag in the program state, which can be later retrieved using retrieve during the same program execution.

6 CASE STUDY

Before delving into implementation details, we discuss a case study of the use of the Anglican language and environment.

This case study takes an inference problem for which a solution is not immediately obvious, *the Deli dilemma*, and discusses how this problem can be solved by writing an Anglican program, executing the program, and post-processing results.

The program presented in this section is intentionally short and simple. Anglican is capable of compiling and running elaborate programs and handling large amounts of data. Advanced examples of Anglican programs and inference can be found in literature on applications of probabilistic programming [16, 17, 27].

6.1 The Deli Dilemma

Imagine that we are facing the following dilemma:

A customer wearing round sunglasses came to a deli at 1:13pm, and grabbed a sandwich and a coffee. Later on the same day, a customer wearing round sunglasses came at 6:09pm and ordered a dinner. Was it the same customer?

Additionally, we know that:

- There is an adjacent office quarter, and it takes between 5 and 15 minutes on average to walk from an office to the deli, where different average times are for different buildings in the quarter.
- Depending on traffic lights, the walking time varies by about 2 minutes.
- The lunch break starts at 1:00pm, and the workday ends at 6:00pm.
- Based on the similarity of appearance, the waiter assesses the odds that this is the same customer as 2 to 1.

6.2 Anglican Query

We want to formalize the dilemma as an Anglican query, based on the knowledge we have. Let us formalize the knowledge in Clojure (the times are in minutes). First, we encode our prior information which holds true independently of the customer's visit:

```
(def mean-time-to-arrive "average time to arrive" 10.)
(def sd-time-to-arrive
     "standard deviation of arrival time" 3.)
(def time-sd "walking time deviation" 1.)
```

Then, we record our observations, based on which we want to solve the dilemma:

```
(def lunch-delay
     "time between lunch break and lunch order" 13.)
(def dinner-delay
     "time between end of day and dinner order" 9.)
(def p-same
     "prior probability of the same customer" (/ 2. 3.))
```

For inference, one often chooses a known distribution to represent uncertainty. We choose the normal distribution for representing uncertainty about average arrival time.

```
(def time-to-arrive-prior
     "prior distribution of average arrival time"
     (normal mean-time-to-arrive sd-time-to-arrive))
```

There are two possibilities: either the same customer visited the deli twice, or two different customers came to the deli, one for lunch and the other for dinner. We define an **Anglican** function for each case. Note that this is the first time we switch from Clojure to Anglican. The functions must be written in Anglican (and hence defined using `defm` instead of `defn`) because they contain probabilistic forms `sample` and `observe`.

```
(defm same-customer
  "observe the same customer twice"
  [time-to-arrive-prior lunch-delay dinner-delay]
  (let [time-to-arrive (sample time-to-arrive-prior)]
    (observe (normal time-to-arrive time-sd)
             lunch-delay)
    (observe (normal time-to-arrive time-sd)
             dinner-delay)
    [time-to-arrive]))

(defm different-customers
  "observe different customers"
  [time-to-arrive-prior lunch-delay dinner-delay]
  (let [time-to-arrive-1 (sample time-to-arrive-prior)
        time-to-arrive-2 (sample time-to-arrive-prior)]
    (observe (normal time-to-arrive-1 time-sd)
             lunch-delay)
    (observe (normal time-to-arrive-2 time-sd)
             dinner-delay)
    [time-to-arrive-1 time-to-arrive-2]))
```

Both functions have the same structure: we first 'guess' the average arrival time and then observe the actual time from a distribution parameterized by the guessed time. However, in `same-customer` the average arrival time is the same for both the lunch and the dinner, while in `different-customers` two average arrival times are guessed independently.

We are finally ready to define the query in **Anglican**.

```
(defquery deli [time-to-arrive-prior
                lunch-delay dinner-delay]
  (let [is-same-customer (sample (flip p-same))
        observe-customer (if is-same-customer
                           same-customer
                           different-customers)]
    {:same-customer is-same-customer,
     :times-to-arrive (observe-customer
                       time-to-arrive-prior
                       lunch-delay dinner-delay)}))
```

Performing inference on the query `deli` computes the probability that the same customer visited the deli twice, as well as probability distributions of average arrival times for both cases.

6.3 Inference

Having defined the query, we are now ready to run the query using an inference algorithm. Function `doquery` accepts the inference algorithm, the query, and optional parameters, and returns a lazy sequence of samples. We use here the inference algorithm called Lightweight Metropolis-Hastings (LMH). We bind the results of `doquery` to variable `samples`, to analyse the results later. However, since the sequence is lazy, no inference is performed and no samples are generated until they are retrieved and processed.

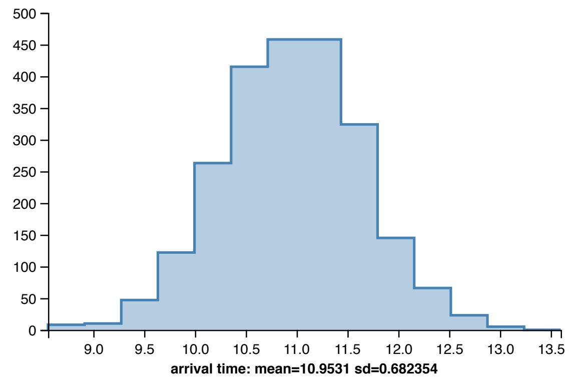

Figure 1: Arrival time distribution for a single customer.

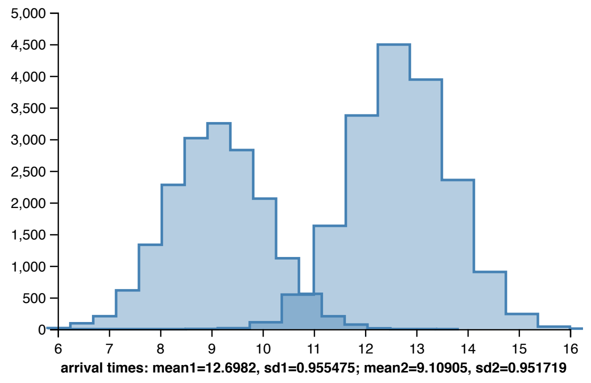

Figure 2: Arrival time distributions for two customers.

```
(def samples (doquery :lmh deli nil))
```

To approximate the inferred distribution, we extract a finite subsequence of samples.

```
(def N 5000)
(def results (map get-result (take N (drop N samples))))
```

Based on the collected samples we compute an approximation of the posterior probability `p-same+` that the same customer visited the deli twice.

```
(def p-same+ (/ (count (filter :same-customer results))
                (double N)))
```

The `:same-customer` here represents a function that looks up the entry with the same name in a given map. The map stores the result of a single sampled execution, and the `:same-customer` entry in the map records whether the same customer visits the deli in the execution. The `filter` function in `p-same+` goes through all the maps in `results` and picks only the ones whose executions involve just one customer.

With the specified observations, `p-same+` is ≈ 0.12. The probability is much lower than the waiter's guess `p-same` $(\frac{2}{3})$. Of course, the results may vary from run to run, and, for a given algorithm, the accuracy depends on the number of samples we decide to retrieve.

Besides computing the posterior probability that the same customer visited the deli twice, we may want to know the average time (or times) to arrive. In Bayesian inference, it is common to report distributions instead of 'most likely' values. We use query results in retrieved samples to approximate the distributions, and plot distribution histograms for the same customer visiting twice (Figure 1) and two different customers (Figure 2).

This completes the case study, where we showed a probabilistic programming solution of a problem, implemented and analysed in Anglican and Clojure.

7 MACRO-BASED COMPILATION

Anglican queries are compiled into Clojure using a variant of CPS transformation. In a basic CPS-transformed program, a continuation receives a single argument — the computed value. In Anglican, two flows of computation are performed in parallel: values are computed by functional code, and, at the same time, the state of the probabilistic program, used by inference algorithms, is updated by probabilistic forms. Because of that, in Anglican a continuation accepts *two* arguments:

- the computed value;
- the internal state, bound to the local variable `$state` in every lexical scope.

The compilation relies on the Clojure *macro* facility, and implemented as a library of *functions* invoked by macros. The CPS transformation is organized in top-down manner. The top-level function is `cps-of-expression`, which receives an expression and a continuation, and returns the expression in the CPS form. For example, the CPS transformation of constant `1` with continuation `cont` thus takes the following form:

```
=> (cps-of-expression 1 'cont)
(cont 1 $state)
```

7.1 The State

The state (`$state`) is threaded through the computation and contains data used by inference algorithms. `$state` is a Clojure hash map:

```
(def initial-state
  "initial program state"
  {:log-weight 0.0, ;; map :log-weight to 0.0
   :result nil,     ;; map :result to nil
   ::store nil,     ;; map ::store to nil
   ::mem {},        ;; map ::mem to the empty map
   ... })
```

which records inference-relevant information under various keys such as `:log-weight` and `::mem`. The full list of map entries depends on the inference algorithm. CPS transformation routines are not aware of the contents of `$state` and do not access or modify it directly. Rather, they just thread the state unmodified through the computation. The sole exception is the transformation of the `mem` form (which converts a function to one with memoization). Algorithm-specific handlers of checkpoints corresponding to the probabilistic forms (`sample`, `observe`) modify the state and insert an updated state into the computation.

7.2 Probabilistic Forms

The purpose of probabilistic forms `sample` and `observe` is to interrupt deterministic computation and transfer control to the inference algorithm. Practically, this is achieved by returning a record that represents the state of program execution at the checkpoint, rather than invoking a continuation. The Clojure record used to interrupt execution at `sample` and `observe` forms are `anglican.trap.sample` and `anglican.trap.observe` respectively. These records contain a unique identifier (which we will discuss in detail in section 8.3), the arguments to the special form, the continuation, and the state.

```
=> (cps-of-expression '(sample dist) 'cont)
(->sample 'S1 dist cont $state)
=> (cps-of-expression '(observe dist v) 'cont)
(->observe 'O1 dist v cont $state)
```

Here `->sample` and `->observe` in the CPS-transformed expressions are constructors for Clojure records, and they take values of their fields as arguments. Calling the continuation resumes the computation.

Upon encountering a `sample` or `observe` record, the inference algorithm computes the updated program state and the value to be passed to the continuation. How the state is updated, the number of times the continuation is called, and the value passed to the continuation of `sample` depend on the inference algorithm executing the program. Section 8 provides more detail on internals of inference algorithms and their interaction with probabilistic programs.

7.3 Memoization

The author of a probabilistic model might want to randomly draw a feature value for each entity in a collection, but to retain the same drawn value for the same entity during a single run of the probabilistic program. For example, a person may have brown or green eyes with some probability, but the *same* person will always have the same eye color. This can be achieved through the use of memoized functions. In Anglican, one might write:

```
(let [eye-color (mem (fn [person]
                       (sample (categorical ['brown 0.5]
                                            ['green 0.5]))))]
  (if (not= (eye-color 'bill) (eye-color 'john))
    (eye-color 'bill)
    (eye-color 'john)))
```

The `mem` form converts a function to a memoized variant, which remembers past inputs and the corresponding outputs, and returns the remembered output when given such a past input, instead of calling the original function with it. As a result, for every input, random draws will be made only for the first time that the memoized function is called with the input; all subsequent calls with the input will just reuse these draws and return the same output.

Memoization is often implemented on top of a mutable dictionary, where the key is the argument and the value is the returned value. However, there are no mutable data structures in a probabilistic program. Hence, `mem`'s memory is stored as a nested dictionary in the program state introduced during CPS transformation (function `mem-cps`). Every memoized function gets a unique automatically generated identifier. Each time a memoized function is called, one of two continuations is chosen, depending on whether the same function (a function with the same identifier) was previously called in the same run of the probabilistic program with the same argument. If the memoized result is available, the continuation of the memoized function call is immediately called with the stored result. Otherwise, the argument of `mem` is called with a continuation which first creates an updated state with the memoized result, and then calls the 'outer' continuation with the result and the updated state:

```
=> (mem-cps '(foo))
(let [M (gensym "M")]
  (fn mem23623 [C $state & P]
    (if (in-mem? $state M P)
      ;; previously memoized result
      (fn [] (C (get-mem $state M P) $state))
      ;; new computation
      (clojure.core/apply foo
        ;; memoize result in state
        (fn [V $state]
          (fn [] (C V (set-mem $state M P V))))
        $state P)))))
```

Memoized results are not shared among multiple runs of a probabilistic program, which is intended. Otherwise, it would be impossible to memoize functions with random results.

7.4 Managing stack size

Continuations may lead to unbounded stack growth in recursive code. In implementations of functional programming languages stack growth is avoided through *tail call optimization* (TCO). However, Clojure does not support a general form of TCO, and CPS-transformed code that creates deeply nested calls will easily exhaust the stack. Anglican employs a workaround called *trampolining* [3] — instead of inserting a continuation call directly, the transformer always wraps the call into a *thunk*, or parameterless function. The thunk is returned and called by the trampoline (Clojure provides function `trampoline` for this purpose) — this way the computation continues, but the stack is collapsed on every continuation call. Function `continue` implements the wrapping and is invoked on every continuation call:

```
=> (continue 'cont 'value 'state)
(fn [] (cont value state))
```

Correspondingly, the full, wrapped CPS form of

```
(fn [x y] (+ x y))
```

becomes

```
(fn [cont $state x y] (fn [] (cont (+ x y) $state)))
```

When the CPS-transformed function is called, it returns a *thunk* (a parameterless function) which is then re-invoked through the trampoline, with the stack collapsed.

7.5 Primitive Procedures

When an Anglican function is transformed into a Clojure function, two auxiliary parameters are added to the beginning of the parameter list — continuation and state. Correspondingly, when a function *call* is transformed, the current continuation and the state are passed to the called function. Anglican can also call Clojure functions; however, Clojure functions do not expect these auxiliary parameters. To allow the mixing of Anglican (CPS-transformed) and Clojure function calls in Anglican code, the Anglican compiler must CPS-transform expressions selectively [12]. It must be able to recognize 'primitive' (that is, implemented in Clojure rather than in Anglican) functions, and invoke those functions in a direct, not CPS, style.

Providing an explicit syntax for differentiating between Anglican and Clojure function calls would be cumbersome. Another option would be to use meta-data to identify Anglican function calls at runtime. However, this would impact performance, and good runtime performance is critical for probabilistic programs. The approach taken by Anglican is to maintain a list of unqualified names of primitive functions, as well of namespaces in which all functions are primitive, and recognize primitive functions by name — if a function name is not in the list, the function is an Anglican function. Global dynamically-bound variables `*primitive-procedures*` and `*primitive-namespaces*` contain the initial lists of names and namespaces, correspondingly. Of course, local bindings can shade global primitive function names. For example, `first` is a Clojure primitive function that takes the first element from an ordered collection such as list and vector, but inside the let block in the following example, `first` is an Anglican function:

```
(let [first (fn [[x & y]] x)]
  (first '[1 2 3]))
```

The Anglican compiler takes care of the shading by rebinding `*primitive-procedures*` in every lexical scope.

8 INFERENCE ALGORITHMS

A probabilistic program in Anglican may look almost like a Clojure program. However, the purpose of executing a probabilistic program is different from that of a 'regular' program: instead of producing the result of a single execution, a probabilistic program computes, exactly or approximately, the distribution from which execution results are drawn. Characterising this distribution is done by the inference algorithm.

Probabilistic programming system Anglican provides a variety of approximate inference algorithms. Ideally, Anglican should automatically choose the most appropriate algorithm for each probabilistic program. In practice, selecting an inference algorithm, or a combination thereof, is still a challenging task, and program structure, intended use of the computation results, performance, approximation accuracy, and other factors must be taken into consideration. New algorithms are being developed and added to Anglican [20, 25, 27], as a part of ongoing research.

In the implementation of Anglican, inference algorithms are instantiations of the (ad-hoc) polymorphic function `infer` declared as Clojure's multimethod. The function accepts an algorithm identifier, a query (the probabilistic program in which to perform the inference), an initial value for the query, and optional algorithm parameters.

8.1 The `infer` Function

The sole purpose of the algorithm identifier of `infer` is to invoke an appropriate overloading or implementation of the function — conventionally, the identifier is a Clojure keyword (a symbolic constant starting with colon) related to the algorithm name, such as `:lmh` for Lightweight Metropolis-Hastings and `:pcascade` for Particle Cascade. The second parameter is a query as defined by the `defquery` form or its anonymous version `query`. For instance, the following Clojure code invokes `infer` on an Anglican query defined anonymously via the `query` form:

```
(let [x 1]
  (infer :pgibbs (query x) nil))
```

A query is executed by calling the initial continuation of the query, which accepts a value and a state. The state is supplied by the inference algorithm, while the value is provided as a parameter of `infer`. A query does not have to have any parameters, in which case the value can be simply `nil`. When a query is defined with a binding for the initial value, the value becomes available inside the query. A query may accept multiple parameters using Clojure's structured binding. For instance, it may take multiple parameters as components of an input vector. In this case, the initial value is given as a structured value, such as a vector, and the components of this value are matched to the corresponding parameters of the query via the destructuring mechanism of Clojure. For example,

```
(defquery my-query [mean sd]
  (sample (normal mean sd)))

(def samples (infer :lmh my-query [1.0 3.0]))
```

Finally, any number of auxiliary arguments can be passed to `infer`. By convention, the arguments should be keyword arguments, and are interpreted in the algorithm-specific manner.

8.2 Internals of an Inference Algorithm

Implementing an inference algorithm in Anglican amounts to defining an appropriate version of the `infer` function and checkpoint handlers for `sample` and `observe`. The definitions may just reuse default implementations or override them with new algorithm-specific treatment of `sample` and `observe` forms and inference state.

Let us illustrate this implementation step with *importance sampling*, the simplest inference algorithm. Here is an implementation of the `infer` function for the algorithm:

```
;; Make ::algorithm be converted to
;; :anglican.inference/algorithm whenever necessary
(derive ::algorithm :anglican.inference/algorithm)
```

```
;; invoked when algo parameter is :importance
(defmethod infer :importance [algo prog value & {}]
  (letfn
    ;; recursive function without any parameter
    [sample-seq []
      ;; lazy infinite sequence
      (lazy-seq (cons
                  (:state (exec ::algorithm prog value
                                initial-state))
                  (sample-seq)))]
    (sample-seq)))
```

This implementation is dispatched when `infer` receives `:importance` as its `algo` parameter. (Anglican includes multiple implementations of `infer` for different inference algorithms.) Once dispatched, it lazily constructs an infinite sequence of inference states by repeatedly executing the program `prog` using `exec` and retrieving the final inference state of the execution using `:state`. For checkpoint handlers, importance sampling simply relies on their default implementations.

Typically, an inference algorithm has its own implementations of `checkpoint` for `sample`, `observe`, or both, as well as invokes `exec` from an elaborated conditional control flow. LMH (`anglican.lmh`) and SMC (`anglican.smc`) are examples of inference algorithms where either `observe` (SMC) or `sample` (LMH) handler is overridden. In addition, SMC runs multiple particles (corresponding to user-level threads) simultaneously, while LMH re-runs programs from an intermediate continuation rather than from the beginning.

8.3 Addressing of Checkpoints

Many inference algorithms memoize and reuse earlier computations at checkpoints. Variants of Metropolis-Hastings reuse previously drawn values [29], as well as additional information used for adaptive sampling [25] at `sample` checkpoints. Asynchronous SMC (Particle Cascade) computes average particle weights at `observe` checkpoints [15]. Implementations of black-box variational inference [27, 30] associate with random variables the learned parameters of variational posterior distribution.

Checkpoints can be uniquely named at compilation time; however, at runtime a checkpoint corresponding to a single code location may occur multiple times due to recurrent invocation of the function containing the checkpoint. Every unique occurrence of a checkpoint must receive a different address. An addressing scheme based on computing of stack addresses of checkpoints was described in the context of Lightweight Metropolis-Hastings [29]. This scheme has advantages over the naive scheme where dynamic occurrences of checkpoints are numbered sequentially. However, it impacts the performance of probabilistic programs because of the computation cost of computing the stack addresses:

(1) On every function call, a component is added to the address. Hence, the address size is linear in stack depth.
(2) Every function call must be augmented by symbolic information required to compute stack addresses.

In addition, the above scheme can still lead to inferior correspondence between checkpoints in different traces: in Anglican and other probabilistic languages where distributions are first-class objects checkpoints with incompatible arguments can correspond to the same stack address. Consider the following program fragment:

```
(let [dist (if use-gamma (gamma 2. 2.) (normal 0. 1.))]
  (sample dist))
```

The `sample` checkpoint has the same stack address in different traces. However, the random values should not be reused between different distributions. Further on, in some algorithms, such as black-box variational inference, the role of checkpoint addresses is semantic rather than heuristic — appropriate correspondence must be established between checkpoints in different traces for the algorithm to work.

To overcome the above problems, Anglican introduces a new addressing scheme which is almost as efficient as the scheme based on stack addresses for reuse of previously drawn values, while producing addresses of constant size, and allows manual computation of checkpoint addresses at runtime when the default automatic scheme is insufficient. According to the scheme:

- A checkpoint may accept an auxiliary argument — the checkpoint identifier. If specified, the identifier is the first argument of a checkpoint. For example `(sample 'x1 (normal 0 1))` defines a `sample` checkpoint with identifier `x1`.
- If the identifier is omitted, a unique identifier is generated automatically as a fresh symbol.
- At runtime, the address of a checkpoint invocation has the form `[`*checkpoint-identifier number-of-previous-occurrences*`]`, where the occurrences are of a checkpoint with the same checkpoint identifier.
- If a sequence of invocations of the same checkpoint is interrupted by a different checkpoint, the number of previous occurrences is rounded up to a multiple of an integer. For efficiency, a small power of 2 is used, such as $2^4 = 16$.

Consider the following simplified Anglican query:

```
(defm foo []
  (if (sample 'C1 (flip 0.5)) (foo) (bar)))

(defm bar []
  (case (sample 'C2 (uniform-discrete 0 3))
    0 (bar)
    1 (foo)
    2 (sample 'C3 (normal 0. 1.))))

(defquery baz (foo))
```

An execution of the query may result in the following sequence of checkpoint invocations:

```
  (sample 'C1 ...)
  (sample 'C2 ...)
  (sample 'C2 ...)
  (sample 'C1 ...)
  (sample 'C1 ...)
  (sample 'C1 ...)
  (sample 'C2 ...)
```

```
(sample 'C3 ...)
```

According to the addressing scheme, the addresses generated for these invocations are

```
[C1 0][C2 0][C2 1][C1 16][C1 17][C1 18][C2 16][C3 0]
```

If the program is run by a Metropolis-Hastings algorithm, then a small change usually takes place in the sequence of checkpoints with each invocation, and the new sequence may become

```
(sample 'C1 ...)
(sample 'C2 ...)
(sample 'C1 ...)
(sample 'C1 ...)
(sample 'C2 ...)
(sample 'C2 ...)
(sample 'C3 ...)
```

The addresses for the new sequence are

```
[C1 0][C2 0][C1 16][C1 17][C2 16][C2 17][C3 0]
```

Despite the change, the correspondence between checkpoints of similar types occurring in similar positions (relative positions in contiguous subsequences of a certain type) is preserved, and drawn values can be reused efficiently:

old	**new**
[C1 0]	[C1 0]
[C2 0]	[C2 0]
[C2 1]	*unused*
[C1 16]	[C1 16]
[C1 17]	[C1 17]
[C1 18]	*unused*
[C2 16]	[C2 16]
missing	[C2 17] *drawn*
[C3 0]	[C3 0]

Note that correspondence between checkpoints in different traces plays the role of an heuristic in Metropolis-Hastings family of algorithms. Any correspondence (or no correspondence, meaning all values must be re-drawn from their proposal distributions) is valid, and reused values from the previous invocation which are not in support or have zero probability in the new invocation are simply ignored and new values are re-drawn.

This way, each occurrence of a checkpoint has a unique address, but small disturbances — removal or addition of a single or just a few checkpoints — are unlikely to derail the entire sequence. The probability of derailment depends on the padding. The padding can be safely, and without any impact on performance, set to rather large numbers. However, rounding up to a multiple of 16 proved to be appropriate for all practical purposes.

9 DEFINITIONS AND RUNTIME LIBRARY

A Clojure namespace that includes a definition of an Anglican program imports ('requires') two essential namespaces: `anglican.emit` and `anglican.runtime`. The former provides macros for defining Anglican programs (`defquery`, `query`) and functions (`defm`, `fm`, `mem`), as well as Anglican bootstrap definitions that must be included with every program — first of all, CPS implementations of higher-order functions. `anglican.emit` can be viewed as the Anglican *compiler tool*, which helps transform Anglican code into Clojure before any inference is performed.

`anglican.runtime` is the Anglican runtime library. For convenience, it exposes common mathematical functions (`abs`, `floor`, `sin`, `log`, `exp`, etc.), but most importantly, it provides definitions of common distributions. Each distribution object implements a distribution interface with the `sample*` and `observe*` methods; this interface is defined using Clojure's protocol mechanism (`anglican.runtime/distribution`). The `sample*` *method* returns a random sample and roughly corresponds to the default implementation of the `sample` checkpoint. The `observe*` *method* returns the log probability of the value, which roughly corresponds to the default implementation of the `observe` checkpoint. The methods can be, and sometimes are called from handlers of the corresponding checkpoints, but do not have to be. For example, in LMH either the `sample*` or the `observe*` *method* is called for a `sample` checkpoint, depending on whether the value is drawn or reused.

The macro `defdist` is used to define distributions. `defdist` takes care of defining a separate type for every distribution so that Clojure multimethods (or overloaded methods) can be dispatched on distribution types when needed, e.g. for custom proposal distributions used in an inference algorithm. The Bernoulli distribution could be defined as follows:

```
(defdist bernoulli "Bernoulli distribution" [p]
  (sample* [this] (if (< (rand) p) 1 0))
  (observe* [this value]
    (Math/log (case value
                1 p
                0 (- 1. p)))))
```

In addition to distributions, Anglican provides *random processes*, which define sequences of random variables that are not independent and identically distributed. Random processes are similar to the so called 'exchangeable random primitives' in Church [6] and Venture [9]. However, random sequences generated by Anglican random processes are not required to be exchangeable. Random processes are defined using the `defproc` macro and implement the `anglican.runtime/random-process` protocol. This protocol has two methods

- `produce`, which returns the distribution on the next random variable in the sequence, and
- `absorb`, which incorporates the value for the next random variable and returns an updated random process.

As an example, here is a definition of a beta-Bernoulli process, in which each random variable is distributed according to a Bernoulli distribution with an unknown parameter that is drawn from a beta distribution:

```
(defproc beta-bernoulli "beta-Bernoulli process" [a b]
  (produce [this] (bernoulli (/ a (+ a b))))
  (absorb [this x]
    (case x
      0 (beta-bernoulli a (inc b))
      1 (beta-bernoulli (inc a) b))))
```

Unlike typical implementations of exchangeable random processes, Anglican's random processes do not have mutable state. The produce and absorb methods are deterministic and functional, and therefore do not have corresponding special forms in Anglican. A sequence of random values can be generated using a recursive loop in which absorb returns the updated process for the next iteration. For example:

```
(defm sample-beta-binomial [n a b]
  (loop [process (beta-bernoulli a b)
         values []]
    (if (= (count values) n)
      values
      (let [dist (produce process)
            value (sample dist)]
        (recur (absorb process value)
               (conj values value))))))
```

Similarly, random processes can also be used to recursively observe a sequence of values:

```
(defm observe-beta-bernoulli [values a b]
  (loop [process (beta-bernoulli a b)
         values values]
    (when (not-empty values)
      (let [dist (produce process)
            value (first values)]
        (observe dist value)
        (recur (absorb process value)
               (rest values))))))
```

10 PERFORMANCE

To justify the claim that Anglican is an *efficient* implementation, performance of Anglican programs must be evaluated both against Clojure programs and against inference tasks in other probabilistic programming languages and environments. Anglican passes two additional parameters to every function, adds extra code to variable bindings, and passes every function call through the trampoline. Hence difference in performance between Anglican and Clojure should be most noticeable on programs which involve many function calls. One such example is the famous *Towers of Hanoi*:

```
(defn towers-of-hanoi [n from to via]
  (when (not= n 1) ;; return nil if n = 1.
    (towers-of-hanoi (dec n) from via to)
    (towers-of-hanoi (dec n) via to from)))
```

We run this program for n equal to 10, 15, 20, and 25. Clojure is consistently only twice as fast as Anglican (Table 1); considering the amount of overhead Anglican introduces, this is a good result.

The other comparison must be made between Anglican and another probabilistic programming environment. WebPPL [7] is similar to Anglican in expressiveness. WebPPL runs on top of V8, a high performance JavaScript engine featuring a native compiler. Table 2 shows results for a few models from WebPPL distribution, as well as from Anglican test suite. For the comparison, we re-implemented WebPPL models in Anglican, and Anglican models in WebPPL, following as close as possible the original implementations, and run the same inference algorithm (called MCMC in WebPPL, LMH in Anglican) with the same number of iterations. Anglican outperforms WebPPL on all models. This can be attributed both to a better transformed representation of probabilistic programs and to more efficient architecture of the inference engine. The source code of the examples is available in the paper repository.

In addition, [16, pp. 32 – 33] provides a comparison of Anglican to an older, interpreted version of Anglican [31], and Probabilistic Scheme [14] on a rather complicated inference task of program synthesis. According to this comparison, Anglican is at least 10 times faster than the older interpreted version, and almost as fast as Probabilistic Scheme which uses inference engine implemented in C. In addition, [21, p. 171] compares Anglican to a DSL for probabilistic programming in Haskell and to Probabilistic C [14] on two simple probabilistic programs. Anglican shows similar performance to the Haskell implementation of Sequential Monte Carlo [2] and scales better with the number of particles. Given the simplicity of the programs used in the evaluation, this comparison is a confirmation that the flexible checkpoint and trampoline based interaction between an Anglican program and the inference engine does not introduce any noticeable overhead compared to more rigid designs.

11 CONCLUSION

In this paper, we presented design and implementation internals of the *probabilistic programming system* Anglican. Implementing a language is an interesting endeavour, in particular when the language implements a new paradigm, in this case probabilistic programming. Functional programming is a natural complement of probabilistic programming —

n	10	15	20	25
Anglican	0.63	3.00	75.9	2381.
Clojure	0.32	1.45	38.1	1245.

Table 1: Towers of Hanoi. Running times, in seconds, of Clojure and Anglican, averaged over 100 runs. Clojure is only twice as fast as Anglican.

model	**Anglican**	**WebPPL**
Latent Dirichlet allocation	22.2	31.5
Linear regression	6.1	8.5
Logistic regression	8.4	10.4
Multivariate regression	6.9	14.2
Simple branching	2.4	5.7
Hidden Markov model	8.7	10.2

Table 2: Anglican vs. WebPPL. Running times, in seconds, of Anglican and WebPPL, for 100 000 iterations of Markov Chain Monte Carlo (Lightweight Metropolis-Hastings), averaged over 100 runs. Anglican outperforms WebPPL on all models.

the latter allows both concise and expressive specification of probabilistic generative models and efficient implementation of inference algorithms.

Implementing a probabilistic language on top of and in tight integration with a functional language, Clojure, both helped us to accomplish an ambitious goal in a short time span, and provided important insights on structure and semantics of probabilistic concepts incorporated in Anglican. Computational efficiency and expressive power of Anglican owe to adherence to the functional approach as much as to rich inference opportunities of the Anglican environment.

ACKNOWLEDGEMENTS

This work was partially supported under DARPA PPAML through the U.S. AFRL under Cooperative Agreement number FA8750- 14-2-0006, Sub Award number 61160290-111668. Yang was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security).

REFERENCES

[1] A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 293–302.

[2] Nando de Freitas & Neil Gordon Arnaud Doucet (Ed.). 2001. *Sequential Monte Carlo Methods in Practice*. Springer.

[3] Henry G. Baker. 1995. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. *SIGPLAN Not.* 30, 9 (Sept. 1995), 17–20. DOI:http://dx.doi.org/10.1145/214448.214454

[4] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*.

[5] Hong Ge, Adam Ścibior, Kai Xu, and Zoubin Ghahramani. 2016. Turing: A fast imperative probabilistic programming language. (June 2016).

[6] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*.

[7] N. D. Goodman and A. Stuhlmüller. 2015. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org/ electronic; retrieved 2015/3/11.

[8] Rich Hickey. 2008. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*. ACM, New York, NY, USA, Article 1, 1 pages.

[9] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014).

[10] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic Models with Unknown Objects. In *Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press.

[11] T Minka, J Winn, J Guiver, and D Knowles. 2010. Infer .NET 2.4, Microsoft Research Cambridge. (2010).

[12] Lasse R. Nielsen. 2001. A Selective CPS Transformation. *Electr. Notes Theor. Comput. Sci.* 45 (2001), 311–331.

[13] Henrik Nilsson and Thomas A. Nielsen. 2014. Declarative Modelling for Bayesian Inference by Shallow Embedding. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT '14)*. ACM, New York, NY, USA, 39–42.

[14] Brooks Paige and Frank Wood. 2014. A compilation target for probabilistic programming languages. In *Proceedings of The 31st International Conference on Machine Learning*. 1935–1943.

[15] B. Paige, F. Wood, A. Doucet, and Y.W. Teh. 2014. Asynchronous Anytime Sequential Monte Carlo. In *Advances in Neural Information Processing Systems*.

[16] Yura N. Perov. 2016. Applications of Probabilistic Programming (Master's thesis, 2015). *CoRR* abs/1606.00075 (2016). http://arxiv.org/abs/1606.00075

[17] Yura N. Perov, Tuan Anh Le, and Frank D. Wood. 2015. Data-driven Sequential Monte Carlo in Probabilistic Programming. *CoRR* abs/1512.04387 (2015). http://arxiv.org/abs/1512.04387

[18] Avi Pfeffer. 2009. Figaro: An Object-Oriented Probabilistic Programming Language. In *Charles River Analytics Technical Report (2009)*.

[19] K. Pitman and K. Chapman. 1994. *Information Technology – Programming Language – Common Lisp*. Number 226-1194 in NCITS. ANSI.

[20] Tom Rainforth, Christian A Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem van de Meent, Arnaud Doucet, and Frank Wood. 2016. Interacting Particle Markov Chain Monte Carlo. In *Proceedings of the 33rd International Conference on Machine Learning (JMLR: W&CP)*, Vol. 48.

[21] Adam Scibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 165–176.

[22] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2010. *Revised [6] Report on the Algorithmic Language Scheme* (1st ed.). Cambridge University Press, New York, NY, USA.

[23] Stan Development Team. 2014. Stan: A C++ Library for Probability and Sampling, Version 2.4. (2014).

[24] S. Staton, H. Yang, C. Heunen, O. Kammar, and F. Wood. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Thirty-First Annual ACM/IEEE Symposium on Logic In Computer Science*.

[25] David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank Wood. 2015. Output-Sensitive Adaptive Metropolis-Hastings for Probabilistic Programs. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II*, Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, João Gama, Alípio Jorge, and Carlos Soares (Eds.). Springer International Publishing, Cham, 311–326.

[26] David Tolpin, Jan-Willem van de Meent, and Frank Wood. 2015. Probabilistic Programming in Anglican. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*, Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavalda, Dino Pedreschi, Francesco Bonchi, Jaime Cardoso, and Myra Spiliopoulou (Eds.). Springer International Publishing, Cham, 308–311.

[27] Jan-Willem van de Meent, Brooks Paige, David Tolpin, and Frank Wood. 2016. Black-Box Policy Search with Probabilistic Programs. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. 1195–1204.

[28] Jan-Willem van de Meent, Hongseok Yang, Vikash Mansinghka, and Frank Wood. 2015. Particle Gibbs with Ancestor Sampling for Probabilistic Programs. In *Artificial Intelligence and Statistics*. arXiv:1501.06769 http://arxiv.org/abs/1501.06769

[29] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proc. of the 14th Artificial Intelligence and Statistics*.

[30] David Wingate and Theophane Weber. 2013. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299* (2013).

[31] Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics*.

[32] Frank Wood, Jan-Willem van de Meent, David Tolpin, Brooks Paige, Hongseok Yang, Tuan Anh Le, and Yura Perov. 2014. The Probabilistic Programming System Anglican. http://robots.ox.ac.uk/~fwood/anglican/index.html. (2014). Accessed: 2016-06-30.

[33] Lingfeng Yang, Pat Hanrahan, and Noah D Goodman. 2014. Generating Efficient MCMC Kernels from Probabilistic Programs. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. 1068–1076.