Design and Implementation of Probabilistic Programming Language Anglican

David Tolpin Jan Willem van de Meent Hongseok Yang Frank Wood
University of Oxford
dtolpin@robots.ox.ac.uk jwvdm@robots.ox.ac.uk hongseok.yang@cs.ox.ac.uk 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 modeling concepts are mapped naturally to the functional foundation.

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 (Goodman et al. 2008; Mansinghka et al. 2014; Wood et al. 2014; Goodman and Stuhlmüller 2015) 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 backend, such as Metropolis-Hastings (Wingate et al. 2011; Mansinghka et al. 2014; Yang et al. 2014), Hamiltonian Monte Carlo (Team), expectation propagation (Minka et al. 2010), and extensions of Sequential Monte Carlo (Wood et al. 2014; van de Meent et al. 2015; Paige et al. 2014) methods. These generic techniques may be less statistically efficient than techniques tailored to a specific model. However, probabilistic programming facilitates simpler implementations of models making the inference inherently faster.

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 (Tolpin et al. 2015b; Wood et al.), a probabilistic programming language that tightly integrates with Clojure (Hickey 2008; Clo), 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 programming language on top of or besides another language. The easiest is an interpreter — a program that reads a program, in its entirety or line-by-line, and executes it by applying operational semantics of a certain kind to the language. Basic is famous for line-by-line interpreted implementations.

Another approach is to write a compiler, either to a virtual architecture, so called p-code or byte-code, or to real hardware. Here, the whole program is translated from the 'higher-level' source language to a 'lower-level' object language, which can be directly executed, either by hardware or by an interpreter — but the latter interpreter can be made simpler and more efficient than an interpreter for the source language.

On top of these two approaches are methods in which a new language is implemented 'inside' another language of the same level of abstraction. Different languages provide different means for this; Lisp is notorious 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.

An implementation of Anglican must therefore address three issues:

- the Clojure syntax to introduce probabilistic Anglican code inside Clojure modules;
- source-to-source transformation of Anglican programs into Clojure, so that probabilistic execution becomes possible;
- algorithms which run Clojure code, obtained by transforming Anglican programs, according to the probabilistic operational semantics.

Execution of probabilistic programs by inference algorithms is different from execution of deterministic programs. A probabilistic program is executed multiple times, often hundreds of thousands or even millions of times for a single inference task. Random choices may affect which parts of the program code are executed and how often. Many inference algorithms require re-running the program multiple times partially, from a certain point on. Different executions may employ different random choices. However, for efficient inference a correspondence between random choices in different executions should be maintained. These are just some of the challenges which were faced and solved during development of Anglican.

Comparisons of Anglican with other implementations of probabilistic programming languages (Scibior et al. 2015)(Perov 2016, 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.

2. 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 data
  "chooses a distribution
   which describes the 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',
        ;; and store the rest of observations in
        ;; 'observations*'.
        (let [[o & observations*] observations]
          ;; Observe 'o' from the guessed
          ;; distribution 'd'.
          (observe d o))
        ;; Proceed to the next iteration with
        ;; the rest of observations.
        (recur 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, $\texttt{d} \in \{\texttt{normal}, \texttt{gamma}\}$ 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 adapted distribution (also called posterior distribution) can be obtained by running the query under one of Anglican's inference algorithms.

Internally, an Anglican query is represented by a computation in *continuation passing style* (CPS) (Appel and Jim 1989), and inference algorithms exploit the CPS structure of the code to intercept probabilistic operations in an algorithm-specific way[1]. Among the available inference algorithms there are Particle Cascade (Paige et al. 2014), Lightweight Metropolis-Hastings (Wingate et al. 2011), Iterative Conditional Sequential Monte-Carlo (Particle Gibbs) (Wood et al. 2014), and others. Inference on Anglican queries generates a lazy sequence of samples, which can be processed asynchronously in Clojure code for analysis, integration, and decision making.

Clojure (and Anglican) runs on the JVM and gets access to a wide choice 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 JVM is available.

A probabilistic program, or query, mostly runs deterministic code, except for certain checkpoints, in which probabilities are involved, and normal, linear execution of the pro-

[1] (Goodman and Stuhlmüller 2015) also describe a CPS-based implementation of a probabilistic programming language.

gram is disrupted. In Anglican and similar languages there are two types of such checkpoints:

- drawing a value from a random source (`sample`);
- conditioning a computed value on a random source (`observe`).

Anglican can be mostly implemented as a regular programming language, except for the handling of these checkpoints. Depending on the *inference algorithm*, `sample` and `observe` may result in implicit input/output operations and control changes. For example, `observe` in particle filtering inference algorithms (Wood et al. 2014) is a nondeterministic control statement at which a particle (corresponding to a thread executing a program) can be either replicated or terminated. Similarly, in Metropolis-Hastings (Wingate et al. 2011), `sample` is both an input statement which 'reads' values from a random source, and a nondeterministic control statement (with delayed effect), eventually affecting acceptance or rejection of a sample.

Because of the checkpoints, Anglican programs must allow the inference algorithm to step in, recording information and affecting control flow. This can be implemented through coroutines/cooperative multitasking, and parallel execution/preemptive multitasking, as well as through explicit maintenance of program continuations at checkpoints. Anglican follows the latter option. Clojure is a functional language, and continuation-passing style (CPS) transformation is a well-developed technique in the area of functional languages. Implementing a variant of CPS transformation seemed to be the most flexible and lightweight option — any other form of concurrency would put a higher burden on the underlying runtime (JVM) and the operating system. Consequently, Anglican has been implemented as a CPS-transformed computation with access to continuations in probabilistic checkpoints. Anglican 'compiler', represented by a set of functions in the `anglican.trap` namespace, 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.

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 (defined in the `anglican.emit` namespace), 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 inference algorithms through the definitions of *probabilistic functions* using `defm` (Anglican counterparts 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.

Operational semantics of Anglican queries is different from that of Clojure code, therefore queries must be called through inference algorithms, rather than 'directly'. The `anglican.inference` namespace declares the (ad-hoc) polymorphic function `infer` using Clojure's multimethod mechanism. This function accepts an Anglican query and returns a lazy sequence of weighted samples from the distribution defined by the query. When inference is performed on an Anglican query, the query is run by a particular inference algorithm. Inference algorithms must provide an implementation for `infer`, as well as 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 `anglican.runtime` namespace, 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.

3. Language

The Anglican language is a subset of Clojure[2]. Anglican queries are defined within `defquery` as shown in the previous section, using `if`, `when`, `cond`, `case`, `let`, `and`, `or`, `fn` forms; other forms of Clojure may be supported in the future but are not now. In `let` bindings and `fn` argument lists, Clojure's pattern matching mechanism for vector data type (called vector destructuring) is supported. Also, compound literals for vectors, hash maps, and sets are supported just like in Clojure.

3.1 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`.

3.2 Tail Call

Clojure provides special forms `loop` and `recur` for writing tail-recursive programs. Such forms are not necessary in Anglican, because Anglican programs are CPS-converted and do not use the stack. For instance, no recursive call in Anglican can lead to stack overflow. In fact, in Anglican, it is recommended to use recursive calls to functions instead of `recur`. However, `loop`/`recur` is provided in Anglican for convenience as a way to express loops. `recur` outside of `loop` will lead to unpredictable behaviour and hard-to-catch errors.

4. Macro-based Compilation

Compilation of Anglican into Clojure is built around a variant of CPS transformation. In a basic CPS-transformed program, a continuation receives a single argument — the computed value. In Anglican, there are two flows of computation going in parallel: values are computed by functional code, and, at the same time, the state of the probabilistic

[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 (Sperber et al. 2010), or Common Lisp (Pitman and Chapman 1994), the result of macro-expansion of derived special forms is not well specified and implementation specific.

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* in namespace `anglican.trap`, which are 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)
```

4.1 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,
   :result nil,
   ::mem {},
   ::store nil,
   ... })
```

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. Except for transformation of the `mem` form (which converts a function to one with memoization), CPS transformation routines are not aware of contents of `$state`, do not access or modify it directly, but rather just thread the state unmodified through the computation. Algorithm-specific handlers of checkpoints corresponding to the probabilistic forms (`sample`, `observe`) modify the state and reinject a new state into the computation.

4.2 Expression Kinds

There are three different kinds of inputs to CPS transformation:

- Literals, which are constant expressions. They are passed as an argument to the continuation unmodified.
- Value expressions such as the `fn` form (called *opaque* expressions in the code). They must be transformed to CPS, but the transformed object is passed to the continuation as a whole, opaquely.
- General expressions (which we call *transparent* expressions). The continuation is threaded through such an expression in an expression-specific way, and can be called in multiple locations of the CPS-transformed code, such as in all branches of an `if` statement.

4.2.1 Literals

Literals are the same in Anglican and Clojure. They are left unmodified; literals are a subset of opaque expressions. However, the Clojure syntax has a peculiarity of using the syntax of compound literals (vectors, hash maps, and sets) for data constructors. Hence, compound literals must be traversed recursively, and if there is a nested non-literal component, transformed into a call to the corresponding data constructor. Functions `cps-of-vector`, `cps-of-hash-map`, `cps-of-set`, called from `cps-of-expression`, transform Clojure constructor syntax (`[...]`, `{...}`, `#{...}`) into the corresponding calls:

```
=> (cps-of-vector [0 1 2] 'cont)
(cont (vector 0 1 2) $state)
=> (cps-of-hash-map {:a 1 :b 2} 'cont)
(cont (hash-map :a 1, :b 2) $state)
=> (cps-of-set #{0 1} 'cont)
(cont (set (list 0 1)) $state)
```

4.2.2 Opaque Expressions

Opaque, or value, expressions, have a different shape in the original and the CPS form. However, their CPS form follows the pattern `(continuation transformed-expression $state)`, and thus the transformation does not depend on the continuation parameter, and can be accomplished without passing the parameter as a transformation argument. Primitive (non-CPS) procedures used in Anglican code, `(fn ...)` forms, and (mem ...) forms are opaque and transformed by `primitive-procedure-cps`, `fn-cps`, and `mem-cps`, correspondingly: a slightly simplified CPS form of expression

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

would be

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

In the actual code an automatically generated fresh symbol is used instead of `cont`.

4.2.3 General Expressions

The most general form of CPS transformation receives an expression and a continuation as parameters, and returns the expression in CPS form with the continuation parameter potentially called in multiple tail positions. General expressions can be somewhat voluntarily divided into several groups:

- binding forms — `let` and `loop/recur`;
- flow control — `if`, `when`, `cond`, `case`, `and`, `or` and `do`;
- function applications and `apply`;
- probabilistic forms — `observe`, `sample`, `store`, and `retrieve`.

Functions that transform general expressions accept the expression and the continuation as parameters, and are consistently named `cps-of-`*form*, for example, `cps-of-do`, `cps-of-store`.

4.3 Implementation Highlights

So far we introduced the basics of Anglican compilation to Clojure. The described approaches and techniques are important for grasping the language implementation but relatively well-known. In the rest of the section we focus on challenges we met and resolved while implementing Anglican, as

well as on implementation of unique features of Anglican as a probabilistic programming language.

4.3.1 Continuations

Continuations are functions that are called in tail positions with the computed value and state as their arguments — in CPS there is always a function call in every tail position and never a value. Continuations are passed to CPS transformers, and when transformers are called recursively, the continuations are generated on the fly.

There are two critical issues related to generation of continuations:

- unbounded *stack growth* in recursive code;
- code size *explosion* when a non-atomic continuation is symbolically substituted in multiple locations.

Managing stack size 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* — 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.

Avoiding exponential code growth To realize potential danger of code size explosion, consider CPS transformation of code

```
(if (adult? person)
  (if (male? person)
     (choose-beer)
     (choose-wine))
  (choose-juice))
```

with continuation

```
(fn [choice _]
  (case (kind choice)
    :beer (beer-jar choice)
    :wine (wine-glass choice)
    :juice (juice-bottle choice)))
```

During CPS transformation, if we substitute the code of this continuation for all of its calls, the code will be repeated three times in the CPS-transformed expression. In general, CPS code can grow extremely large if the code of continuations is substituted repeatedly.

To circumvent this inefficiency, CPS transformers for expressions with multiple continuation points (`if` and derivatives, `and`, `or`, and `case`) bind the continuation to a fresh symbol if it is not yet a symbol. Macro `defn-with-named-cont` establishes the binding automatically:

```
=> (cps-of-if '(c t f) '(fn [x] (* 2 x)))
(let [cont (fn [x] (* 2 x))]
  (if c
      (fn [] (cont t $state))
      (fn [] (cont f $state))))
```

4.3.2 Primitive Procedures

When an Anglican function is transformed into a Clojure function by `fn-cps`, two auxiliary parameters are added to the beginning of the parameter list — continuation and state. Correspondingly, when a function *call* is transformed (by `cps-of-application` or `cps-of-apply`), 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 be able to recognize 'primitive' (that is, implemented in Clojure rather than in Anglican) functions.

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 a 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 (`fn-cps`, `cps-of-let`). Macro `shading-primitive-procedures` automates the shading.

4.3.3 Probabilistic Forms

There are two proper probabilistic forms turning Anglican into a probabilistic programming language — `sample` and `observe`. Their purpose is to interrupt deterministic computation and transfer control to the inference algorithm. Practically, this is achieved through returning **checkpoints** — Clojure values of the corresponding types (`anglican.trap.sample` or `anglican.trap.observe`). The types contain fields specific to each form, as well as the

continuation; calling the continuation resumes the computation. Checkpoints expose the program state to the inference algorithm, and the updated state is re-injected into the computation when the continuation is called:

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

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

In addition to checkpoints, there are a few other special forms — `store`, `retrieve`, `mem` — which modify program state. These forms are translated into expressions involving calls of functions from the `anglican.state` namespace. The `mem` form, which implements memoization, deserves a more detailed explanation.

Memoization The author of a probabilistic model might want to randomly draw a feature value of 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, randon 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 return the same output.

Memoization is often implemented on top of a mutable dictionary, where the key is the argument list 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 arguments. 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.

5. 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. Computing the distribution is facilitated by an 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 (Tolpin et al. 2015a; van de Meent et al. 2016; Rainforth et al. 2016), 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 in the `anglican.inference` namespace. 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.

5.1 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-Hasting 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.

5.2 Internals of an Inference Algorithm

The simplest inference algorithm is *importance sampling*:

```
(derive ::algorithm
        :anglican.inference/algorithm)
(defmethod infer :importance [_ prog value & {}]
  (letfn
    [(sample-seq []
       (lazy-seq
         (cons
           (:state (exec ::algorithm prog value
                         initial-state))
           (sample-seq))))]
    (sample-seq)))
```

`anglican.importance/infer` just calls `anglican.inference/exec` and relies on default implementations of checkpoint handlers. A different inference algorithm would provide its own implementations of `checkpoint` for `sample`, `observe`, or both, as well as invoke `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 (program instances) simultaneously, while LMH re-runs programs from an intermediate continuation rather than from the beginning.

5.3 Addressing of Checkpoints

Many inference algorithms memoize and reuse earlier computations at checkpoints: variants of Metropolis-Hastings reuse previously drawn values (Wingate et al. 2011), as well as additional information used for adaptive sampling (Tolpin et al. 2015a) at `sample` checkpoints, Asynchronous SMC (Particle Cascade) computes average particle weights at `observe` checkpoints (Paige et al. 2014). Implementations of black-box variational inference (Wingate and Weber 2013; van de Meent et al. 2016) 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 time 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 (Wingate et al. 2011). 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 re-used 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 uses a different scheme which is almost as efficient as the scheme based on stack addresses for re-use of previously drawn values, as produces 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 this 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 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.

Function `checkpoint-id` in the `anglican.inference` namespace automates generation of checkpoint addresses and can be used from any inference algorithm.

6. 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 the `anglican.runtime/distribution` protocol, with two methods: `sample*` and `observe*`. The `sample*` *method* returns a random sample and roughly corresponds to the `sample` checkpoint, the `observe*` *method* returns the log probability of the value and roughly corresponds to 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 re-used.

Macro `defdist` should be used to define distributions. `defdist` takes care of defining a separate type for every distribution so that multimethods can be dispatched on

distribution types when needed, e.g. for custom proposals. 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)
                0.))))
```

Similar to distributions, *random processes*, related to so called 'exchangeable random procedures'[3], are defined using macro `defproc` and implement the `anglican.runtime/random-process` protocol. The protocol has two methods — `produce`, which returns a distribution object, and `absorb`, which returns a new random process object obtained by absorbing a value drawn from the process. Here is a definition of the Poisson process:

```
(defproc PP
  "Poisson point process"
  [rate] [dist (exponential rate)]
  (produce [this] dist)
  (absorb [this value] this))
```

While distributions are random and non-functional, random processes are deterministic and functional, hence `produce` and `absorb` are called directly and do not have corresponding special forms in Anglican:

- Values produced by the process are sampled from the distribution returned by `produce`.
- When a value is observed from the distribution, a new process, with the observed value absorbed, is returned by `absorb`.

A common pattern of programming with processes in Anglican is recursively computing a new process based on the observed value:

```
(loop [process (HawkesPointProcess)
       [value & values] values]
  (let [dist (produce process)]
    (observe dist value)
    (recur (absorb process value) values)))
```

Unlike conventional implementations of random processes, Anglican's random processes do not have mutable state.

7. Case Study

The description of Anglican language and environment would be incomplete without a case study showing the process of building and executing a solution of an inference problem. This case study takes a problem for which a solution is not immediately obvious, *the Deli dilemma*, and guides through writing an Anglican program for the problem, executing the program, and post-processing results.

The program presented in this section is intentionally short and simple. Aglican is capable of compiling and running elaborated programs handling large amounts of data. Advanced examples of Anglican programs and inference can be found in literature on applications of probabilistic programming (Perov et al. 2015; Perov 2016; van de Meent et al. 2016).

[3] However, random sequences generated by Anglican random processes are not required to be exchangeable.

7.1 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 sandwitch 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?

In addition to the story above, 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, were 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.
- The waiter's odds that this is the same customer are 2 to 1.

7.2 Anglican Query

We want to formalize the dilemma in 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 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, one 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 for `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-customer)]
    {:is-same-customer is-same-customer
     :times-to-arrive (observe-customer
                        time-to-arrive-prior
                        lunch-delay
                        dinner-delay)}))
```

Performing inference on 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.

7.3 Inference

Having defined the query, we are now ready to run the query using an inference algorithm. Function `doquery` provided by the `anglican.core` namespace 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). The algorithm is somewhat slow to converge but can be used with any Anglican query, and should be robust enough for our simple problem.

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.

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

To approximate the inferred distribution, we extract a finite subsequence of samples. Many algorithms use an initial subsequence of samples to converge to the target distribution, hence we drop initial N samples (N is 5000 in the code snippet below), and collect the next N 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)))
```

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

In addition to 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)

```
;; single customer
(def time-to-arrive+
     (map (comp first :times-to-arrive)
          (filter :same-customer results)))
(def mean-to-arrive+ (mean time-to-arrive+))
(def sd-to-arrive+ (std time-to-arrive+))

;; two customers
(def times-to-arrive+
     (map :times-to-arrive
          (filter (complement :same-customer)
                  results)))
(def mean-1-to-arrive+
     (mean (map first times-to-arrive+)))
(def sd-1-to-arrive+
     (std (map first times-to-arrive+)))
(def mean-2-to-arrive+
     (mean (map second times-to-arrive+)))
(def sd-2-to-arrive+
     (std (map second times-to-arrive+)))
```

In addition to plotting the distribution histograms, we use functions `mean`, `std` provided, along with other useful statistical functions, in the `anglican.stat` namespace. This

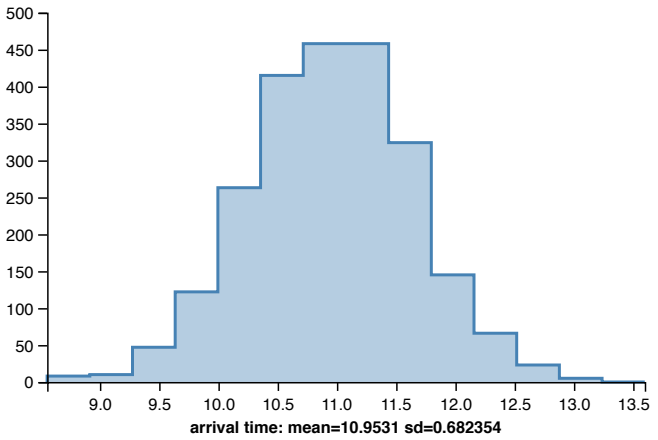

Figure 1. Arrival time distribution for a single customer.

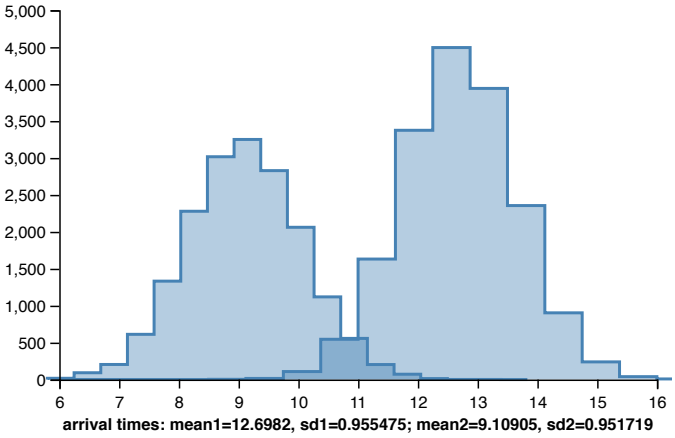

Figure 2. Arrival time distributions for two customers.

is another illustration of advantage of tight integration between Clojure and Anglican — probabilistic models are expressed in Anglican. However, processing of data and results can rely on the full power of Clojure.

For inference, we chose to use Lightweight Metropolis-Hastings, perhaps somewhat voluntarily. A strength of probabilistic programming is that models are separated from inference. To switch to a different inference algorithm we just need to pass a different value to `doquery`. For example, we may decide to use Black-Box Variational Bayes (BBVB), which may not work equally well for all probabilistic programs, but is much faster to converge.

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

We can still use samples and summary statistics with BBVB to approximate the posterior distribution. However, variational inference approximates the posterior by known distributions, and we can directly retrieve the distribution parameters.

```
(clojure.pprint/pprint
  (anglican.bbvb/get-variational (nth samples N)))
```

```
{S28209
 {(0 anglican.runtime.normal-distribution)
  {:mean 10.99753360180663,
   :sd 0.7290976433082352}},
 S28217
 {(0 anglican.runtime.normal-distribution)
  {:mean 12.668050292254202,
   :sd 0.9446695174790353}},
 S28215
 {(0 anglican.runtime.normal-distribution)
  {:mean 9.104132559955836,
   :sd 0.9479290526821788}},
 S28219
 {(0 anglican.runtime.flip-distribution)
  {:p 0.11671977016875924,
   :dist
   {:min 0.0,
    :max 1.0}}}}
```

We can guess that the variational distributions correspond to the prior distributions in the `sample` forms, in the order of appearance. However, it would help if we could use more informative labels instead of automatically generated symbols `S28209`, `S28217`, etc. Here the option to specify identifiers for probabilistic forms explicitly comes handy. If we modify the `sample` forms to use explicit identifiers (only the forms are shown for brevity), the output becomes much easier to analyse.

```
          (sample :arrival-time-same
                  time-to-arrive-prior)]

          (sample :arrival-time-first
                  time-to-arrive-prior)

          (sample :arrival-time-second
                  time-to-arrive-prior)]

  (let [is-same-customer (sample :same-or-different
                                 (flip p-same))
```

The output becomes much easier to interpret and analyse programmatically.

```
{:arrivate-time-same
 {(0 anglican.runtime.normal-distribution)
  {:mean 10.99753360180663,
   :sd 0.7290976433082352}},
 :arrival-time-first
 {(0 anglican.runtime.normal-distribution)
  {:mean 12.668050292254202,
   :sd 0.9446695174790353}},
 :arrival-time-second
 {(0 anglican.runtime.normal-distribution)
  {:mean 9.104132559955836,
   :sd 0.9479290526821788}},
 :same-or-different
 {(0 anglican.runtime.flip-distribution)
  {:p 0.11671977016875924,
   :dist
   {:min 0.0,
    :max 1.0}}}}
```

This completes the case study, where we showed a probabilistic programming solution of a problem, implemented

and analysed in Anglican and Clojure, using two different inference algorithms.

8. Summary

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

Implementing a probabilistic language on top 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.

Acknowledgments

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.

References

Clojure. `http://clojure.org`. Accessed: 2016-06-30.

A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2.

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

N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

R. Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2.

V. K. Mansinghka, D. Selsam, and Y. N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

B. Paige, F. Wood, A. Doucet, and Y. Teh. Asynchronous anytime sequential Monte Carlo. In *Advances in Neural Information Processing Systems*, 2014.

Y. N. Perov. Applications of probabilistic programming (master's thesis, 2015). *CoRR*, abs/1606.00075, 2016. URL `http://arxiv.org/abs/1606.00075`.

Y. N. Perov, T. A. Le, and F. D. Wood. Data-driven sequential Monte Carlo in probabilistic programming. *CoRR*, abs/1512.04387, 2015. URL `http://arxiv.org/abs/1512.04387`.

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

T. Rainforth, C. A. Naesseth, F. Lindsten, B. Paige, J.-W. van de Meent, A. Doucet, and F. Wood. Interacting particle Markov chain Monte Carlo. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR: W&CP*, 2016.

A. Scibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 165–176, 2015.

M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521193990, 9780521193993.

S. D. Team. Stan: A C++ library for probability and sampling, version 2.4.

D. Tolpin, J. W. van de Meent, B. Paige, and F. Wood. Output-sensitive adaptive Metropolis-Hastings for probabilistic programs. In A. Appice, P. P. Rodrigues, V. Santos Costa, J. Gama, A. Jorge, and C. Soares, editors, *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II*, pages 311–326. Springer International Publishing, Cham, 2015a.

D. Tolpin, J. W. van de Meent, and F. Wood. Probabilistic programming in anglican. In A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, and M. Spiliopoulou, editors, *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*, pages 308–311. Springer International Publishing, Cham, 2015b. ISBN 978-3-319-23461-8.

J. W. van de Meent, H. Yang, V. Mansinghka, and F. Wood. Particle Gibbs with ancestor sampling for probabilistic programs. In *Artificial Intelligence and Statistics*, 2015. URL `http://arxiv.org/abs/1501.06769`.

J. W. van de Meent, B. Paige, D. Tolpin, and F. Wood. 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*, pages 1195–1204, 2016.

D. Wingate and T. Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. of the 14th Artificial Intelligence and Statistics*, 2011.

F. Wood, J. W. van de Meent, D. Tolpin, B. Paige, H. Yang, T. A. Le, and Y. Perov. The probabilistic programming system Anglican. `http://robots.ox.ac.uk/~fwood/anglican/index.html`. Accessed: 2016-06-30.

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, 2014.

L. Yang, P. Hanrahan, and N. D. Goodman. Generating efficient MCMC kernels from probabilistic programs. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 1068–1076, 2014.