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 describe the implementation of Anglican and illustrate how its design facilitates both explorative and industrial use of probabilistic programming.

In these notes, I explain the rationale behind the macro-compiled implementation of Anglican, outline design choices, and describe important implementation aspects.

1. Introduction

For data science practitioners, statistical inference is typically but 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 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 (Stan Development Team 2014), 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. Although these generic techniques are not always as statistically efficient as techniques that take advantage of model-specific optimizations, probabilistic programming makes it easier to optimize models for a specific application in a manner that is efficient in terms of the dimensionality of its latent variables.

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 demonstration, we describe the implementation of Anglican (`http://www.robots.ox.ac.uk/~fwood/anglican/`), a probabilistic programming language that tightly integrates with Clojure (`http://clojure.org/`), 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.

Anglican is a probabilistic programming language integrated with Clojure. There are several ways to build a programming language on top or besides another language. The easiest to grasp 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.

The following sections explain the way Anglican is implemented, from source-to-source transformation and syntactic wrappers to inference algorithms which accept Clojure functions built from Anglican code as a parameter, and produce probabilistic outputs.

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 return values, or *predicts*. 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:

Internally, an Anglican query is represented by a computation in *continuation passing style* (CPS), and inference algorithms exploit the CPS structure of the code to intercept probabilistic operations in an algorithm-specific way. 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) run on the JVM and get 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 program is disrupted. In Anglican and similar languages there are two types of such checkpoints:

- drawing a value from a random source (`sample`);
- conditioning the posterior distribution by 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 is a non-deterministic control statement at which a particle can be either replicated or terminated. Similarly, in Metropolis-Hastings, `sample` is both an input and a non-deterministic 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, parallel execution/preemptive multitasking and shared memory, as well as through explicit maintenance of program continuations at checkpoints. 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 objects suitable for passing as arguments to inference algorithms (`defquery`, `query`). In addition to defining entire queries, Anglican promotes modularization of inference algorithms through definition of *probabilistic functions* using `defm` and `fm` (Anglican counterparts of Clojure `defn` and `fn`). Probabilistic functions are written in Anglican, may include probabilistic forms `sample` and `observe` (as well `predict` for the output), 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 supplies the `infer` multimethod, which 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 some of the methods of the `checkpoint` multimethod, called to handle `sample` and `observe` in an algorithmic-specific manner, as well as on 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 additional distributions can be defined by the user by implementing the `distribution` protocol. The `defdist` macro provides a convenient syntax for defining primitive distributions.

Compilation, invocation, and runtime support of Anglican queries are discussed in detail in further sections.

3. Macro-based Compilation

Compilation of Anglican into Clojure relies on the Clojure *macro* facility. However, the compilation algorithm is implemented as a library of *functions* in namespace `anglican.trap`, which are invoked by macros. Namespace

anglican.trap-test contains many transformation tests, which both help assure proper functioning of the transformation functions, and serve as illustrative examples of individual transformations, such as of functions, flow control, and probabilistic contstructs.

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, with the computed result passed to the continuation. A continuation accepts two arguments:

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

The state ($state) is threaded through the computation and contains data used by inference algorithm. $state is a Clojure hash map, and the map entries are algorithm-dependent. Except for transformation of mem, the memoization form, 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.

3.1 Expression Kinds

There are three different kinds of inputs to CPS transformation:

- literals, which are passed as an argument to the continuation unmodified;
- value expressions (e.g. the fn form) (called *opaque* expressions in the code) which must be transformed to CPS, but the transformed object is passed to the continuation as a whole, opaquely;
- general expressions (let's call them *transparent*), through which the continuation is threaded 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.

3.1.1 Literals

Literals are the same in Anglican and Clojure. They are left unmodified. Literals are a subset of opaque expressions, and are identified by test simple? called from opaque?. 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.

3.1.2 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), and thus the transformation does not depend on the continuation, and can be accomplished without passing the continuation 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.

3.1.3 General Expressions

The most general form of CPS transformation receives an expression and a continuation, and returns the expression in CPS form with the continuation 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 — predict, 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.

3.2 Implementation Highlights

3.2.1 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.

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.

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)))
```

The code of the continuation, represented by an fn form, will be repeated three times. In general, CPS code can grow extremely large if symbolic continuations are inserted 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.

3.2.2 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 a Clojure function. Global dynamically bound variables `*primitive-procedures*` and `*primitive-namespace*` 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 function inside the let block in the following example:

```
(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.

3.2.3 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 records of the corresponding types (`anglican.trap.sample` or `anglican.trap.observe`). The records 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.

In addition to checkpoints, there are a few other special forms — `predict`, `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 stochastic memoization, deserves a more detailed explanation.

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 (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 that first creates an updated state with the memoized result, and then calls the 'outer' continuation with the result and the updated state.

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.

4. Inference Algorithms

An inference algorithm is an implementation of `infer` multimethod from `anglican.inference` namespace. The multimethod 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.

4.1 The `infer` multimethod

The sole purpose of the algorithm identifier is method dispatch — conventionally, the identifier is a keyword related to the algorithm name (`:lmh` for Lightweight Metropolis-Hasting, `:pcascade` for Particle Cascade etc.). The second parameter is a query as defined by `query` or `defquery` forms. If needed, the query can be defined anonymously, right in the argument list of a call to `infer`:

```
(let [x 1]
  (infer :pgibbs (query (predict 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, however 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. The value is destructured in case of a structured binding, for example:

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

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

Finally, an indefinite 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.

4.2 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 runs re-runs programs from an intermediate continuation rather than from the beginning.

5. 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 names of the methods coincide with Anglican special forms `sample` and `observe`, but they are by no means the same. The `sample` *method* returns a random sample, the `observe` *method* returns the log probability of the value. 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 record type for every distribution so that multimethods can be dispatched on distribution types when needed, e.g. for custom proposals.

Similar to distributions, *random processes*, related to so called 'exchangeable random procedures', are defined using macro `defproc` and implement the `anglican.runtime/random-process` protocol. The protocol has twho 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. 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.

Acknowledgments

References

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.

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.

Stan Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4. 2014.

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`.

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, 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.