Quicksand: A Lightweight Implementation of Probabilistic Programming for Procedural Modeling and Design

Daniel Ritchie
Stanford University
dritchie@stanford.edu

1 Introduction

The past several years have seen the development of multiple probabilistic programming languages (PPLs) in the artificial intelligence community [9, 5, 12, 13, 7, 11]. In addition to their expressiveness, PPLs allow programmers to develop models modularly and independently from inference algorithms. Over the same period, computer graphics research has begun to demonstrate how probabilistic inference can enable complex procedural content creation [6, 14, 18]. But this work has relied on specialized models and inference algorithms, making the methods difficult for technical artists—who possess programming skills but not necessarily inference expertise—to adopt. Using probabilistic programming for procedural content creation has the potential to close this adoption gap.

Graphics applications demand high performance, so a procedural content design PPL should compile to efficient machine code. Tracing JIT compilers such as Shred can deliver high performance, but they suffer from excessive re-compilation overhead when programs exhibit extensive structure change (which procedural modeling programs typically do) [17]. Instead, we asked: why not embed probabilistic programming directly in a compiled, statically-typed, high-performance language?

Along these lines, we present Quicksand, a probabilistic programming system embedded in the Terra high-performance computing language [4]. We chose Terra as an embedding target because:

- Terra's multi-stage code generation simplified the embedding process—Quicksand is just a library in Terra, not a language extension.
- Terra is tightly integrated with Lua, a language already used for scripting in many game engines.

Other statically-typed PPLs exist, but do not meet our goals. Figaro uses Scala's static type system to assemble type-correct programs, but it incurs performance overhead by running on the JVM [13]. Probabilistic C compiles directly to machine code, but it only supports forward inference methods based on Sequential Monte Carlo [11]. In contrast, Quicksand produces efficient machine code and supports a variety of Markov Chain Monte Carlo methods via its lightweight embedding strategy [16]. This paper introduces the programming model and implementation strategies underlying Quicksand.

2 Programming in Quicksand

Figure 1 shows some example Quicksand programs. While Quicksand was motivated by procedural modeling and design, it is general enough to express classical machine learning problems as well. For instance, Figure 1 Left is an implementation of the Bayesian linear regression model from the

```
qs = terralib.require("qs")
-- Model definition
linreg = qs.program(function()
    return terra()
        -- The data
        var xs = array(0.0, 1.0, 2.0, 3.0)
        var ys = array(0.0, 1.0, 4.0, 6.0)
        -- Parameters
        var m = qs.gaussian(0.0, 2.0, {struc=false})
        var b = qs.gaussian(0.0, 2.0, {struc=false})
        var v = qs.gamma(1.0, 1.0, {struc=false})
        -- Condition on all the provided data
        for i=0,4 do
            qs.gaussian.observe(ys[i], m*xs[i]+b, v)
        end
        -- Predict the value at x=4
        return m*4.0 + b
    end
end)
-- Query the model for the MAP prediction
queryfn = qs.infer(linreg, qs.MAP,
                           qs.MCMC(qs.TraceMHKernel()))
print(queryfn())
-- output: 8.024
```

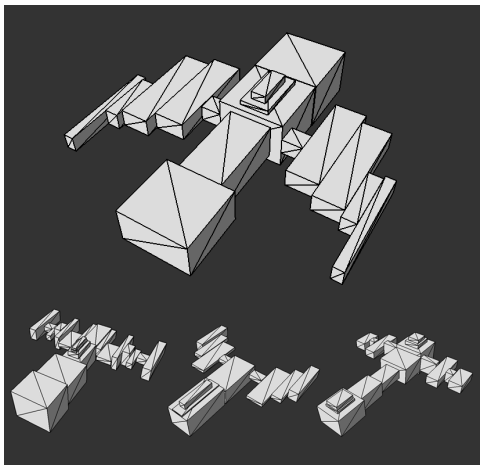

Figure 1: Quicksand example programs. *(Left)* Code for Bayesian linear regression. *(Right)* Procedurally-generated spaceships (see Appendix for code).

Forest generative model repository [1]. Quicksand programs are embedded in the low-level language Terra, which has similar semantics to C and similar syntax to Lua (the language used to metaprogram Terra). Figure 1 Right shows some screenshots from a procedural modeling application for generating spaceships. The user can input the desired aspect ratio of the generated ships; here, a square aspect ratio is specified. Model code for this example is shown the Appendix.

Using a low-level language introduces some limitations. In particular, there are no higher-order functions in Terra. Consequently, Quicksand does not have stochastic memoization (though memoized functions are possible, just not memoized closures). Quicksand also currently has no Bayesian non-parametrics, since these are typically implemented with stochastic memoization (though other implementation strategies are possible). In our experience, the lack of these features has not proved detrimental to procedural modeling ability.

A more complete guide to programming in Quicksand can be found online at `http://dritchie.github.io/quicksand`

3 Implementation

Quicksand follows the standard lightweight embedding strategy for PPLs [16]. It stores the values of random choices made during a program execution in a structurally-addressed `Trace` object. Random choice addresses are computed using Terra's hygienic code-generation macros, rather than in a separate transformational compilation pass. This approach allows Quicksand to exist as a Terra library.

Inference in Quicksand uses MCMC, which defaults to the typical single-site Metropolis-Hastings algorithm for PPLs (`qs.TraceMHKernel`) [16]. Quicksand is geared toward procedural modeling programs, which typically feature many continuous variables whose existence and structure is determined by a smaller set of discrete choices. Thus, Quicksand explicitly separates 'potentially structure-changing' from 'structure non-affecting' random choices (the `struc` tag) and provides several MH kernels which propose to all 'non-structural' choices at once. These include Gaussian drift, Hit-and-Run Monte Carlo [3], and Hamiltonian Monte Carlo [10] (which Terra's flexible operator overloading makes easy to implement succinctly and efficiently). By proposing to as many variables as possible, we also minimize the computational overhead of complete program re-execution that is intrinsic to all lightweight PPL embeddings. Quicksand also provides the Locally-Annealed Reversible Jump kernel for boosting the acceptance rate of structure-changing proposals that introduce multiple new continuous variables [18].

Quicksand programs use statically-determined type layout and function dispatch, so compiling them introduces some challenges. For example, compiling a program `P` requires compiling a trace object type `Trace(P)`, since the program code will include access to members of `Trace(P)`. However, compiling `Trace(P)` in turn requires compiling `P`, since (a) we must know the types of all random choices used in `P` to determine the layout of `Trace(P)` and (b) the `Trace(P):update` method (which propagates a change in random choices through the program) must call `P`. We break these cylical dependencies as follows:

1. Compile `P`, but do not generate code for random choices. Instead, record their types.
2. Begin compiling `P` again, generating random choice code. This requires the layout of `Trace(P)`, which can be finalized using the types recorded in Step 1.
3. Generate code for `Trace(P):update`. This code will reference the still-compiling `P`, but since Terra functions are JIT-compiled, it will not be compiled until it is first called (at which point `P` will have finished compiling).

4 Performance

Quicksand programs compile directly to efficient machine code, allowing them to run faster than previous lightweight PPL embeddings. Figure 2 shows a performance comparison of Quicksand against probabilistic-js, a state-of-the-art lightweight embedding in Javascript [2]. Both implementations were run for 100,000 MH iterations on three standard benchmark models from the Forest repository: Discrete-Time Hidden Markov Model (with ten states and five observations), Medical Diagnosis, and Bayesian Linear Regression [1]. Evaluations were performed on an Intel Core i7-3840QM with 16GB of RAM running OSX 10.8.5. In general, the more computation a model requires, the more speedup Quicksand achieves (as much as 5x in these examples). We also plan to compare Quicksand with Shred and Probabilistic C [11] to see how it fares against other low-level, compiled implementations, particularly in the presence of structure change.

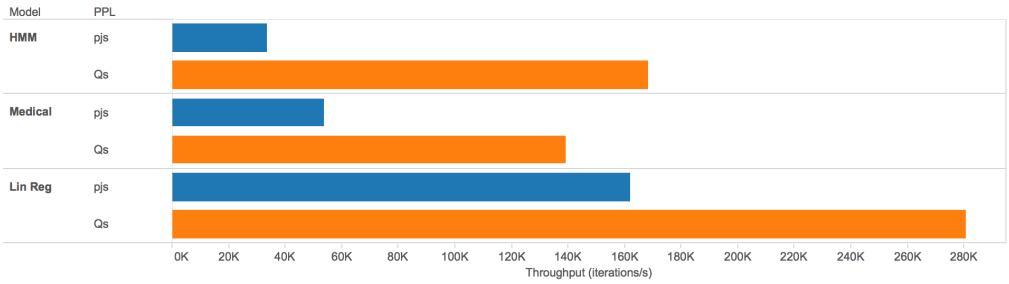

Figure 2: Throughput in MH iterations per second achieved by Quicksand (Qs) and probabilistic-js (pjs) when sampling from a discrete-time HMM (HMM), a medical diagnosis network (Medical), and a linear regression model (Lin Reg).

5 Future Work

Quicksand is still under active development, and there are several avenues for future work. First, Terra code can be easily compiled to run on CUDA-enabled GPUs, a capability which suggests a host of parallel inference algorithms, such as particle filters and parallel tempering. Parallel tempering in particular has been shown to enable near-interactive inference in some procedural design applications [8]. Parallelizing execution *within* a program could also be valuable, as demonstrated by the Augur system for data-parallel inference [15].

In addition, we would like to see Quicksand embedded in games that use Lua scripting. This could allow game developers and even end-users, given a small amount of extra training, to add complex procedurally-generated content to their virtual worlds.

Acknowledgments Quicksand is being developed as part of the DARPA Probabilistic Programming for Advanced Machine Learning (PPAML) program. This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Forest: a reposistory for generative models. http://forestdb.org/, 2014.

[2] probabilistic-js. https://github.com/dritchie/probabilistic-js, 2014.

[3] Claude J. P. Blisle, H. Edwin Romeijn, and Robert L. Smith. Hit-and-run algorithms for generating multivariate distributions. *Mathematics of Operations Research*, 18(2), 1993.

[4] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. In *Proc. PLDI 2013*, 2013.

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

[6] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. In *Proc. SIGGRAPH 2012*, 2012.

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

[8] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. In *Proc. SIGGRAPH 2011*, 2011.

[9] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. Blog: Probabilistic models with unknown objects. In *In Proc. IJCAI*, 2005.

[10] Radford M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2010.

[11] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. *Journal of Machine Learning Research*, 32, 2014.

[12] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *In Proc. IJCAI*, 2001.

[13] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 2009.

[14] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2), 2011.

[15] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam Pocock, Stephen J. Green, and Guy L. Steele Jr. Augur: a modeling language for data-parallel probabilistic inference. *CoRR*, 2013.

[16] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. AISTATS 2011*, 2011.

[17] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating efficient mcmc kernels from probabilistic programs. In *Proc. AISTATS 2014*, 2014.

[18] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. In *Proc. SIGGRAPH 2012*, 2012.

Appendix: Procedural Modeling Code

```
local spaceship = qs.program(function()

    local terra lerp(lo: qs.real, hi: qs.real, t: qs.real)
        return (1.0-t)*lo + t*hi
```

```
    end

    local uniform = qs.func(terra(lo: qs.real, hi: qs.real)
        return lerp(lo, hi, qs.uniform(0.0, 1.0, {struc=false}))
    end)

    -- Wings are just horizontally-symmetric stacks of boxes
    local genWing = qs.func(terra(mesh: &Mesh, xbase: qs.real, zlo: qs.real, zhi: qs.real)
        var nboxes = qs.poisson(5) + 1
        for i in qs.range(0,nboxes) do
            var zbase = uniform(zlo, zhi)
            var xlen = uniform(0.25, 2.0)
            var ylen = uniform(0.25, 1.25)
            var zlen = uniform(0.5, 4.0)
            addBox(mesh, Vec3.create(xbase + 0.5*xlen, 0.0, zbase), xlen, ylen, zlen)
            addBox(mesh, Vec3.create(-(xbase + 0.5*xlen), 0.0, zbase), xlen, ylen, zlen)
            xbase = xbase + xlen
            zlo = zbase - 0.5*zlen
            zhi = zbase + 0.5*zlen
        end
    end)

    -- Fins protrude up from ship body segments
    local genFin = qs.func(terra(mesh: &Mesh, ybase: qs.real, zlo: qs.real, zhi: qs.real, xmax: qs.real)
        var nboxes = qs.poisson(2) + 1
        for i in qs.range(0,nboxes) do
            var xlen = uniform(0.5, 1.0) * xmax
            xmax = xlen
            var ylen = uniform(0.1, 0.5)
            var zlen = uniform(0.5, 1.0) * (zhi - zlo)
            var zbase = 0.5*(zlo+zhi)
            addBox(mesh, Vec3.create(0.0, ybase + 0.5*ylen, zbase), xlen, ylen, zlen)
            ybase = ybase + ylen
            zlo = zbase - 0.5*zlen
            zhi = zbase + 0.5*zlen
        end
    end)

    -- The ship body is a forward-protruding stack of boxes
    -- Wings and fins are randomly attached to different body segments
    local genShip = qs.func(terra(mesh: &Mesh, rearz: qs.real)
        var nboxes = qs.poisson(4) + 1
        for i in qs.range(0,nboxes) do
            var xlen = uniform(1.0, 3.0)
            var ylen = uniform(0.5, 1.0) * xlen
            var zlen = uniform(2.0, 5.0)
            addBox(mesh, Vec3.create(0.0, 0.0, rearz + 0.5*zlen), xlen, ylen, zlen)
            rearz = rearz + zlen
            -- Gen wing? (More likely closer to rear of ship)
            var wingprob = lerp(0.4, 0.0, i/qs.real(nboxes))
            if qs.flip(wingprob) then
                var xbase = 0.5*xlen
                var zlo = rearz - zlen
                var zhi = rearz
                genWing(mesh, xbase, zlo, zhi)
            end
            -- Gen fin?
            var finprob = 0.7
            if qs.flip(finprob) then
                var ybase = 0.5*ylen
                var zlo = rearz - zlen
                var zhi = rearz
                var xmax = 0.6*xlen
                genFin(mesh, ybase, zlo, zhi, xmax)
            end
        end
    end)

    return terra()
        -- Generate ship mesh
        var mesh : Mesh
        mesh:init()
        genShip(&mesh, -5.0)

        -- Enforce desired dimensions
        var bbox = mesh:bbox()
        var dims = bbox:extents()
        var targetWidth = 10.0
        var targetLength = 10.0
        qs.factor(qs.softeq(dims(0), targetWidth, 0.25))
        qs.factor(qs.softeq(dims(2), targetLength, 0.25))

        return mesh
    end
end)
-- Constrained sampling with MCMC
local query = qs.infer(spaceship, qs.Samples, qs.MCMC(qs.TraceMHKernel(), {numsamps=2000}))
return queryfn()
```