G2C: an optimizing transcompiler for probabilistic programming languages

Robert Brignull

1 Introduction

Recently research has been made about a new type of programming language, they are called probabilistic programming languages. There are probabilistic programming languages that extend various existing languages, examples include C, Java, Python, Haskell, Lisp, SQL, .NET and MATLAB. Obviously these base languages differ in terms of their paradigms and their execution speed, what we attempt in this project is bring techniques from compilers and programming languages to make a compiler from a high level probabilistic probabilistic language to a lower level one. We do this both for the performance increase of using a lower level language and also so we can perform some probabilistic optimizations along the way.

The field of probabilistic programming is a very interesting one with far reaching applications. One of its main uses is to open up the field of machine learning to non-experts. It is currently the case that to do any non-trivial machine learning one must design and build both the model and the inference algorithm oneself, this requires a large amount of statistical and general computer science knowledge. Probabilistic probabilistic languages aim to remove those needs by taking care of all the inference in a very general yet still efficient way. The user should be able to write the model they desire in what is almost a normal programming language using all of its powerful and expression, then the inference will be done automatically and efficiently by the language without any need for deep statistical knowledge.

Another benefit of probabilistic programming is that it allows generic and yet efficient inference on very complex models, add to this the ability to explore new models rapidly and with relatively little code and it could open up whole new approaches to machine learning. This not to say that probabilistic programming languages are trying to compete with current machine learning frameworks, they will likely never be as efficient at running current machine learning models as the current highly specialized algorithms. The point of probabilistic programming languages is to enable the construction of entirely new models without the need to write a new specialized inference algorithm each time.

Our source language is based off of the probabilistic programming language Anglican, which is itself built upon a subset of the language Lisp. This is a high level functional language and is interpreted. Our target language will be Probabilistic-C which is of course based on the language C. This is a much lower level imperative language and is compiled, it is also a very unconstrained PPL and this should allow us to do some interesting optimizations.

The bulk of the compiler is very standard, we transform into continuation passing style, perform closure conversion, hoist functions to the top level, and then output C code. One fortunate thing is that both Anglican and Prob-C handle observations in a similar way which makes the compilation considerably easier. We perform some non-probabilistic optimizations such as identifier reassignment and constant expression calculation. We then perform some probabilistic optimizations such as merging multiple samples from one distribution family into one sample, merging together multiple observes from the same family, merging any consecutive observes, and potentially completely removing observes when they forms a conjugate prior.

To give an idea of what the system achieves we will examine one model in detail and see what optimizations can be done to it. The situation is this: some crickets and grasshoppers will chirp more during hot whether, scientists think that this relationship is linear but with some various between individual animals. The scientists find some crickets and take readings of chirp frequency at different temperatures, they record the following data.

Temperature (°F)	Chirp frequency (Hz)
88.6	20.0
71.6	16.0
93.3	19.8
84.3	18.4
80.6	17.1
75.2	15.5

They then try to calculate the gradient of the relationship using the following probabilistic program. As ever with beyesian probability they have some prior beliefs, they think that the gradient is between 0 and 1 and they have some bound on the variation between individual animals.

```
[assume gradient (uniform-continuous 0 1)]

[assume coeff (normal gradient 0.05)]
[assume const (normal 0 0.2)]

[observe (normal (+ (* 88.6 coeff) const) 0.1) 20.0]
[observe (normal (+ (* 71.6 coeff) const) 0.1) 16.0]
[observe (normal (+ (* 93.3 coeff) const) 0.1) 19.8]
[observe (normal (+ (* 84.3 coeff) const) 0.1) 18.4]
[observe (normal (+ (* 80.6 coeff) const) 0.1) 17.1]
[observe (normal (+ (* 75.2 coeff) const) 0.1) 15.5]

[predict gradient]
```

The compilation follows standard steps of transforming into continuation passing style, closure conversion, hosting of functions to the top level. Below is the output if we did not then apply any probabilistic optimizations.

```
#include <stdlib.h>
#include <stdio.h>

#include "probabilistic.h"

int main(int argc, char **argv) {
    double gradient = uniform_rng(0.0, 1.0);

    double coeff = normal_rng(gradient, 0.05);
    double const_ = normal_rng(0.0, 0.2);

    observe(normal_lnp(20.0, coeff * 88.6 + const_, 0.1));
    observe(normal_lnp(16.0, coeff * 71.6 + const_, 0.1));
    observe(normal_lnp(19.8, coeff * 93.3 + const_, 0.1));
    observe(normal_lnp(18.4, coeff * 84.3 + const_, 0.1));
    observe(normal_lnp(17.1, coeff * 80.6 + const_, 0.1));
    observe(normal_lnp(15.5, coeff * 75.2 + const_, 0.1));

    predict("%s,%f\n", "gradient", gradient);

    return 0;
}
```

However, there are a few optimizations that we can perform and we will list them in the order that they are performed.

1. The first thing to notice is that the normal distributions of the coeff and const samples are conjugate priors to the normal distributions of the observations. We will explain what we mean by this later on. What this optimization does when we apply it is to remove the reference to the sampled variable from the observe, the observe will then be in terms of the parameters to the sample instead. If we apply this ten times we can remove all references to the coeff and const variables from the observes and have them only in terms of the gradient variable.

   After this optimization we are making three samples and six observes.

2. At this point the coeff and const variables are not actually used at all and so we can remove them completely from the program.

   After this optimization we are making one sample and six observes.

3. We want to collect as many observes as possible together so we try to push them as far back down as we can. We do this in the obvious way by blindly pushing observes backwards, even going as far as to push them inside continuations. After doing this we successfully manage to collect together the six observes.

   After this optimization we are making one sample and six observes.

4. The final optimization is to merge together any consecutive observes into one, this is why we collected them together. Here we can merge all six into one. This optimization works because the probability of all of a collection of events happening is the product of their individual probabilities. Observes work using the log of the probability however so we can calculate the log of the product as the sum of the logs.

After this optimization we are making one sample and one observe.

So after all that we have reduced three samples and six observes into just one sample and one observe. This will improve both performance and sample quality greatly.

To make the program prettier here we have made all arithmetic calculations in-line and reduced them as much as possible. This is something the system could do but currently does not, in practice it would not affect program performance much so there is little encouragement to do this. What follows is the final program after reducing the arithmetic by hand.

```
#include <stdlib.h>
#include <stdio.h>

#include "probabilistic.h"

int main(int argc, char **argv) {
    double gradient = uniform_rng(0.0, 1.0);

    observe(normal_lnp(20.0,
                       gradient * 88.6,
                       392.798)
          + normal_lnp(16.0,
                       gradient * 0.009572350164697 + 16.1603674152755,
                       0.172665339437)
          + normal_lnp(19.8,
                       gradient * 0.0215414920690030 + 20.9779100181940,
                       0.171472119913)
          + normal_lnp(18.4,
                       gradient * 0.00714550738045528 + 18.5274790332219,
                       0.132812797595)
          + normal_lnp(17.1,
                       gradient * 0.0032241962331263 + 17.6916967883102,
                       0.123114249847)
          + normal_lnp(15.5,
                       gradient * -0.0005048414824 + 16.4109671553285,
                       0.118203847628));

    predict("%s,%f\n", "gradient", gradient);
    return 0;
}
```

2 Background

2.1 Probabilistic programming

Superficially a probabilistic programming language may appear to be just like a normal one, in fact one could say that if you've ever written a program that calls a random number generator then you've written a probabilistic program. PPLs do indeed make it easy to sample from various probability distributions by having them built in to the language, but they do much more than that as well. A PPL allows you to condition the entire execution trace on some observed data, and I will explain what I mean by that in the next few paragraphs.

For the sake of this explanation a good way of thinking about all programs would be to consider the distribution of outputs they produce. For completely deterministic programs they may only produce one possible output, once you start involving random number generators they produce a distribution over many values. What we're interested in is the output or posterior distribution of the program. There are a few ways to calculate this posterior distribution, either by running it many times to generate an approximation which gets closer to the true posterior the more you run it, or by using static analysis on the program to track the distributions through it and calculate the posterior precisely.

Where things get more complicated is the when we introduce the true power of PPLs which is the ability to observe, this allows you to condition the entire program execution on some observed data and change the poterior distribution to match. Essentially what you're doing is saying that we observed this statement to be true and any run of the program which does not match that statement can be discarded, we only want program executions that match our observations. When getting output via sampling this can either be done in a hard (rejection sampling) or soft (particles) way. In rejection sampling the observe statements act like an assume statement would, you run the program and if an observe fails then you restart the program from the beginning and try again. With particle sampling you run the program many times simultaneously and have them synchronize at observe statements where you calculate not just whether an observe is satisfied but rather what is the probability of it being satisfied, you then kill off less likely particles and clone new ones from those left. Of course actual sampling techniques are far more sofisticated, but this hopefully gives you an idea of how they work.

Usually a probabilistic programming language is based on an existing language rather than being a completely new one, examples include: Venture (Lisp), Anglican (Lisp), IBAL (OCaml), Infer.NET (.NET), PSQL (SQL), FACTORIE (Scala), Alchemy (C++), Church (Lisp), Stan (none). More are being designed constantly as people discover new ways of using them.

The applications of probabilistic programming are diverse, anything that requires estimating a distribution given prior beliefs and conditioned on some observed data. They can be used in artificial intelligence, predicting stock prices, recomending products to customers, diagnosing problems in computers/cars/whatever, cyber security, image detection, etc.

2.2 Anglican and Prob-C

Anglican and Prob-C are the two languages we'll be concentrating on, Anglican because it is the basis for the source language of my compiler, and Prob-C because it is the target language of my compiler.

Anglican is based on the Venture modeling language which is itself built upon Lisp. It is effectively one language embedded within another, specifically it is built from a sequence of Assume, Observe, and Predict directives, each of which contain expressions which can from a safe, functional subset of Lisp. Being Lisp like, Anglican is quite high level and this makes it nice and easy to create complex models, however it is interpreted and therefore for those same complex models it runs quite slowly.

Prob-C on the other hand is based on full C rather than a subset, in fact the only difference is the addition of a library of sampling functions, two extra

functions to observe and predict, and a macro which redefines the main function. So Prob-C has all the power of C and most of the speed. Unlike almost all other probabilistic programming languages, Prob-C is unsafe in the sense that it is possible to create programs which make no sense statistically, for this reason as well as C's unfriendliness it is not recommended to code in Prob-C directly.

Combine the facts about these two languages and you can see there is an obvious motive for creating a compiler between them.

3 The compiler

3.1 The source language

For this project I needed a high level probabilistic programming language to be the source language of my compiler. I had two choices: to use an existing one or to write my own. In fact I went somewhere in between by modifying Anglican to include some extra type information to make compilation easier. Compiling from an untyped language to a typed one is difficult, one way would be to forget the idea of using native types and introduce a layer of abstraction around all variables, but this incurs a large performance penalty. I therefore made the choice to introduce some extra type information into my source language so that I always know the type of a variable or function argument or a function return type.

What follows are a couple of examples of my language, which I will refer to as G. This first example shows that for programs not involving user-defined functions or lists, G is identical to Anglican.

```
[assume iscloudy (flip 0.5)]

[assume israining (if iscloudy (flip 0.8) (flip 0.2))]

[assume sprinkler (if iscloudy (flip 0.1) (flip 0.5))]

[assume pwetgrass (if (and sprinkler israining) 0.99
(if (or  sprinkler israining) 0.9
0))]

[observe (flip pwetgrass) true]

[predict israining]
```

However for more complicated programs you can see the extra type information in the function arguments, function return type, and anywhere that something is extracted from a list. This second example demonstrates all of these.

```
; Prior drill beliefs
[assume state-of-nature (list 0.5 0.3 0.2)]

; Oil quantity: dry, wet or soaking (0, 1 or 2)
[assume oil-quant (discrete state-of-nature)]

; Sensor reliability conditional distribution
[assume sound-state-dist (lambda (o : Num) -> List
(cond ((= o 0) (list .6 .3 .1))
((= o 1) (list .3 .4 .3))
(else    (list .1 .4 .5))))]

; Utility function
[assume utility (lambda (drill : Bool) -> Num
(if drill
(nth Num (list -70 50 200) oil-quant)
0))]

; Make a sound observation
[observe (discrete (sound-state-dist oil-quant)) 2]

; Predict the oil quantity in the well
[predict oil-quant]

; What is the utility if we decide to drill?
[predict (utility true)]

; What is the utility if we don't?
[predict (utility false)]

; With what probability should I drill?
[assume should-i-drill (if (> (utility true) (utility false)) 1 0)]
[predict should-i-drill]
```

What follows is an exact description of the syntax of G.

```
<program> ::= <directive> | <directive> <program>
<directive> ::= "[" "assume" <id> <expr> "]" |
"[" "observe" <expr> <expr> "]" |
"[" "predict" <expr> "]"
<exprs> ::= <expr> | <expr> <exprs>
<expr> ::= <number> | <boolean> | <empty-list> | <id> |
 <let-expr> |
 <lambda-expr> | <mem-expr> |
 <if-expr> | <cond-expr> |
 <prim-expr> | <typed-prim-expr> |
 <app-expr>
<let-expr> ::= "(" "let" <id> <expr> <expr> ")"
<lambda-expr> ::= "(" "lambda" <lambda-type> <expr> ")"
<lambda-type> ::= "(" <lambda-args> ")" "->" <type>
<lambda-args> ::= <id> ":" <type> | <id> ":" <type> <lambda-args>
<mem-expr> ::= "(" "mem" <expr> ")"
<if-expr> ::= "(" "if" <expr> <expr> <expr> ")"
<cond-expr> ::= "( "cond" <cond-args> ")"
<cond-args> ::= "(" "else" <expr> ")" |
"(" <expr> <expr> ")" <cond-args>
<prim-expr> ::= "(" <prim> <exprs> ")"
<typed-prim-expr> ::= "(" <typed-prim> <type> <exprs> ")"
<app-expr> ::= "(" <exprs> ")"
<prim> ::= "+" | "-" | "*" | "/" |
 "=" | "!=" | "<" | ">" | "<=" | ">=" |
 "and" | "or" | "not" |
 "list" | "cons" | "rest" | "empty" | "count" |
 "log" | "log10" | "exp" |
 "pow" | "sqrt" | "cbrt" |
 "floor" | "ceil" | "round" | "rint" |
 "abs" | "signum" |
 "sin" | "cos" | "tan" |
 "asin" | "acos" | "atan" |
 "sinh" | "cosh" | "tanh" |
 "inc" | "dec" | "mod" |
 "flip" | "beta" | "gamma" | "normal" | "poisson" |
 "geometric" | "exponential" |
 "uniform-continuous" | "uniform-discrete" | "discrete"
<typed-prim> ::= "first" | "second" | "nth" | "categorical"
<types> ::= <type> | <type> "," <types>
<type> ::= "Num" | "Bool" | "List" | "(" <types> ")" "->" <type>
```

3.2 Typing the source language

The first stage of compilation is to type the source language, that is to assign a type to every node of the abstract syntax tree (AST). The way typing works throughout the compiler is that here we assign a type to every single expression and then later we reduce that to only remembering the types of ids. The typing itself is very easy, all the information is there and only minimal inference has to be done. The checks performed are:

- That all variables are defined before they are used.
- For all built-in (untyped) prims (e.g. `+`, `-`, `and`, `or`) we know their type or can give them one from their arguments. Then we just check that the

types of the arguments match what we expected. What I mean by give them a type is that for example if `=` had 4 arguments and the first was a number, then we would give it the type

```
(Num, Num, Num, Num) -> Bool
```

For functions such as `first` we also use the user supplied type, often simply using it as the return type. It is worth nothing that although in G the first argument appears to be a type it is not a true argument, a better way of thinking about it that it specifies which of a set of functions you're referring to.

- That the given return type of a lambda matches the type that its body evaluates to. The reason for making the user specify a return type at all is for recursive functions so we don't have to do any type guesswork at all.
- That the two branches of an if expression have the same type.
- In a plain old application we use the map from expressions to types that we have built up to type each expression, we then check that the first expression is a function type and its argument types match the types of the rest of the expressions.
- For an observe directive we check that the outer expression of the first argument is a probabilistic primitive. The list of probabilistic primitives is: `flip`, `beta`, `gamma`, `normal`, `poisson`, `geometric`, `exponential`, `uniform-continuous`, `uniform-discrete`, `discrete`, `categorical`.

3.3 Making ids unique

At this point in the compilation we change all ids to be unique, so whenever two ids are the same they always refer to the same variable. This means that we can work out scoping rules now and then never have to think about them again for the rest of the compilation. It also removes the complication that Lisp's list of characters allowed in ids is a lot larger than C's. This does have the downside that it makes the outputted C harder to understand, but that is a price worth paying.

3.4 Into continuation passing style

The next step of compilation is to transform into continuation passing style (CPS). The primary difference here is that functions and arithmetic operators and such like do not return values but instead take as an extra argument an explicit continuation, that is a function of one argument which caries on the computation. The reason for doing this is that it makes explicit a lot of things that were originally implicit, e.g. intermediate values, order of evaluation. This immediately brings the style of execution a lot closer to that of C, in that it brings it away from a functional language with nested expressions and more towards an imperative one where statements are executed in order. The exception to the transformation is built-in primitives which we assume can be done atomically by the target language, hence they can be calculated directly and stored into variables.

I wont say too much about the actual transformation as it's fairly standard, the two points I will mention are:

- When transforming an if expression, generate the continuation before and then use the same one in each branch, rather than generating it within the branch. This has the effect of increasing the number of continuations but reduces the overall amount of code generated and reduces the nesting level.
- When transforming a let expression of a lambda, make sure that the resulting generated lambda is assigned to the correct id rather than to a new or temporary one. This is because the lambda could be recursive and we must make sure that it is assigned to the id it is expecting.

3.5 Closure conversion and hoisting

The goal of this step of compilation is to lift all functions to the top level of the program, as that is the way that functions must be defined in a C program. The problem we must overcome is that functions may have free variables, that is variables which are used in the function but not defined within it or one of the arguments. The way we will fix this to effectively add more arguments to functions so that they no longer contain any free variables. In practice however we keep the original arguments and the new ones separate, what we do is that whenever a function is created we store the values of all its free variables into some kind of bundle or closure, then when the function is executed we unpack the bundle to get at the values of any free variable that are needed.

The first thing we do is to go though the entire program and every function to find the free variables, that is to work out which variables are used in the function but not defined within it or one of the arguments. Once that is done we then make the following changes:

- The list of types and names of a function's free variables is added to the function definition and that definition is then lifted or hoisted to the top level. The function is also changed to accept as its first argument a bundle containing the free variables, when executed the first thing the function will do is to unpack the bundle into variables of the correct name and then it will execute its original body.
- Where a function was originally defined we instead form a bundle containing the name of the function being called and a list of all its free variables.
- Any invocations of functions will have to be changed to have a previously created bundle passed as the first argument.

3.6 Into C

Before outputting C code we first do a final transformation into a C-like AST. We try to do as much of the calculation here to make the printing itself trivial, most of the work is about allocating ids to bundles and making sure that they are packed and unpacked correctly.

We alter the idea of a bundle to be made of two parts. Firstly there is the part that we call the data, this contains the actual values of the free variables and is specific to a particular function. Then there is the part we will continue to call the bundle, this contains a pointer to the function to the executed and a pointer to a data object, each bundle relates to a function signature (i.e. a function type). We make one pass though the code, creating bundles and data objects and explicitly packing each individual free variable into a data object. We also make explicit the unpacking of variables from data objects.

One important point to make is that we do not try to do any explicit memory management. The main reason for this is that it is unnecessary. It is important to understand that the program execution happens within many short lived threads, as each thread dies all its allocated memory is freed by the operating system. Therefore as it is such a complex thing to do and the memory usage benefits are small, we decide to not do explicit memory management.

3.7 Outputting C code

We need to output valid C code, including struct definitions and any extra library functions used. Mostly the translation here is obvious and direct, the only difficult point is to make sure all structs and functions are declared before they are defined to allow full recursion.

There are a few times that we need to add in some extra C code to provide an implementation of linked lists or an extra math function, also some optimizations require observing the result of a specific calculation. To this end some C code has been written in files to be included in the output if needed. Usually this is done by checking to see if any of a set of primitive functions is used.

4 Optimizations

Although just having a compiler from a high level probabilistic language into a lower lever one such as Prob-C would be a good enough project in itself, there is potential for massive increases in both performance and the quality of samples though a combination of a few simple optimizations.

All the optimizations described here are performed immediately after CPS transformation. In this stage the syntax is in a good balance of simple and expressive, for example all expressions have been split up so we only calculate one primitive or application at a time, but lambdas are still in-line and so we don't have to worry about variable scope as much. It is however still often useful to know the full history of a variable, that is exactly how it was calculated rather than just the last step, so we calculate this history before doing any optimizations and rebuild the information after each one.

4.1 Non-probabilistic optimizations

We do some standard non-probabilistic optimizations. None of these are technically necessary for the rest of the optimizations to work and some would be performed by the C compiler anyway. They still make our life a little easier however so that is why they are included here.

4.1.1 Constant expression calculation

When the arguments to a primitive such as plus or times are all constant, we can easily calculate what the answer should be and set the id equal to that rather than applying a primitive.

For example we could do the following optimization

```
let x = prim + 2 5 in      ⇒    let x = 7 in
...                             ...
```

This can be extended to the case when only some of the arguments are constant, now we can't remove the whole primitive application but we can reduce the number of arguments and hence the complexity of it.

For example we could do the following optimization

```
let x = ...  in                 let x = ...  in
let y = prim + x 2 5 in   ⇒     let y = prim + x 7 in
...                             ...
```

We can optimize all of the following when at least some of the arguments are constant:

`+`, `-`, `*`, `/`, `eq`, `neq`, `<`, `>`, `<=`, `>=`, `and`, `or`, `not`

This sort of optimization would definitely be performed by any half decent C compiler, so we are not actually improving the performance of the final compiled program by doing this. The reason for doing it is it reduces the size and complexity of the generated Prob-C code greatly, therefore making other stages of compilation faster and easier and also making it easier for a human to scan and understand the outputted Prob-C code.

4.1.2 Merging multiple arithmetic expressions

Another thing we can do with arithmetic expressions is to merge ones of the same sort together.

For example we could do the following optimization

```
let x = ...  in
let y = prim + x 5 in          let x = ...  in
let z = prim + y 8 in    ⇒     let w = prim + x 9 in
let w = prim + z -4 in         ...
...
```

This is a case that comes up in compiled programs surprisingly often, especially after other optimizations, so by optimizing it away we reduce the final complexity of the Prob-C.

Again this optimization would likely be performed by the C compiler anyway so the final compiled program is unchanged.

4.1.3 Removing let expressions where the id is not used

A very simple optimization is to remove any let expression where the variable being assigned to is never used later on in the program.

This optimization doesn't do much on its own, as the user is unlikely to have written a program that doesn't use one of its variables, but almost all of the other optimizations we do here will leave orphaned variables that this process will clean up.

This is an optimization that a C compiler potentially would not perform. This is because C is a language with side effects so just because a variable is never used does not mean that the process of calculating it did not change program state. In our language however variables are immutable and there is no state so it is safe to remove the unused variable.

4.1.4 Removing let expressions where an id is assigned to another id

Another very simple optimization is to remove a variable when it is assigned to the value of another variable. We then change all references to the removed variable in the rest of the program to the variable it was assigned to.

For example we could do the following optimization

```
let x = ...  in                    let x = ...  in
let y = x in                   ⇒   let z = prim + x 1 in
let z = prim + y 1 in              ...
...
```

This optimization is admittedly rather inconsequential, but we include it anyway.

4.1.5 Removing trivial continuations

The final non-probabilistic optimization we do it to remove trivial functions which are equivalent to another function. The place where this happens a lot is continuations generated from the translation into CPS.

For example we could do the following optimization

```
let f = lambda x y z -> g x y z in    ⇒    let f = g in
...                                        ...
```

This will then of course be removed on the next pass as it is an id assigned to another id.

This optimization could be moderately important for performance. Although a decent C compiler will inline a small function such as the above and hence remove the extra application, after the other transformations that G2C performs it will not be simple, so removing it at this stage will greatly simplify later stages of compilation.

4.2 Samples

One of the things we'd like to do is to reduce the number of samples as some of them can be quite costly, and more importantly improve the quality of the ones we do. By this I mean we'd like to have a high level of mixing, so we need to make fewer runs of the program before the output has converged enough towards to the correct output.

4.2.1 Merging samples

One way we can improve performance is to reduce the number of samples as some of them can be moderately costly to perform. The ideal case for this optimization is shown below.

```
[assume x_1 (normal m_1 b_1)]
...
[assume x_n (normal m_n b_n)]
[assume y (+ x_1 ... x_n)]
```

The general form is where we make many samples and perform some function on them, we then use that function output and never use the individual samples anywhere else.

Assuming that none of the `x_i` are used anywhere else then the above can be optimized into the following

```
[assume y (normal (+ m_1 ... m_n) (+ b_1 ... b_n))]
```

We can even perform this optimization in the case where some of the `x_i`s are used again. If only `x_1` were used elsewhere then we could optimize to the following

```
[assume x_1 (normal m_1 b_1)]
[assume t (normal (+ m_2 ... m_n) (+ b_2 ... b_n))]
[assume y (+ x_1 t)]
```

Although less of an optimization we have still reduced the number of samples.

There are many cases where we can apply this optimization, some of which I have listed below. I have only included here a few for the normal distribution, but the optimization can be applied to many other distributions and operations.

Before optimization		After optimization

`[assume x_1 (normal m_1 b_1)]`
`...`
`[assume x_n (normal m_n b_n)]`
`[assume y (+ x_1 ... x_n)]`
	⇒	`[assume y (normal (+ m_1 ... m_n) (+ b_1 ... b_n))]`

`[assume x_1 (normal m_1 b_1)]`
`...`
`[assume x_n (normal m_n b_n)]`
`[assume y (- x_1 ... x_n)]`
	⇒	`[assume y (normal (- m_1 ... m_n) (+ b_1 ... b_n))]`

I will also prove the correctness of the first example.

Claim: The sum of independent normally distributed variables is normally distributed.
Proof: The easiest way to do this is using characteristic functions. For a random variable X this is

$$\varphi_X(t) = E(e^{itX})$$

The characteristic function of a normal distribution with mean μ and variance σ^2 is

$$\varphi(t) = \exp(it\mu - \frac{\sigma^2 t^2}{2})$$

The characteristic function of the sum of two independent random variables X and Y is the product of their separate characteristic functions. Hence if X and Y are both normally distributed with means and variances μ_X, μ_Y and σ_X^2, σ_Y^2 respectively, then

$$\begin{aligned}\varphi_{X+Y}(t) &= \varphi_X(t)\varphi_Y(t) \\ &= \exp\left(it\mu_X - \frac{\sigma_X^2 t^2}{2}\right)\exp\left(it\mu_Y - \frac{\sigma_Y^2 t^2}{2}\right) \\ &= \exp\left(it(\mu_x + \mu_Y) - \frac{(\sigma_X^2 + \sigma_Y^2)t^2}{2}\right)\end{aligned}$$

Which is the characteristic function of a normal distribution with mean $\mu_X + \mu_Y$ and variance $\sigma_X^2 + \sigma_Y^2$.

From here if we have n independent normally distributed random variables $X_i \sim N(\mu_i, \sigma_i^2)$ then we proceed by induction using the above to get that

$$\sum_{i=1}^{n} X_i \sim N(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \sigma_i^2)$$

□

4.2.2 Doing arithmetic on samples

Another thing we can do is merge some arithmetic operations into the sample, as shown below. Although not very helpful at all on its own, the hope is it would allow us to later merge more samples as in the previous optimization. With these two together we can merge any linear combination of samples into only one sample.

Here are some examples of the sort of optimizations we could do. However note that there are many others of this form that could be done and not just involving the normal distribution.

Before optimization		After optimization
`[assume x (normal m b)]` `[assume y (+ x r)]`	⇒	`[assume y (normal (+ m r) b)]`
`[assume x (normal m b)]` `[assume y (* x s)]`	⇒	`[assume y (normal (* m s) (* b s s))]`

4.2.3 Pushing samples back

Something we can do which has the capacity to greatly improve the quality of samples made is to move samples after potentially costly observe statements.

The idea is that when an observe is made that kills almost all of the particles, the ones which remain will all have been cloned from some small set and hence all samples that have been made also come from some small set. However by moving the sample after the observe, whatever happens with the cloning of particles, they will all have different values of the sample just made.

An example of what we might do is

```
[assume x (poisson 1)]                [assume x (poisson 1)]
[assume y (normal 0 1)]       ⇒       [observe (normal x 1) 100]
[observe (normal x 1) 100]            [assume y (normal 0 1)]
[predict y]                           [predict y]
```

The only check we make when pushing samples back is whether the operation immediately following the samples uses the sampled value, if not then we move the sample past. Note that in a perfect world we'd be able to remove the observe and the value `x` completely as they have no effect on the predicted value `y`, however we do not consider this as it is hard to work out when it is safe to do.

Care must be taken when implementing this so as to not end up in a loop, for example where you are swapping two samples back and forth forever. The crucial point when implementing this therefore is to avoid just swapping with another sample.

So, in that vein, to push a sample back, scan from its position considering each statement encountered in the following manner:

- If the statement depends on the sampled value, then the sample cannot be moved back any further.
- If the statement is a `let` expression defining a new variable that does not depend on the sampled value, then just ignore it and move on to the next statement.
- If the statement does depend on the sampled value and is not a `let` expression, then the original sample is moved to be right after this statement. The process is then started again from the sample's new position.

This process is run repeatedly for every sample until no more changes can be made.

4.3 Merging of observe statements

In most cases it is the observations that govern running time rather than any computation being performed, so much so that the cost of sampling and other computation is often negligible. Observations also impact on sample quality by causing the death and cloning of particles. This set of optimizations focus on merging observations together and can improve the speed of a program greatly.

4.3.1 Merging observes of the same distribution

The scenario here is that there are multiple consecutive observes of the same distribution. What we note is that it is usually the case then that their conjunction can be written in a nicer form, as a function of some selection of properties of the observed data such as the sum or the sample mean and variance.

This optimization can be applied to many different distributions, some are obvious such as multiple identical Bernoulli observes being equivalent to one binomial observe, some are more complicated such as the case of normal distributions which we will focus on.

If we have the situation

$$[\mathsf{observe}\ (\mathsf{normal}\ m\ b)\ x_1]$$
$$\ldots$$
$$[\mathsf{observe}\ (\mathsf{normal}\ m\ b)\ x_n]$$

then we can show that it is equivalent to an observe that only relies on the sample mean and variance of the x_i's. Note however that the resulting observe will not be of a normal distribution, in fact it will not be of any of the distributions that Prob-C has a primitive for.

To work out what distribution it will have, consider the joint probability density function, which can be written as a product of the individual densities as all the observations are independent. Here $\bar{x}$ denotes the sample mean and s^2 the sample variance.

$$\begin{aligned}
&\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}} \\
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2} \\
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}((x_i-\bar{x})-(\mu-\bar{x}))^2} \\
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}(\sum_{i=1}^{n}(x_i-\bar{x})^2+\sum_{i=1}^{n}(\mu-\bar{x})^2-2\sum_{i=1}^{n}(x_i-\bar{x})(\mu-\bar{x}))}
\end{aligned}$$

Then since $\sum_{i=1}^{n}(x_i-\bar{x})(\mu-\bar{x}) = 0$

$$\begin{aligned}
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}(\sum_{i=1}^{n}(x_i-\bar{x})^2+\sum_{i=1}^{n}(\mu-\bar{x})^2)} \\
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\bar{x})^2} e^{-\frac{n}{2\sigma^2}(\mu-\bar{x})^2} \\
&= (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{n-1}{2\sigma^2}s^2} e^{-\frac{n}{2\sigma^2}(\mu-\bar{x})^2}
\end{aligned}$$

And importantly for us

$$\begin{aligned}
&\log(\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}) \\
&= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{n-1}{2\sigma^2}s^2 - \frac{n}{2\sigma^2}(\mu-\bar{x})^2
\end{aligned}$$

We can then replace the sequence of observes or normal distributions by one observe of a function which calculates the above. This is the first optimization that is only possible within Prob-C, because you are not limited to observing a standard builtin distribution but can pass any numerical value you want to an observe.

4.3.2 Merging any consecutive observes

This optimization relies on the way that observing works in Prob-C, it is specific to Prob-C and might not be possible in other languages, this makes it rather

interesting. The way that Prob-C observes work is that you call the observe function with the log of the probability of the event happening. For all the standard distributions this relates to the log of the p.d.f. at that point, but the crucial point is that we're not limited to named distributions, any floating point value can be passed to the observe function.

We therefore perform the following optimization

```
observe(dist_1(args_1));
...                          ⇒  observe(∑_{i=1}^{n} dist_i(args_i));
observe(dist_1(args_2));
```

4.4 Removal of observe statements

The best way of improving both performance and mixing is to remove observe statements completely, so that is what we try to do here.

4.4.1 Commuting samples and observe statements

In certain cases we can compute the effect that an observe will have on a sample directly, we can then make those changes to the sample and potentially remove the observe. The term for what we're doing here is conjugate prior and there is much literature on the subject.

As an example we could perform the two following optimizations

```
[assume p (beta a b)]
[observe (flip p) true]   ⇒  [assume p (beta (+ a 1) b)]
```

```
[assume m (normal 3 1)]
[observe (normal m 1) 6]  ⇒  [assume m (normal 4.5 0.5)]
```

These are for the simple cases where the variable `a`, `b`, `m` are constants, if they are not then we might not be able to remove the observe completely as I explain later. This optimization has potential for huge increases in both performance and sample quality and could make it possible to write program which we infeasible before.

I will now give a proof of correctness for the first example in order to show what happens when parameters are unknown. Suppose we have the scenario as in the first example, then this corresponds to having random variables $p \sim Beta(\alpha, \beta)$ and $x \sim Bernoulli(p)$, so $\mathrm{P}(p) = \frac{p^{\alpha-1}(1-p)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}$ and $\mathrm{P}(x \mid p) = p$.

By Bayes theorem

$$\begin{aligned}
\mathrm{P}(p \mid x) &= \frac{\mathrm{P}(x \mid p)\mathrm{P}(p)}{\mathrm{P}(x)} = \frac{\mathrm{P}(x \mid p)\mathrm{P}(p)}{\int_0^1 \mathrm{P}(x \mid p')\mathrm{P}(p')dp'} \\
&= \frac{\frac{p^{\alpha}(1-p)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}}{\int_0^1 \frac{p'^{\alpha}(1-p')^{\beta-1}}{\mathrm{B}(\alpha,\beta)}dp'} \\
&= \frac{p^{\alpha}(1-p)^{\beta-1}}{\int_0^1 p'^{\alpha}(1-p')^{\beta-1}dp'} \\
&= \frac{p^{\alpha}(1-p)^{\beta-1}}{\mathrm{B}(\alpha+1,\beta)}
\end{aligned}$$

Hence $p \mid x \sim Beta(\alpha + 1, \beta)$. Then what's left is

$$\begin{aligned}
\mathrm{P}(x) &= \int_0^1 \mathrm{P}(x \mid p')\mathrm{P}(p')dp' \\
&= \int_0^1 \frac{p'^{\alpha}(1-p')^{\beta-1}}{\mathrm{B}(\alpha,\beta)}dp' \\
&= \frac{\mathrm{B}(\alpha+1,\beta)}{\mathrm{B}(\alpha,\beta)} \\
&= \frac{\alpha}{\alpha+\beta}
\end{aligned}$$

Thus as $\mathrm{P}(x \mid p)\mathrm{P}(p) = \mathrm{P}(x)\mathrm{P}(p \mid x)$ we can replace the sample to be from the $Beta(\alpha + 1, \beta)$ distribution and the observe to be of the value $\frac{\alpha}{\alpha+\beta}$.

If α and β are constant then this observe can then be removed entirely as is explained in the next optimization, but if not then the observe must remain.

Note that the observe is no longer one of Anglican's distributions and so the optimization cannot be performed within Anglican, however within Prob-C or out intermediate language we can observe any distribution we like or even just values.

I will also detail here the formula used for the conjugate prior of the normal distribution. With this optimization it is possible to go beyond merely the example given but instead to have the mean of the observed normal distribution be some linear function of a normally sampled variable. Suppose we have the following scenario

```
[assume m_2 (normal m_1 b_1)]
[observe (normal (+ (* m_1 coeff) const) b_2) v]
```

Then they can be replaced by

```
[assume m_2 (normal nm_1 nb_1)]
[observe (normal nm_2 nb_2) v]
```

where

$$\begin{aligned}
\texttt{nm_1} &= \left(\frac{1}{\texttt{b_1}} + \frac{\texttt{coeff}^2}{\texttt{b_2}}\right)^{-1}\left(\frac{\texttt{m_1}}{\texttt{b_1}} + \frac{\texttt{coeff}}{\texttt{b_2}}\left(\texttt{v} - \texttt{const}\right)\right) \\
\texttt{nb_1} &= \left(\frac{1}{\texttt{b_1}} + \frac{\texttt{coeff}^2}{\texttt{b_2}}\right)^{-1} \\
\texttt{nm_2} &= \texttt{coeff} \times \texttt{m_1} + \texttt{const} \\
\texttt{nb_2} &= \texttt{b_1} \times \texttt{coeff}^2
\end{aligned}$$

Again if `m_1` and `b_1` are constant then the observe can be removed completely, but even if not the observe is no longer dependent on the value `m_2` and this is an improvement. It is possible to remove a normal observe whose mean is a linear combination of any number of normally sampled variables with constant parameters.

4.4.2 Removal of constant observe statements

Here the idea is that when all of the arguments to an observe statement are constant, that observe can have no influence on the posterior distribution and

so we can remove the observe. The reason for this is that when it comes to allocating children to the particles, the distribution used to pick them is a uniform one. This does not mean that every particle gets precisely one child, but averaging over many iterations the posterior distribution is unchanged. Removing the observe will increase performance and possibly increase the mixing as well but only to a slight extent.

5 Experimental data

Here I will show to you with experimental data just how much benefit the different optimizations can give. For each one I will usually show both how it affects performance and mixing, that is how quickly we can make samples and how quickly those samples converge to the correct posterior distribution.

5.1 Pushing samples backwards

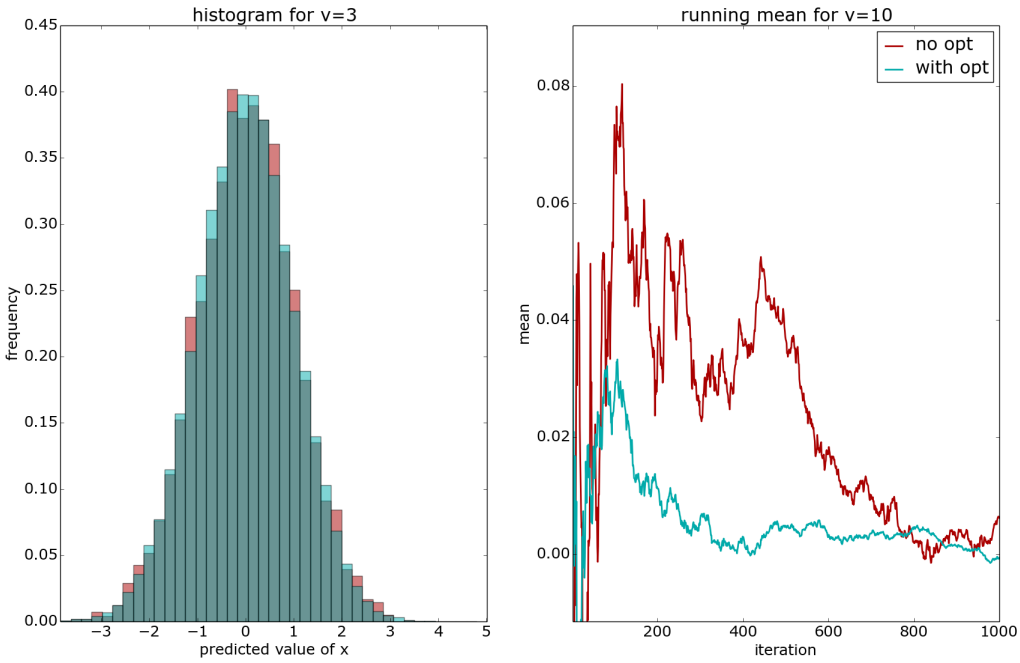
To test this optimization I used a program of the following form

```
[assume x (normal 0 1)]
[assume y (poisson 1)]
[observe (normal y 0.1) v]
[predict x]
```

where the value of `v` can be varied. Programs of the above form were compiled with either no optimizations enabled or all optimizations enabled, then the resulting programs were run for 1000 iterations with 100 particles.

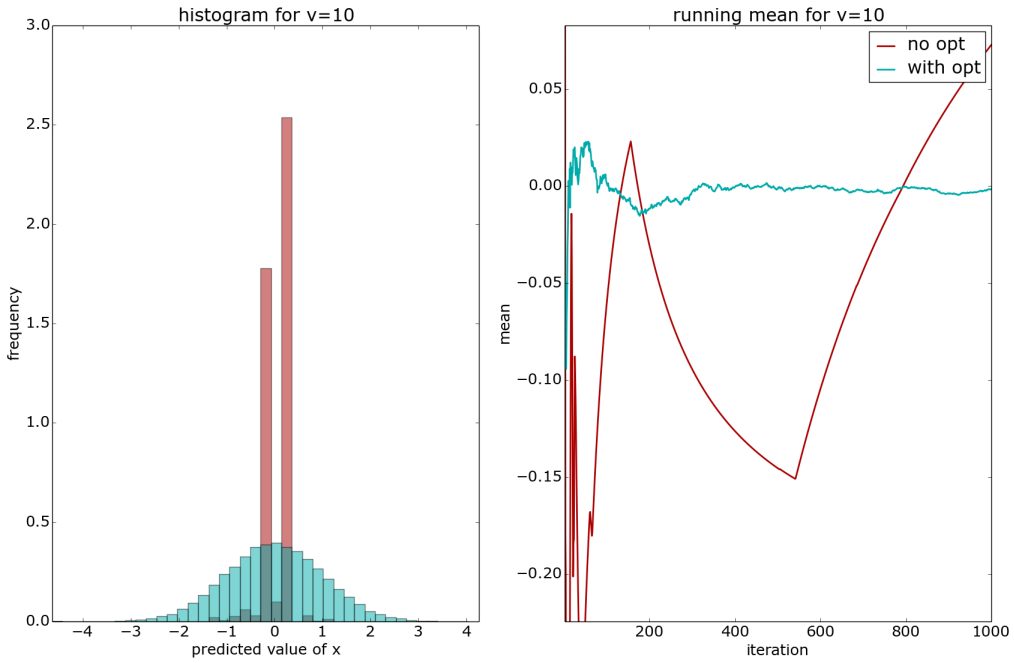
I will show the resulting histograms and a visualization of how quickly it converges to the final posterior distribution. I should point out that the running time of the optimized and non-optimized programs is identical.

First consider the case of `v = 3`, as shown in the following graphs.



This case demonstrates when the non-optimized program is still usable but the optimized program is superior in its mixing.

Now consider instead the case of `v = 10`, as shown in the next two graphs.



This case demonstrates when the non-optimized program is completely useless however the optimized program is working exactly as well as it was before.

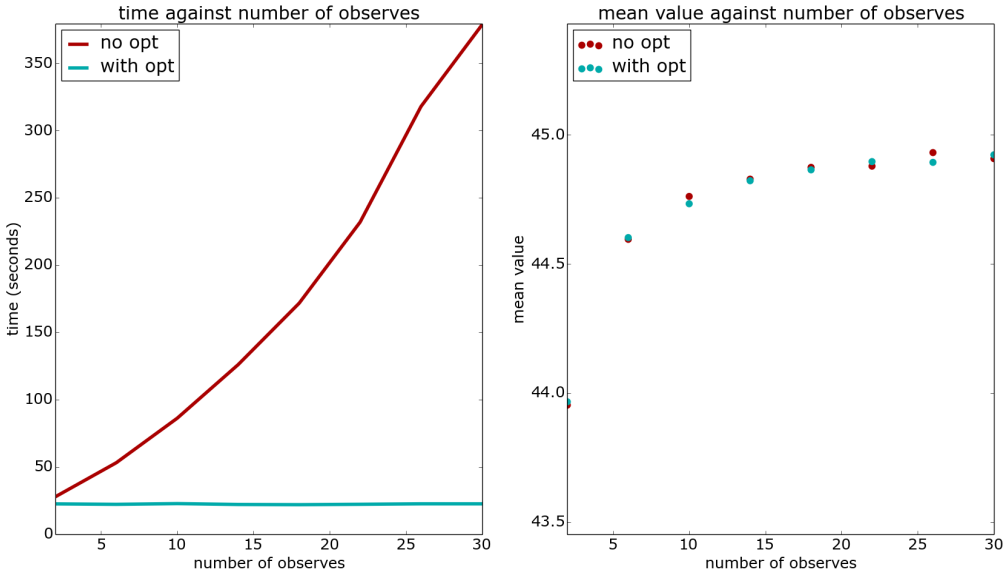
5.2 Merging observes of the same distribution

To test this optimization I used a program of the following form

```
[assume m (poisson 40)]
[assume b 20]
[observe (normal m b) 45]
...
[observe (normal m b) 45]
[predict m]
```

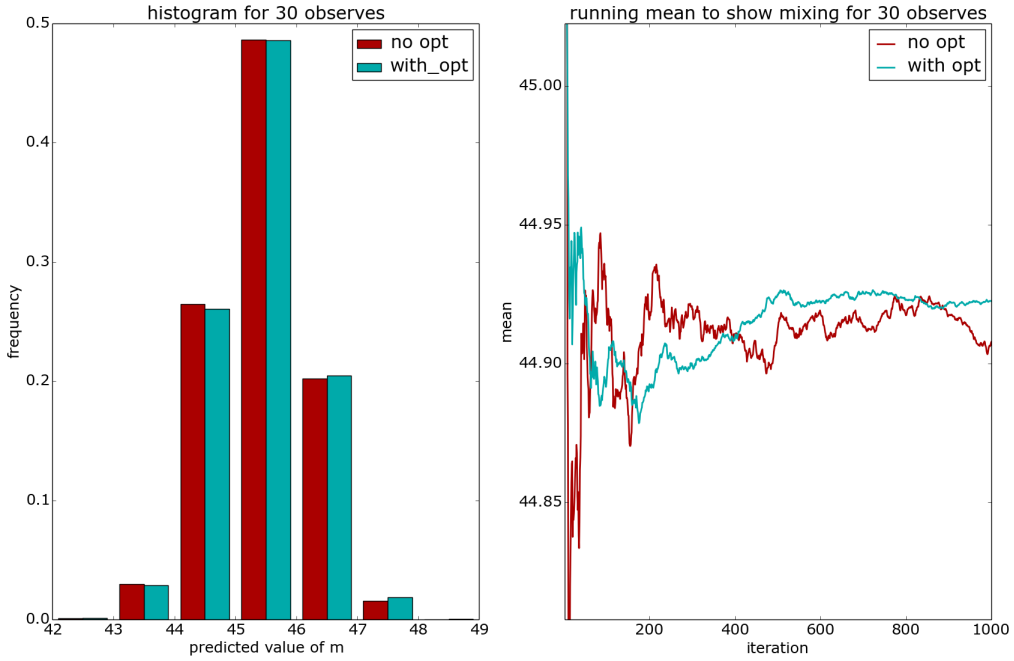
where the number of observe statements was varied. Programs of the above form were compiled with either no optimizations enabled or all optimizations enabled, then the resulting programs were run for 1000 iterations with 100 particles.

The following two graphs show running time against number of observes and mean value of the posterior distribution against number of observes.



What you can see from these is firstly that the optimized and non-optimized programs give the same answer, but that the optimized program is far faster. Considering the complexity, it appears that the non-optimized program is close to linear in the number of observes, however the optimized program running time remains constant.

I will now investigate the $n = 30$ case in more detail. These aspects can be seen in the following two graphs. The first graph shows the final posterior distribution. The second graph shows how the mean varies over time, so the value at t is the mean of the first t iterations.



What you see from the first graph is again that the optimized and non-optimized programs give the same answer. Looking at the second graph it appears that the optimized program reaches it's final answer after around 600 iterations whilst the non-optimized program has still not converged satisfactorily after 1000 iterations, so at first glance the optimized program has better mixing, however I don't feel this is conclusive and would instead say that they perform roughly equally.

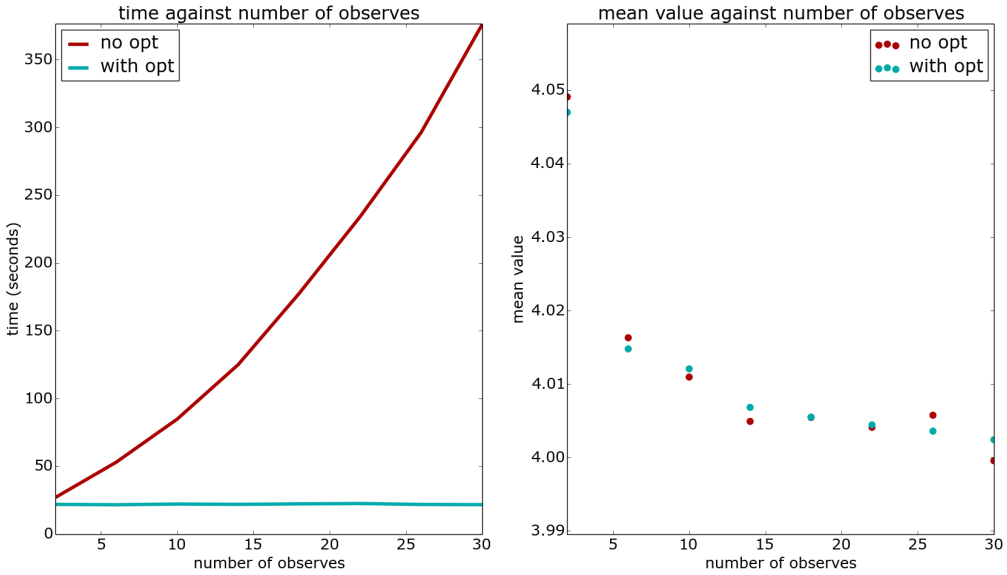
5.3 Merging any consecutive observes

To test this optimization I used a program of the following form

```
[assume m (normal 5 1)]
[assume f (lambda () -> Num m)]
[observe (normal (f) 0.1) 4]
...
[observe (normal (f) 0.1) 4]
[predict m]
```

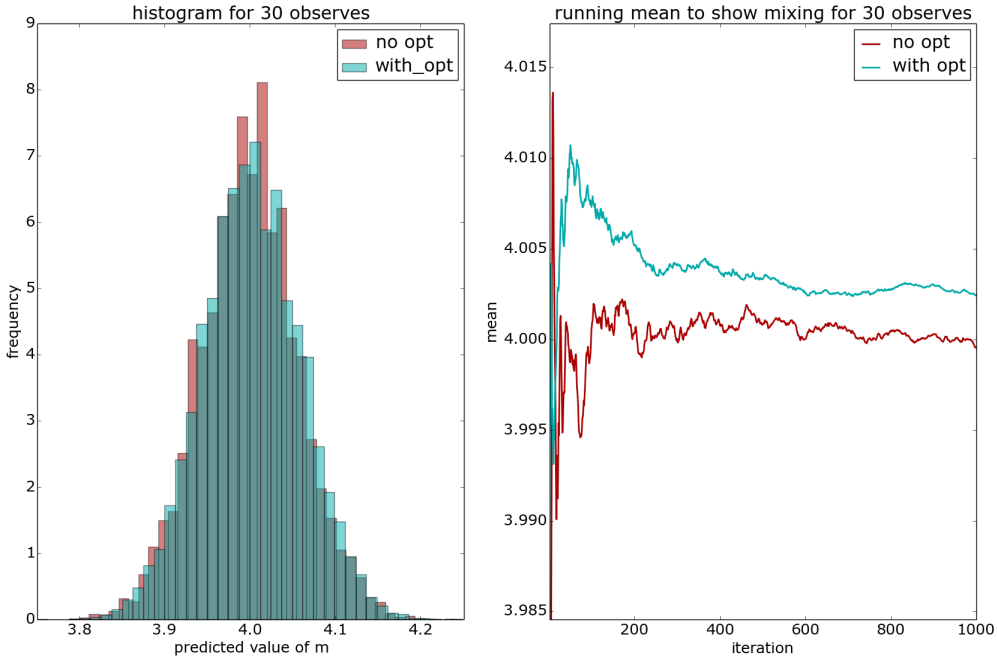
where the number of observe statements was varied. Programs of the above form were compiled with either no optimizations enabled or all optimizations enabled, then the resulting programs were run for 1000 iterations with 100 particles.

The following two graphs show running time against number of observes and mean value of the posterior distribution against number of observes.



What you can see from these is firstly that the optimized and non-optimized programs give the same answer to within an acceptable margin, but again that the optimized program is far faster. Considering the complexity, it again appears that the non-optimized program is close to linear in the number of observes, however the optimized program running time remains constant.

I will now investigate the $n = 30$ case in more detail. These aspects can be seen in the following two graphs. The first graph shows the final posterior distribution. The second graph shows how the mean varies over time, so the value at t is the mean of the first t iterations.



What you see from the first graph is again that minus a bit or noise the optimized and non-optimized programs give the same answer. Looking at the second graph they appear to have similar levels of mixing, that is they both approach their final mean value of posterior distribution at the same rate.

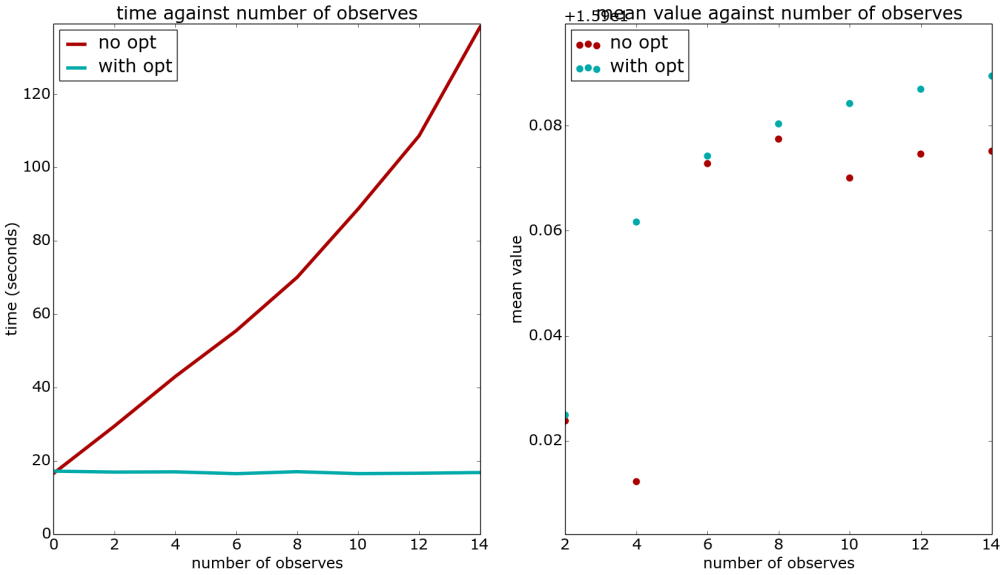
5.4 Removing observes

To test this optimization I used a program of the following form

```
[assume m (normal 10 4)]
[observe (normal m 0.1) 15]
...
[observe (normal m 0.1) 15]
[predict m]
```

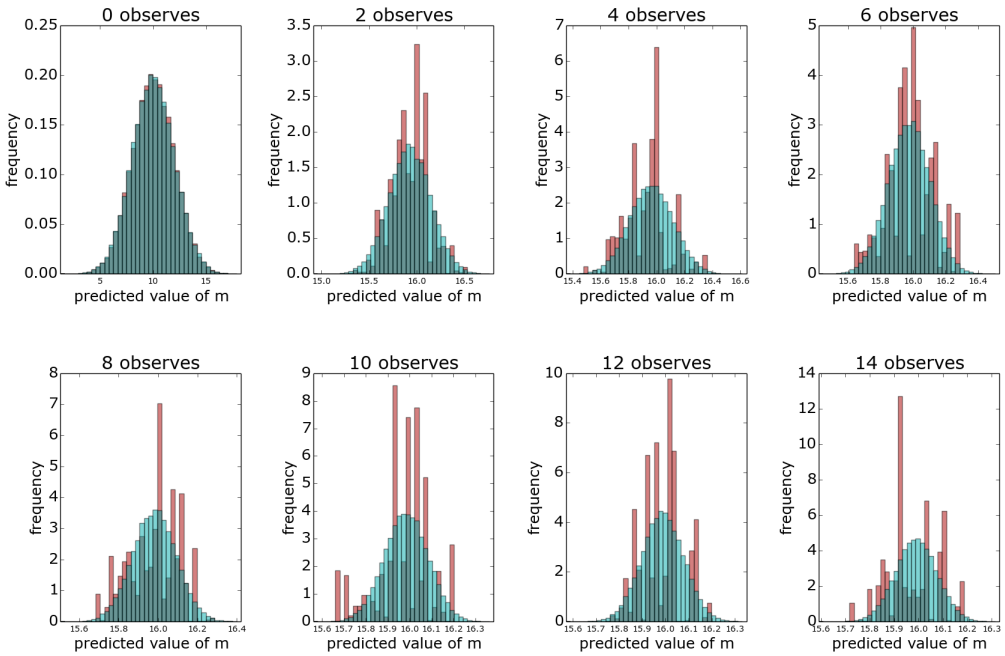
where the number of observe statements was varied. Programs of the above form were compiled with either no optimizations enabled or all optimizations enabled, then the resulting programs were run for 1000 iterations with 100 particles.

The following two graphs show running time against number of observes and mean value of the posterior distribution against number of observes.



What you can see here is firstly that the optimized program is far faster than the non-optimized one, and the optimized program gives a sensible and correct answer whereas the non-optimized programs are much less consistent. Considering complexity, it again appears that the non-optimized program is close to linear in the number of observes, however the optimized program running time remains constant.

I will now show the extend to which the non-optimized programs fail to give usable output, looking at the posterior distribution of each one and showing how it deteriorates as the number of observes increases. Consider the following set of graphs.



What you can see here is that the optimized program gives a perfect normal distribution every time, in fact we could have run far fewer than 1000 iterations and still achieved a good distribution. However, the non-optimized program's distributions are much less clean and if we were to increase the number of observes more or move the observed value further from the value `m` then the distributions would deteriorate even more. The non-optimized program is not even usable except in easy cases, so this optimization actually opens whole new areas of program that can be written rather than just making things faster.

5.5 Removing constant observes

To test this optimization I used the following program

```
[assume x (uniform-continuous 0 1)]
[observe (flip 0.5) true]
[predict x]
```

where the point is that the optimization would remove the observe and change nothing else. The above program was compiled with either no optimizations enabled or all optimizations enabled, then the resulting programs were run for 1000 iterations with 100 particles.

The result on speed was that the optimized program was approximately 27% faster. When counting the number of unique samples in an iteration, the non-optimized program had on average 47% unique samples, but the optimized program had on average 63% unique samples. What we see here is that removing a constant observe improves the program in all ways.