G2C: an optimizing transcompiler for probabilistic programming languages

Robert Brignull

1 Introduction

1.1 The purpose of the project

Talk about how there a many different probabilistic programming languages.

A lot are both functional and interpreted as this makes them easy to code in and easy to implement, but it lessens their performance.

Prob-C is the opposite, being based on C it is more difficult to code in but by being a compiled language and highly multi-threaded it has good performance.

The purpose of this project is to create a transcompiler from a functional probabilistic programming language into Prob-C.

If any optimizations can be done along the way, either probabilistic or not, then that is even better.

1.2 How this was achieved

Because both my source language and Prob-C handle sampling and observing in a very similar way, the bulk of the compiler is very standard.

It first transforms into continuation passing style, then adds closures to functions and hoists them to the top level, before outputting C code.

It performs some non-probabilistic optimizations such as identifier reassignment and constant expression calculation.

It performs probabilistic optimizations such as merging multiple samples from one distribution family into one sample, and potentially removing observes when it forms a conjugate prior.

2 Background

2.1 Probabilistic programming

A probabilistic programming language is just like a normal one except with the ability to easily draw from random distributions and to condition the program execution on other random variables.

Usually a probabilistic programming language is based on an existing language rather than being a completely new one, examples include: Venture (Scheme), Anglican (Scheme), IBAL (OCaml), Infer.NET (.NET), PSQL (SQL), FACTORIE (Scala), Alchemy (C++), Church (Scheme), Stan (none).

Probabilistic programming languages do their inference in one of a few different ways, including Markov Chain Monte Carlo (MCMC) and Sequential Monte Carlo (SMC) algorithms.

The applications of probabilistic programming are diverse, anything that requires estimating a distribution given prior beliefs and conditioned on some observed data. It can be used in machine learning, financial modeling, etc.

2.2 Anglican and Prob-c

Anglican and Prob-C are the two languages we'll be concerned with most, Anglican because it's what I've based my source language on, and Prob-C because it is the target language of my compiler.

Anglican is based on the Venture modeling language which is itself based on Scheme. Anglican is effectively one language embedded inside another, specifically it is build from a sequence of the commands Assume, Observe and Predict, each of which can contain expressions which come from a functional subset of Scheme.

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. 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.

2.3 Bayesian probability

Bayesian probability revolves around the idea of conditional probability, that is having some prior belief about an event and then estimating it's probability once you've observed that some other event happened.

The central theorem is Bayes' theorem $p(A|B) = \frac{p(A \wedge B)}{p(B)}$

3 The compiler

3.1 The source language

For this project I had to invent my own toy language, I chose to base it on Anglican but with some extra type information to make compilation easier.

** Small example program **

Without this extra type information, when translated to C all variables would have to be data-structures rather than native types, this would largely eliminate the benefit of transforming to a compiled language.

** BNF of G **

3.2 Typing the source language

Typing the source language is very easy, all the information is there and only minimal inference has to be done.

We know the types of simple values such as numbers and booleans.

For all built-in functions (eg. +, -, and, or) we know their type or in the case of = can give them one based on their argument types. Then we just check that the types of the arguments match the types of the formal parameters.

For if expressions we check that the test is a boolean type and that both of the branches have the same type.

For lambdas the user must specify the parameter types and return type, so we just check that the actual type of the body matches the return type they gave. We have to do a little bit of extra work to deal with recursive functions, this is the reason why function return types must be specified.

For an observe, check that the outer expression of the first argument is a probabilistic primitive. This is a restriction of the sampling method and hence of the language.

3.3 Making ids unique

At this point we change all ids to be unique, that is so that whenever two ids are the same then they refer to the same variable.

This means that for the rest of the compilation we do not have to worry about the scope of variables at all.

3.4 Into continuation passing style

The first step is to transform into continuation passing style.

The main difference is that now functions do not return values but instead take as another parameter a function to call when they're done with the value they would have returned.

The exception to this is built-in primitives which we assume can be done atomically by the target language and hence they can be calculated directly and stored into variables.

It is this step that is the defining transition from a functional to an imperative language.

3.5 Closure conversion and hoisting

What we want to do in this step is to lift all functions to the top level of the program, as they must be defined that way in a C program. The problem is that functions may reference variables defined outside of their scope.

Closure conversion concerns working out which are the variables used by a function are free variables and which are present in the arguments or defined by let expressions within that function.

The effect of this is almost to add more arguments to the function so that it has no free variables, in practice however we keep separate the original arguments and the new arguments are formed into one package, a closure or bundle.

Once this has been done all function definitions can be lifted to the top level of the program. Whenever a function would have been defined we instead instantiate a data structure containing the name of the function and the values of all variables it references formed into a closure.

3.6 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, only difficult point is to make sure all structs and functions are defined before they are described to allow full recursion.

One important point is we don't need to worry about memory management at all, which simplifies like tremendously. This is because each run of the program happens within its own thread and when it ends the operating system will release all memory it owns. Because each thread is short lived and doesn't use much memory individually, the memory usage of the program is low and does not grow over time.

4 Optimizations

All optimizations are performed immediately after CPS transformation.

4.1 Non-probabilistic optimizations

We do some standard non-probabilistic optimizations, including...

Constant expression calculation, including arithmetic and boolean expressions.

Merging multiple arithmetic expressions into one.

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

Removing trivial continuations.

4.2 Merging samples

If we have the situation

$$\begin{array}{l}[\mathsf{assume}\ x_1\ (\mathsf{normal}\ m_1\ b_1)]\\ \ldots\\ [\mathsf{assume}\ x_n\ (\mathsf{normal}\ m_n\ b_n)]\\ [\mathsf{assume}\ y\ (+\ x_1\ \ldots\ x_n)]\end{array}$$

and each x_i is not used anywhere else, then we can simplify it to just

$$[\mathsf{assume}\ y\ (\mathsf{normal}\ (+\ m_1\ \ldots m_n)\ (+\ b_1 \ldots b_n))]$$

Similar things can be done with other distributions.

4.3 Removal of observe statements

Here we start doing some actual probability in an attempt to swap sample and observe statements and hopefully remove the observe completely.

The term for what we're doing is conjugate prior.

If we have the situation

$$\begin{array}{l}[\mathsf{assume}\ p\ (\mathsf{beta}\ a\ b)]\\ [\mathsf{observe}\ (\mathsf{flip}\ p)\ \mathsf{true}]\end{array}$$

then it can be simplified to

$$[\mathsf{assume}\ p\ (\mathsf{beta}\ (+\ a\ 1)\ b)]$$

This is the easy case where a and b are constants, if they are not then the observe remains, but we might reduce 100 observes to just one so it will hopefully improve performance and definitely improve the quality of the samples generated.