Verified Density Compilation for a Probabilistic Programming Language

JOSEPH TASSAROTTI, NYU, USA
JEAN-BAPTISTE TRISTAN, AWS, USA

This paper presents ProbCompCert, a compiler for a subset of the Stan probabilistic programming language (PPL), in which several key compiler passes have been formally verified using the Coq proof assistant. Because of the probabilistic nature of PPLs, bugs in their compilers can be difficult to detect and fix, making verification an interesting possibility. However, proving correctness of PPL compilation requires new techniques because certain transformations performed by compilers for PPLs are quite different from other kinds of languages. This paper describes techniques for verifying such transformations and their application in ProbCompCert. In the course of verifying ProbCompCert, we found an error in the Stan language reference manual related to the semantics and implementation of a key language construct.

CCS Concepts: • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Compilers**; • **Mathematics of computing** → *Markov-chain Monte Carlo methods.*

Additional Key Words and Phrases: compilers, probabilistic programming, formal verification

ACM Reference Format:
Joseph Tassarotti and Jean-Baptiste Tristan. 2023. Verified Density Compilation for a Probabilistic Programming Language. *Proc. ACM Program. Lang.* 7, PLDI, Article 131 (June 2023), 23 pages. https://doi.org/10.1145/3591245

1 INTRODUCTION

Probabilistic programming languages (PPLs) help users perform statistical analyses of data. A data analyst uses the language to write a program that describes a statistical model for a data set. A compiler then generates code from this model that can make statistical inferences about the model's unknown parameters. The compiler's task is often quite different from a conventional compiler because the purpose of the generated code is not to execute the model description directly. Instead, the compiler translates the program into functions that are called by a statistical inference algorithm. As a result, compilers for probabilistic programming languages perform transformations that would appear invalid for a traditional compiler to do, but which are nevertheless correct because of how the generated code is used. These differences lead to new challenges in implementing and debugging compilers for PPLs.

To describe these challenges more concretely, we consider the Stan probabilistic programming language [Carpenter et al. 2017]. Stan is one of the most widely used PPLs, with widespread applications in both academia and industry across multiple scientific fields. A simple example Stan program for analyzing flips of a biased coin is shown in Figure 1a. Stan programs are structured into *blocks*. First, the `data` block (lines 1–3) describes the data set to be analyzed, which in this example is just `flips`, an array of integer values giving the outcomes of the coin flips, represented as 0s or 1s. Next, the `parameter` block specifies the unknown variables which the analyst wants

Authors' addresses: Joseph Tassarotti, jt4767@nyu.edu, NYU, New York, USA; Jean-Baptiste Tristan, trjohnb@amazon.com, AWS, Boston, USA.

2475-1421/2023/6-ART131
https://doi.org/10.1145/3591245

to make inferences about. Here, the sole parameter is `mu`, declared on line 5, which represents the probability that a flip of the biased coin returns 1. This variable is a floating point value, indicated by the type `real` in its declaration. The declaration further specifies a *constraint* of the form `<lower=0.0,upper=1.0>` specifying that the value of `mu` must lie in the interval $(0, 1)$. As in this example, Stan allows data variables to be discrete integer values or continuous floating point values (or arrays of such values), while parameters are required to be continuous variables.

Finally, the core of the program is the `model` block, which describes how to compute the probability of observing an assignment of values for the parameters and data. In this case, the analyst first states that they have a *prior* belief about the distribution that the `mu` parameter was drawn from, here expressed on line 8 using the sampling statement $\sim$. We read `mu ~ uniform(0,1)` as saying that `mu` is sampled from the uniform distribution on the interval $(0, 1)$. Then, the `for` loop on lines 9-11 postulates that each coin flip was generated by sampling from a Bernoulli distribution, using `mu` as the probability of returning a 1.

Although the `model` block here looks as though it is describing how to randomly generate the parameters and data using the $\sim$ sampling statement, in actuality, the `model` block is specifying how to compute a probability density function.[1] Stan allows users to write the `model` block in an alternate style where this computation is made more explicit. An equivalent version of the same program in this alternate style is shown in Figure 1b. In place of the sampling statement, this version increments a special floating point variable called `target`. This variable is implicitly initialized to 0 and accumulates the logarithm of the model's probability density value. The code adds to `target` the log probability density of `mu` (lines 8–9) and `flips[i]` (lines 11–12) according to the probability functions for the uniform and Bernoulli distributions, respectively. At the conclusion of the block, this `target` variable is implicitly returned. In fact, the $\sim$ sampling statement used in Figure 1a is just syntactic sugar that translates to incrementing the `target` variable as in Figure 1b.

Once the user has specified a model, the Stan compiler supports several different types of statistical analyses. The main supported analysis is to do Bayesian inference. Specifically, in this example, the analyst had a prior belief that the parameter `mu` is equally likely to be any value in $(0, 1)$. After observing some data about the coin flips, the Bayesian paradigm specifies that according to Bayes' rule, the analyst should *update* their priors to a different distribution called the *posterior distribution*. In general, calculating the posterior distribution is intractable. Instead, the compiler for Stan generates code that draws samples that approximate the posterior. By generating enough samples, the analyst can form an approximate summary of the posterior distribution.

How are the samples generated? Stan, like many other PPLs, uses a Markov chain Monte Carlo (MCMC) algorithm. Figure 2 represents schematically the components of the implementation of these MCMC algorithms. The core "runtime" of the language is a loop that iterates steps of a Markov chain. The states of this Markov chain are assignments of values for the parameter variables of the model. At each iteration, the runtime queries a *proposal generator*, generated by the compiler, which randomly suggests a new state for the chain to potentially move to. This new state is a set of values for the parameter variables called a *proposal candidate*. The runtime calculates the probability density of these proposed parameter values and the data. This calculation is done by a *density function*, which is produced by the compiler. The runtime similarly calculates the probability for the parameter values of the current state of the chain. The ratio of the proposed and current probabilities is called the "acceptance ratio". The runtime then randomly either transitions to the proposed state, called "accepting" the proposal, or stays at its current state, "rejecting" the proposal, where the probability of acceptance is calculated using the acceptance ratio.

[1]We sometimes write "probability" when it is clear from context that we mean a "probability density" or "probability mass".

```
data {
  int flips[100];
}
parameters {
  real<lower=0.0,upper=1.0> mu;
}
model {
  mu ~ uniform(0,1);
  for (i in 1:100) {
    flips[i] ~ bernoulli(mu);
  }
}
```

(a) Coin flip model written in generative style.

```
data {
  int flips[100];
}
parameters {
  real<lower=0.0,upper=1.0> mu;
}
model {
  target +=
    uniform_lpdf(mu | 0,1);
  for (i in 1:100) {
    target +=
      bernoulli_lpmf(flips[i] | mu);
  }
}
```

(b) Equivalent model written in non-generative style.

Fig. 1. Two equivalent Stan programs for analyzing coin flips. The `data` block describes the data used to do inference, while the `parameter` block describes the unknown variables that the analyst wishes to make inferences about. The `model` block defines the logarithm of the probability density function of the model.

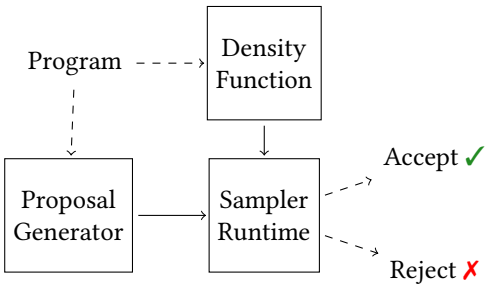

Fig. 2. Schematic overview of components of an MCMC sampler. The source program is compiled into a proposal generator and density function evaluator. The runtime queries these two components to determine whether to probabilistically accept or reject states.

The states that the Markov chain passes through as this algorithm is run are returned to the user as a list of generated samples. At a high level, the guarantee provided by this algorithm is that in the limit, as more and more samples are drawn, the distribution of the generated samples converges to the posterior distribution.

That is the intention, at least. In practice, the compiler and runtime can have bugs that lead to violations of this guarantee, thus subtly biasing statistical analyses. In this set up, both the proposal generator and the density function calculation use code emitted by the compiler that is derived from the model. Both of these generated components have different requirements in order for the overall sampling algorithm to function correctly. For example, if a parameter value is in the *support* of the posterior distribution, meaning it has some non-zero probability of occurring, then the proposal generator must be able to generate that state with non-zero probability. Meanwhile, the code generated to compute the density function must correctly calculate these densities. Doing so is made complicated by the fact that the compiler performs transformations to the density function, both to optimize calculations and to make the task of the proposal generator easier. For instance, Stan's density compilation performs a *reparameterization* that removes constraints on parameter variables, such as the `lower=0, upper=1` in the example from Figure 1a, so that the proposal generator does not have to construct proposals satisfying these constraints.

When bugs occur in compilation, they can be difficult to diagnose. First, even for simple examples, the true posterior distribution is difficult to compute, so it is hard for a user to notice that there is an error. Second, since the guarantee provided by the MCMC algorithm is about convergence

to the true distribution *in the limit*, even when the user suspects there is something wrong, they have trouble telling whether there is a bug or they just need to run the chain for more steps. As a result, bugs can go undetected for a long time. For example, Stan versions 2.10–2.13 had a bug that caused small biases in statistical estimates [Becker 2016]. Even though this bug manifested on almost *every* program, the effect was small enough in most cases that it took more than 6 months for someone to notice that there was a systematic bias.

Formal verification for probabilistic programming. The difficulty of implementing and debugging compilers for probabilistic programming languages makes formal verification of such compilers an interesting possibility. Although the semantics of these languages and their compilation are quite different from those of a conventional language, it should still be possible to prove that the compiler *preserves* the semantics of compiled programs, just as the CompCert [Leroy 2009] and CakeML [Kumar et al. 2014] verified compilers do for C and CakeML, respectively.

This paper describes the formal verification of an important part of the compilation pipeline for a PPL. We introduce ProbCompCert, a new compiler for a subset of the Stan programming language written in Coq. ProbCompCert compiles Stan programs to C and connects to CompCert for compilation down to assembly code. We have verified the correctness of three key passes performed by the compiler as part of density compilation. Specifically, these are the compilation passes that perform transformations whose correctness relies on the probabilistic semantics of the language. That is, they change the code in ways that would appear incorrect for a traditional compiler to do, but which are necessary in order for the generated code to lead to correct and efficient sampling behavior.

Our proof shows that these passes of the density compiler are semantics preserving with respect to a new mechanized semantics of Stan. Following the structure of the pencil-and-paper formal semantics proposed by Gorinova et al. [2019], this semantics is split into two parts. First, the *operational* layer gives a small-step description of how expressions and statements are executed, similar in style to CompCert's semantics of C. Next, the *denotational* layer maps programs to probability distributions over parameter values, obtained as an integral of the density function. Semantics preservation then says that this probability distribution is unchanged by compilation.

ProbCompCert's proofs of semantics preservation follow this same two layer structure. For each pass, we first prove a simulation that relates the operational behavior of the input of the pass to its output. These proofs use the forward simulation proof technique used by CompCert. However, in CompCert, simulations essentially show that the exact input/output behavior of compiled code is preserved. In contrast, our simulations typically show that the input/output behavior of the code is transformed according to some mathematical function. In the second part of each proof, we show that this transform preserves the probability measure defined by the program. This latter proof typically exploits some property of integrals, such as a change-of-variables formula or linearity.

In the course of verifying one of these passes, we discovered an error in the Stan language reference manual. The manual claimed that Stan's optimized implementation of the "~" statement had equivalent behavior to writing a `target+=` statement of a particular form. In fact, this equivalence only holds under certain conditions. ProbCompCert implements checks to use this optimized implementation only in cases where the equivalence holds.

In summary, the contributions of this paper are as follows:

- A mechanized semantics for a subset of the Stan programming language that models features not considered in prior work, such as constraints, that play an important role in compilation.
- A methodology for verifying compiler passes for which the justification of correctness relies on the probabilistic semantics of the language.

- A compiler design and implementation for Stan that is structured as a series of small, well-defined passes to make it amenable to verification using this methodology.

Although ProbCompCert handles a large enough subset of the Stan language to cover a range of examples adapted from the Stan user manual, it has limitations. We have focused on features whose compilation is quite different from things seen in other compilers and that therefore pose new verification challenges. Consequently, ProbCompCert does not yet support vectorized operations, nor nested arrays. The current version also does not support all forms of constraints. Furthermore, the proposal generator is a simple Gaussian random walk, which has much lower sample efficiency than Stan's more complex NUTS sampler [Hoffman and Gelman 2014]. Finally, the proposal generator and the runtime are not verified. It would be interesting future work to verify these components and integrate their proofs.

The remainder of the paper is structured as follows. First, we give an overview of the compiler's structure and the passes of the density compiler (§2). Next, we describe the formal semantics of the subset of Stan supported by ProbCompCert (§3). From there, we explain the specification of semantics preservation used by ProbCompCert and describe our methodology for proving semantics preservation for compiler passes (§4). The next two sections describe how this methodology is applied to prove the correctness for two important passes (§5 and §6). Our evaluation (§7) compares the performance of ProbCompCert's generated code with that of Stan's official compiler and discusses the relative effort of different parts of verification. Finally, §8 describes related work in the implementation and semantics of probabilistic programming languages.

2 SYSTEM OVERVIEW

```
1  state = load_initial_params();
2  for (int i = 0; i < num_samples; ++i) {
3    propose(state,candidate);
4    lp_candidate = model(data,candidate);
5    lp_state = model(data,state);
6    lu = log(rand_uniform(0, 1));
7
8    if (lu <= lp_candidate - lp_state) {
9      copy_params(state,candidate);
10   }
11   output_sample(state);
12 }
```

Fig. 3. Code for symmetric Metropolis-Hastings.

As described in §1 (illustrated in Figure 2), ProbCompCert is divided into three components: runtime, proposal generator compiler, and density compiler. Although the focus of this paper is the verified passes of the density compiler, we give an overview here of the entire system to provide context for the design choices and verification challenges in the density compiler.

2.1 Runtime

The main task of the runtime is to drive the Markov chain Monte Carlo algorithm that generates samples. ProbCompCert uses an MCMC algorithm called symmetric Metropolis-Hastings.[2] The core of the algorithm is the loop described at a high level in the introduction.

A simplified C implementation of this loop is shown in Figure 3. In this code, the current state of the Markov chain is stored in `state`. Each iteration of the loop begins by querying the proposal generator to suggest a new state which is stored in `candidate`. Next, the probability density of the candidate and current state are computed by calling the `model` function generated by the

[2]The official Stan compiler's MCMC algorithms are Hamiltonian Monte Carlo [Betancourt 2018] and No U-Turn Sampling [Hoffman and Gelman 2014], which are instances of *asymmetric* Metropolis-Hastings and use a much more sophisticated proposal generator than ProbCompCert. However, the role of the density compiler—the subject of verification in ProbCompCert—is similar in ProbCompCert and these more sophisticated MCMC algorithms.

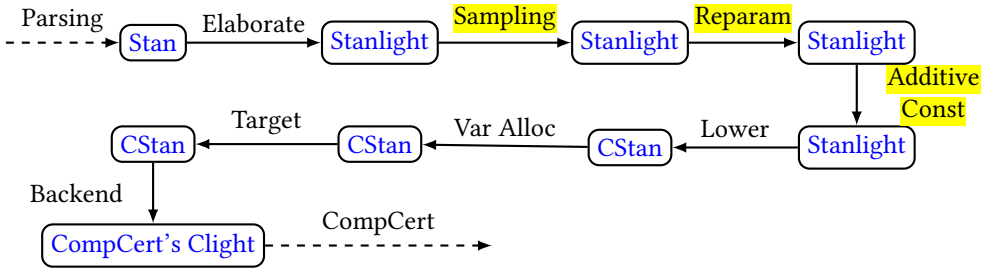

Fig. 4. Density compilation pipeline for ProbCompCert. The three highlighted passes are verified.

density compiler. An optimization that both Stan and ProbCompCert use is to actually compute the *logarithm* of these densities, hence the `lp` in the prefix of the corresponding variables. Next, a random value is sampled from the uniform distribution on the interval $(0, 1)$, and its logarithm is stored in `lu`.

If `lu` is smaller than `lp_candidate - lp_state`, the candidate is said to be *accepted* and becomes the new state of the chain, carried out by copying `candidate` to `state`. Regardless of whether the candidate is accepted or not, the state at the end of the iteration is output to the user as a sample.

2.2 Proposal Generator Compiler

The precise properties required of the proposal generator to ensure overall correctness of the algorithm are technically involved and beyond the scope of this paper. However, one complicating factor in designing a proposal generator that will be relevant in what follows is that, in Stan, parameters can have *constraints*, such as the `<lower=0.0,upper=1.0>` constraint on `mu` in the example from Figure 1a. Respecting these constraints would require that the parameters generated only have values in the specified range.

To handle this issue, both the Stan reference compiler and ProbCompCert perform a *reparameterization* transform to the model block, effectively removing the constraints, so that the proposal generator and runtime operate over an *unconstrained* parameter space. The proposal generator used by ProbCompCert is a simple Gaussian random walk proposal. To generate a candidate, it samples an independent Gaussian with mean 0 and standard deviation 1 for each parameter value, and adds it to the corresponding parameter's value in the current state.

2.3 Density Compiler

The density compiler translates the model block into executable code to compute the densities needed by the runtime's MCMC loop. The compilation pipeline used by ProbCompCert's density compiler is shown in Figure 4, with the three verified passes highlighted. These passes are particularly interesting because they involve explicit reasoning about probabilistic semantics.

Compilation proceeds through three intermediate representations:

- *Stanlight*: This IR represents the subset of Stan supported by the compiler. It is produced by unverified OCaml code that performs a simple elaboration pass to add type information to the parsed Stan program. Stanlight supports Stan's constraint annotations on parameters, the "~" sampling statement we saw in the model from Figure 1, and treats arrays as a first-class primitive. §3 describes the syntax and semantics in more detail.

- *CStan*: This lower-level representation no longer has several Stan specific features from Stanlight: all parameters are unconstrained, sampling statements are replaced by probability density calculations, and array indexing is replaced by pointer offsetting.
- *Clight*: This is one of the C-like internal representation used by CompCert.

The translation between each of these internal representations is done through several smaller intermediate steps that simplify or compile away some of the features that are not found in later stages. A high-level description of the three verified passes is as follows:

- *Sampling*: The underlying semantics of the "~" statement in Stan is to increment the `target` variable that we saw in Figure 1b. This pass compiles away the "~" sampling statement by replacing it with an explicit increment to the `target` variable.
- *Reparameterization*: As alluded to in the previous subsection, this pass simplifies the proposal generator's task by allowing the parameters passed to the `model` block to be *unconstrained*. To do so, it replaces all uses of a constrained parameter in the `model` with a call to a constraint function that remaps the unconstrained representation used by the proposal generator into a constrained value. Taken alone, this remapping would affect the probability distribution that the model block represents. To adjust for this, the pass also adds a correction term called a Jacobian to the `target` variable.
- *Additive Constants*: In §2.1 we saw that after calculating the log probabilities of the candidate and current states, the runtime only performs a comparison involving the *difference* of these two probabilities, `lp_candidate - lp_state`. In other words, the magnitude of the density function does not matter, only the differences between the densities assigned to different parameter values. This means it is sound for the compiler to *shift* the probabilities computed by the model block up to addition by a constant. This pass exploits this fact to optimize code by dropping addition of constants to the `target` variable. For example, it can remove certain statements of the form `target += c` when `c` is a constant, and similarly simplifies arithmetic expressions to remove such additions.

Two of these passes, Reparameterization and Additive Constants, are particularly interesting because they do *not* preserve the input/output behavior of the density code. That is, these passes perform transformations that would be invalid for a typical compiler to do, but are correct in this context because of the probabilistic meaning of Stan programs. In the next section, we describe a formal semantics that justifies the correctness of these transformations.

3 SEMANTICS OF STANLIGHT

This section describes the syntax and semantics of Stanlight, the subset of the Stan programming language supported by the density compiler. The semantics is divided up into two layers: an operational layer and a denotational layer.

3.1 Operational Layer

The operational layer of the semantics specifies how to execute the code in the `model` block so as to compute the log density of a given data and parameter assignment. The semantics is specified in a small-step style, similar to the small step semantics of CompCert's Clight internal language. Figure 5 presents the syntax for expressions, statements, and other syntactic categories used in the semantics. We write an overhead arrow $\vec{\ }$ to indicate lists of a corresponding syntactic category.

Most expressions and statements have behavior analogous to C-like equivalents, with a few exceptions. We have already discussed how the "~" statement implicitly increments the `target` variable. This `target` variable cannot be set directly, and can only be modified through the "~" statement and through an addition statement of the form `target += e`. Although this `+=` syntax

Values (*Val*)

$v ::=$	i	integer
	$\mid r$	float
	$\mid \texttt{ptr}(b, \mathit{off})$	pointers
	$\mid \texttt{undef}$	undefined
	$\mid \cdots$	

Expressions (*Expr*)

$e ::=$	i	integer const.
	$\mid r$	float const.
	$\mid id$	variable access
	$\mid e[e]$	array access
	$\mid \texttt{call}(e_1, \vec{e})$	math function call
	$\mid op(\vec{e})$	arithmetic operation
	$\mid \texttt{target}$	read target value

Constraints (*Constraint*)

$c ::=$	$\texttt{none}$	no constraint
	$\mid \texttt{lower} = r$	
	$\mid \texttt{upper} = r$	
	$\mid \texttt{lower} = r_1, \texttt{upper} = r_2$	

Types (*Typ*)

$\tau ::=$	$\texttt{int}$	
	$\mid \texttt{real}$	
	$\mid \texttt{array}(\tau, \mathit{size})$	

Statements (*Stmt*)

$s ::=$	$\texttt{skip}$	
	$\mid e_1 = e_2$	assignment
	$\mid s_1; s_2$	sequencing
	$\mid \texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$	
	$\mid \texttt{for}(id\ \texttt{in}\ e_1 : e_2)\ \{\ s\ \}$	
	$\mid \texttt{target} \mathrel{+}= e$	target add
	$\mid e \sim f(\vec{e})$	sampling

Programs (*SProg*)

$p ::= \{$
  $\texttt{functions} : \ldots;$
  $\texttt{vars} : \ldots;$
  $\texttt{data\_ids} : \mathit{List}(\mathit{Id});$
  $\texttt{param\_ids} : \mathit{List}(\mathit{Id} \times (\mathit{Expr} \to \mathit{Expr}));$
$\}$

Fig. 5. Syntax of Stanlight

resembles the syntax for adding an expression to the current value of a variable in C-like languages, it is treated as a distinct statement in Stanlight, as in Stan.

A complete program p is a record consisting of (1) function definitions and their signatures, (2) a list of variables with their types and constraints (if any), (3) a list of ids for data variables, and (4) a list of ids for parameters as well as *output* functions that map the unconstrained internal representation of a parameter to a (possibly) constrained representation expected by the user.

During execution, a program state σ has one of three following forms:

1. $\texttt{Start}\ f\ s\ t\ \mathit{env}\ m\ \mathit{pm}$ 2. $\texttt{Running}\ f\ s\ t\ k\ \mathit{env}\ m\ \mathit{pm}$ 3. $\texttt{Return}\ t$

where f is the declaration of the `model` function and its signature, s is the current statement being executed, t is the value of the special `target` variable, m is the global memory, *env* is the model block's local variable environment, *pm* is a map storing the values of each parameter variable for which the density is being calculated, and k is a stack of continuations.

The global memory m uses CompCert's memory model, which supports pointer-based indexing. However at this stage its primary role is for storing external functions and the value of fixed global variables, such as data variables. Local environments *env* and parameter mappings *pm*, on the other hand, are indexed directly by an identifier and offset and do not support pointer-based accesses.

Given lists of values d and γ, $\mathsf{initial}(p, d, \gamma, \sigma)$ holds when σ is of the form $\texttt{Start}\ f\ s\ t\ \mathit{env}\ m\ \mathit{pm}$, with (1) s equal to the `model` block's body; (2) $t = 0$; and (3) m and *pm* are initialized with d and γ loaded for the values of `data` and `parameter` variables. These variables are assigned according to the order their ids are listed in the `data_ids` and `param_ids` fields of the program.

Finally, the relation $\sigma \longrightarrow \sigma'$ holds when σ can take a single step to σ'. We omit most rules for this relation, as they are similar to rules for analogous statements in CompCert's C-like intermediate

representations. One exception is the following rule for the sampling statement ~ in Stanlight:[3]

$$\frac{e \Downarrow v \qquad \vec{e_a} \Downarrow \vec{v_a} \qquad \text{external\_call}(g, v :: \vec{v_a}, m, r)}{\texttt{Running}\ f\ (e \sim g(\vec{e_a}))\ t\ k\ env\ m\ pm \longrightarrow \texttt{Running}\ f\ \texttt{skip}\ (r + t)\ k\ env\ m\ pm}$$

Executing the statement $e \sim g(\vec{e_a})$ first evaluates e and each expression in $\vec{e_a}$, and passes the resulting values as arguments to the density function g. If the return value of the external call to g is a float r, the value of the `target` variable is updated from t to $r + t$.

Stanlight programs can have undefined behavior, represented by reaching a non-`Return` state that cannot take a step. Non-termination is another source of undefined behavior. The reason this is made undefined is that if execution of the `model` were to diverge during the MCMC loop, the program would be unable to generate further samples, so it is unclear how to assign semantics to such a program. Given lists d and γ, we say that is_safe(p, d, γ) holds if p does not get stuck and terminates when run from any state σ satisfying initial(p, d, γ, σ).

The step relation $\longrightarrow$ in Stanlight is deterministic. This is because, although the program describes a probabilistic model, the *calculation* of probability densities performed by the `model` block itself does not involve randomness.

3.2 Denotational Layer

The operational semantics described in the previous subsection defines evaluation of the `model` block as a state machine: an assignment of parameter and data values is set in the initial state, the system takes some number of steps, and (potentially) terminates, returning the `target` variable value. The denotational layer uses this state machine to define a probability distribution on parameter values.

The first step is to derive a density *function* from the state machine. Given a program p, we define a function logdensity$(p) : \mathit{List\ Val} \times \mathit{List\ Val} \longrightarrow \mathbb{R}$, where logdensity$(p, d, \gamma)$ would return the log density calculated by the `model` block when run on an initial state with data d and parameters γ. Once the density function is defined, we want to obtain a probability measure on parameter values by taking an integral over the density. However, there are two major difficulties in defining the density function and this integral that must be resolved.

Floating points and reals. The first issue is that the operational semantics (and the compiled code) uses IEEE floating point values to represent parameters and target values, not real numbers. Yet the Stan reference manual and the compilation passes that Stan and ProbCompCert perform appeal to an idealized semantics in which the probability measures defined by programs are obtained by integrating over mathematical real numbers.

How can we reconcile this? One approach would be to confront the fact that calculations are done with floating points and carefully track the effect of round-off errors in translations done by the compiler and the rest of the MCMC. There has been some prior theoretical analysis of how rounding errors can affect MCMC convergence [Roberts et al. 1998], but those results do not cover the kinds of transformations performed by Stan.

The solution we adopt instead is pragmatic: in defining the denotational semantics and proving correctness of passes, we treat these floating point numbers *as if* they behaved like real numbers and all computations were exact.

Specifically, the Coq development postulates two coercion functions IFR : `float` $\longrightarrow \mathbb{R}$ and IRF : $\mathbb{R} \longrightarrow$ `float`, along with axioms stating that they are inverses that commute with floating point arithmetic operations. Figure 6 contains a subset of these axioms. In the following, we omit writing these coercions when it is clear from context that they are needed.

[3]The presentation of the rule here is simplified to omit looking up the function g in CompCert's environment model.

```
Axiom IFR_IRF_inv :
  forall x, IFR (IRF x) = x.
Axiom IRF_IFR_inv :
  forall x, IRF (IFR x) = x.
Axiom float_add_irf: forall a b,
  (Float.add (IRF a) (IRF b))
     = IRF (a + b).
```

Fig. 6. Axioms for treating floating point calculations as if they were exact real arithmetic.

Assuming the existence of such a pair of coercions may seem problematic, as these axioms are not true. On the other hand, as we mentioned, several of the transformations that Stan and other compilers for probabilistic programming languages perform in practice are only semantics preserving if one assumes that the program is doing computations with exact reals. Thus, to prove properties of these compilation strategies, one has to express a similar assumption at some point in the proof.

One may wonder whether admitting such axioms undermines the guarantees of formal verification. As a precaution, we isolate these axioms to a specific module and only use them when proving passes that would require treating floating point arithmetic as if it were exact. Moreover, CompCert is structured so that the `float` type is generally treated opaquely beyond a carefully delineated API. Thus, it is unlikely one would inadvertently exploit a contradiction from these axioms in the course of a proof.

Nevertheless, these axioms *do* reduce the implied guarantees of ProbCompCert's correctness proof. In particular, it means that verification in ProbCompCert will not rule out bugs related to numerical stability or round-off error. However, verification *will* prevent bugs or miscompilations that are invalid with respect to this idealized, exact real arithmetic model. In the past, bugs in Stan, such as the one described in the introduction, and an issue we will discuss in §6, have occurred at this level and thus would be caught with ProbCompCert's verification approach.

Measure spaces and theory of integration. The second issue in setting up the denotational semantics is the choice of what theory of integration should be used. In the modern measure-theoretic approach to probability theory, probability distributions are defined on the so-called measurable subsets of a measure space, and Lebesgue integration is used to derive distributions from density functions.

Fortunately, certain restrictions in the Stan language mean that we do not require the full flexibility of measure theory and Lebesgue integration. In particular, in Stan, parameters must be reals (or arrays of reals), so there is no mixture of continuous and discrete parameters or more complicated measure spaces that we need to define measures on.

This means that for purposes of defining the denotational semantics of programs, we do not need to define a measure space on a program's parameters or assign probabilities to all measurable subsets of that space. Instead, we only define the probabilities of (open) *rectangular subsets* of the parameter space, which can be done using improper Riemann integration. We call a set $D \subseteq \mathbb{R}^n$ rectangular if it can be written as the Cartesian product of open intervals. For example, for a program with two unconstrained parameters, the denotational semantics will assign probabilities to all rectangular subsets of the form $(a_1, b_1) \times (a_2, b_2)$, where the end-points of the intervals may also be $+\infty$ and $-\infty$.[4]

Using Riemann integration simplifies the machine-checked proofs, as we can reuse results about Riemann integration developed in Coq in the Coquelicot library [Boldo et al. 2015].

Defining the probability measure. With these design choices in place, we first define a predicate

$$\mathsf{returns\_target}(p, d, \gamma, t) = \exists \sigma.\, \mathsf{initial}(p, d, \gamma, t) \wedge \sigma \rightarrow^* (\mathtt{Return}\ t)$$

[4]In some sense, there is no harm in only assigning probabilities to these subsets, since the Carathéodory extension theorem implies that this can be (uniquely) extended to a measure on all Borel subsets of the parameter space.

indicating whether a program will return a log density value of t for a given assignment of data and parameter values. This is a partial relation, in the sense that there may not be a t such that returns_target(p, d, γ, t) holds. We use Hilbert's epsilon operator, and the fact that execution is deterministic, to convert this to a function get_target(p, d, γ). This function has the property that if there exists a t such that returns_target(p, d, γ, t) holds, then get_target$(p, d, \gamma) = t$. Otherwise, if no such t exists, then get_target$(p, d, \gamma) = 0$. Observe that when no such t exists, that means the program triggers undefined behavior at the operational layer, in the sense we described in the previous section, so we are licensed to assign this arbitrary value of 0. We convert the output of this function to a real using IFR to obtain logdensity$(p, d, \gamma) = \mathsf{IFR}(\text{get_target}(p, d, \gamma))$.

To obtain the intervals that each parameter variable can range over, Stanlight constraints are mapped into open interval subsets of $\mathbb{R}$ as follows:

$$\text{interval\_of\_constraint}(c) = \begin{cases} (-\infty, \infty) & c = \mathtt{none} \\ (\mathsf{IFR}(a), \infty) & c = (\mathtt{lower} = a) \\ (-\infty, \mathsf{IFR}(b)) & c = (\mathtt{upper} = b) \\ (\mathsf{IFR}(a), \mathsf{IFR}(b)) & c = (\mathtt{lower} = a, \mathtt{upper} = b) \end{cases}$$

By applying this function to each element of the list of parameter constraints of a program p, we obtain a list of intervals that we call the *parameter rectangle*, written $\text{rect}(p)$. The parameter rectangle is said to be well-formed if for every interval (a, b) in the list, we have $a < b$. We define the *dimension* of p as $\dim(p) = |\text{rect}(p)|$. A list l of real numbers is said to be in a list L of intervals, written $l \in L$ if the two lists have the same length, and for all i, the ith element of l (a real number) is in the ith element of L (an open interval).

Next, we define a notion of iterated integration over a list of intervals. Given a function $f : (\mathit{List}\ \mathbb{R}) \to \mathbb{R}$, and a non-empty list L of open intervals, we recursively define

$$\int_L f = \begin{cases} \int_a^b f([x])dx & L = [(a, b)] \\ \int_a^b \left(\int_{L'} (\lambda l.\, f(x :: l)) \right) dx & L = (a, b) :: L' \end{cases}$$

The integral $\int_L f$ is said to exist if all of the 1-dimensional integrals involved exist.

Recall that for each parameter variable v, the program defines an output map to convert the internal representation of v used in execution into a constrained format expected by the user. For a list l of real numbers such that $|l| = \dim(p)$, we write outmap(p, l) for the list that results by applying the ith parameter's variable map to the ith element of l, coercing the result into a real number using IFR.

As discussed in §2, Stan model blocks only specify the log density of the program up to addition by a constant. Given a list of data values d and a list of intervals γ, we say that the *unnormalized* probability $U(p, d, L)$ of γ is

$$U(p, d, L) = \int_{\text{rect}(p)} \left(\lambda l.\ \exp(\text{logdensity}(p, d, l)) \cdot [\text{outmap}(p, l) \in L]\right)$$

where $[\text{outmap}(p, l) \in L]$ is an instance of Iverson bracket notation, meaning that it equals 1 if $\text{outmap}(p, l) \in L$ and 0 if $\text{outmap}(p, l) \notin L$. That is, we integrate over the whole parameter space of the program p, but only parameter values l that would fall in L when output to the user contribute to the probability.

To obtain the *normalized* probability, we must divide by the so-called normalizing constant, so that given a data assignment d, the integral over the whole space of parameters has probability 1.

The normalizing constant $Z(p, d)$ is

$$Z(p, d) = \int_{\mathsf{rect}(p)} \left(\lambda l.\ \exp(\mathsf{logdensity}(p, d, l))\right)$$

and we assume that this integral exists and is non-zero. A program's behavior is undefined if these assumptions do not hold. Under these assumptions, we can finally define the probability distribution corresponding to a program p and a list of data values d:

$$\Pr(p, d, L) = \frac{U(p, d, L)}{Z(p, d)}$$

4 SPECIFICATION AND PROOF TECHNIQUE

This section defines formally what it means for the density compiler to be correct, and then provides an overview of our proof technique for establishing correctness. The basic strategy is to decompose each proof into two proofs, one at the operational layer and one at the denotational layer. For the operational layer, we re-use CompCert's forward simulation proof style. At the denotational layer, the proofs follow from standard mathematical results about properties of integration.

4.1 Semantic Preservation

The key correctness condition for compilation is preservation of the denotational semantics of a Stanlight `model` block. As usual with such semantic preservation definitions, when the source program has undefined behavior, the output of the compiler is licensed to have arbitrary behavior. To that end, we define the notion of a safe data assignment, $\mathsf{safe\_data}(p, d)$:

$$\mathsf{safe\_data}(p, d) = \forall l. l \in \mathsf{rect}(p) \Rightarrow \mathsf{is\_safe}(p, d, l)$$

That is, a data assignment d is safe whenever execution with that data would be safe for all parameter values in the program's parameter rectangle.

Definition 4.1 (Refinement). Program p_1 refines p_2, written $p_1 \sqsubseteq p_2$ if, assuming $\mathsf{rect}(p_2)$ is well-formed, the following all hold:[5]

(1) $\dim(p_1) = \dim(p_2)$ and $\mathsf{rect}(p_1)$ is well-formed.
(2) For all d, if $\mathsf{safe\_data}(p_2, d)$ then $\mathsf{safe\_data}(p_1, d)$.
(3) For all d, if $\mathsf{safe\_data}(p_2, d)$ then $\forall L. \Pr(p_1, d, L) = \Pr(p_2, d, L)$.

At a high level, this definition requires that if p_2 is well-defined, then p_1 is well-defined and they assign the same probabilities to rectangular subsets of the parameter space. In other words, the probability distribution represented by the original program is preserved.

The correctness specification for the compiler is that compiled programs refine their source:

Definition 4.2 (Compiler Correctness). A Stanlight to Stanlight density compiler C is *correct* if for all programs p, the refinement $C(p) \sqsubseteq p$ holds.

A key property is that refinement is transitive.

Theorem 4.1 (Transitivity of Refinement). If $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_3$, then $p_1 \sqsubseteq p_3$.

This means each pass of the compiler can be verified separately:

Theorem 4.2 (Composition of Compiler Correctness). If C_1 and C_2 are correct Stanlight to Stanlight density compilers, then $C_1 \circ C_2$ is correct.

[5] Our mechanized proofs further require that p_2 and p_1 are linked with all of the standard math libraries that are used by Stanlight programs, and that the library functions have their intended semantics.

As in CompCert itself, this justifies decomposing the compiler into a series of small passes that can each be verified independently. But how do we verify each of those passes? Before explaining our technique for doing so, we recall how those proofs are carried out in CompCert.

4.2 Background: Forward Simulation in CompCert

Recall that a standard (operational) characterization of correctness for a compiler C is that, if p is a well-defined program, then (1) $C(p)$ should be well-defined, and (2) if $C(p)$ has some executable behavior B, then the source program p should have behavior B as well.

One way to establish this form of correctness is to inductively construct a *backward* simulation between execution of $C(p)$ and p.[6] In its simplest form, this technique involves defining a simulation relation R between states of $C(p)$ and p with the following properties:

(1) If σ_c and σ_s are initial states of $C(p)$ and p, respectively, then $R(\sigma_c, \sigma_s)$ holds.
(2) For all states σ_c and σ_s such that $R(\sigma_c, \sigma_s)$ holds, if σ_c steps to σ_c', then there exists σ_s' such that $\sigma_s \rightarrow^+ \sigma_s'$ and $R(\sigma_c', \sigma_s')$.
(3) If $R(\sigma_c, \sigma_s)$ holds and σ_c is a final state (i.e. it is a return value that cannot take any more steps), then σ_s is a final state with an equivalent return value.

The intuition behind the name "backward" is that in (2) above, for each step of the compiled program, we must exhibit matching step(s) by the pre-compiled program, hence backwards with respect to the compilation pipeline. However, it can be difficult to construct a backward simulation this way, since $C(p)$ is often more low-level than p. Variants of this technique allow for optionally showing stuttering steps in part (2), where the compiled program takes some bounded number of steps before having to construct a matching step of the pre-compiled program. Even then, reconstructing steps of p from steps of $C(p)$ is not always easy.

Fortunately, when working with programs expressed in a *deterministic* programming language, a backward simulation can instead be derived from a *forward* simulation. The process for constructing a forward simulation is similar to the above, but crucially, property 2 above becomes:

(2') For all states σ_c and σ_s such that $R(\sigma_c, \sigma_s)$ holds, if σ_s steps to σ_s', then there exists σ_c' such that $\sigma_c \rightarrow^+ \sigma_c'$ and $R(\sigma_c', \sigma_s')$.

That is, given a step in the source program, we exhibit matching steps in the compiled program, which is often easier to do. CompCert exploits this approach heavily, using deterministic intermediate languages to justify doing so.

4.3 Forward Simulation in ProbCompCert

Recall from §3.1 that although the MCMC loop in ProbCompCert's runtime is randomized, the execution of the `model` block is deterministic. Thus, we may also use forward simulation in ProbCompCert to establish properties about the operational behavior of the compiled model block.

However, CompCert's simulation relations show that the exact input/output behavior of the program is preserved, in the sense that the final return value of the compiled program will be the same as the source program. But some of the compilation passes performed by ProbCompCert, such as Reparameterization and Additive Constant optimization, change the density function being computed, even though they preserve the overall denotational semantics of the program.

We can accommodate this by varying the steps of constructing a forward simulation proof described above. The idea is to allow for remapping the input parameters and output target value when constructing the simulation. The general pattern is as follows:

[6]The terms "backward" and "forward" for simulations are sometimes used in a different sense by other authors. Here, we follow the conventions from CompCert.

Definition 4.3. Let $T : \mathit{List\ Val} \to \mathit{List\ Val} \to \mathbb{R} \to \mathbb{R}$ and $\phi : \mathit{List\ Val} \to \mathit{List\ Val}$ be maps such that for all d, l, and r, there exists r' such that $T(d, l, r') = r$. A (T, ϕ)-forward simulation between two Stanlight programs p_c and p_s is a relation R on Stanlight states with the following properties:

(1) If $\mathsf{initial}(p_s, d, \gamma, \sigma_s)$ and $\mathsf{initial}(p_c, d, \phi(\gamma), \sigma_c)$ then $R(\sigma_c, \sigma_s)$ holds.
(2) For all states σ_c and σ_s such that $R(\sigma_c, \sigma_s)$ holds, if σ_s steps to σ_s', then there exists σ_c' such that $\sigma_c \to^+ \sigma_c'$ and $R(\sigma_c', \sigma_s')$.
(3) If $R(\sigma_c, \sigma_s)$ holds and $\sigma_s = \texttt{Return}\ t$ then $\sigma_c = \texttt{Return}\ T(d, \gamma, t)$.

Constructing such a simulation implies a relation between the logdensity of the two programs:

Theorem 4.3. If there is a (T, ϕ)-forward simulation between p_c and p_s, then for all d and γ such that $\mathsf{is\_safe}(p_s, d, \gamma)$ we have

$$\mathsf{logdensity}(p_c, d, \phi(\gamma)) = T(d, \gamma, \mathsf{logdensity}(p_s, d, \gamma))$$

Once we have constructed a (T, ϕ)-forward simulation for appropriate choice of T and ϕ, all of the operational reasoning about the compilation pass is done. We then just need to show that the integrals defining $\Pr(p_c, d, L)$ and $\Pr(p_s, d, L)$ are equivalent, using this theorem to relate the logdensity term that appears in each integrand.

In the simplest cases, where a pass does *not* modify the logdensity function being computed, such as the initial Sampling pass in ProbCompCert, we can take ϕ to be the identity map and let $T(d, \gamma, r) = r$. With that choice, the properties for (T, ϕ)-forward simulation are equivalent to the standard form of forward simulation used in CompCert, and Theorem 4.3 specializes to saying that

$$\mathsf{logdensity}(p_c, d, \gamma) = \mathsf{logdensity}(p_s, d, \gamma)$$

Assuming the pass does not modify the parameter list, this equation implies that p_c refines p_s, as the integrals defining $\Pr(p_c, d, L)$ and $\Pr(p_s, d, L)$ are equivalent.

In other words, a "standard" forward simulation from CompCert implies denotational refinement. This means that although ProbCompCert's correctness proof does not yet formally connect to the correctness proof for CompCert, it should be possible to adapt CompCert's existing simulation proof to eventually achieve an end-to-end result for semantic preservation.

In the next two sections, we turn to discussing passes for which the more general notion of (T, ϕ)-forward simulation is required to prove semantic preservation.

5 REPARAMETERIZATION

The Reparameterization pass transforms the model block in order to remove constraints from parameter variables. This simplifies the job of the proposal generator, as it now no longer needs to worry about constructing proposals that would satisfy these constraints.

To accommodate unconstrained parameter values in the model block, at each use of a parameter that originally had a constraint c, the compiler inserts code to apply a function f_c that maps the unconstrained value into a value satisfying the constraint. The mappings used by ProbCompCert for the three constraints supported by Stanlight are:

Constraint c	**Mapping** $f_c(x)$
$\texttt{lower} = a$	$\exp(x) + a$
$\texttt{upper} = b$	$b - \exp(-x)$
$\texttt{lower} = a, \texttt{upper} = b$	$a + (b - a) \cdot \mathrm{expit}(x)$

where $\mathrm{expit}(x) = 1/(1 + \exp(-x))$. For each constraint c and $x \in \mathbb{R}$, we have that $f_c(x) \in \mathsf{interval\_of\_constraint}(c)$.

```
1  data {
2    int flips[100];
3  }
4  parameters {
5    real<lower=0.0,upper=1.0> mu;
6  }
7  model {
8    target +=
9      uniform_lpdf(mu | 0,1);
10   for (i in 1:100) {
11     target +=
12       bernoulli_lpmf(flips[i] | mu);
13   }
14 }
```

```
1  data {
2    int flips[100];
3  }
4  parameters {
5    real mu;
6  }
7  model {
8    target +=
9      uniform_lpdf((0+(1-0)*expit(mu) | 0,1);
10   for (i in 1:100) {
11     target +=
12       bernoulli_lpmf(flips[i] | (0+(1-0)*expit(mu)));
13   }
14   target +=
15     log((1 - 0) * expit(mu) * (1 - expit(mu)));
16 }
```

Fig. 7. Example of before (left) and after (right) the Reparameterization pass

However, just re-mapping the variables to a constrained form is not sufficient. Since the model block represents a density function that is integrated to obtain a probability distribution, remapping in this way amounts to a change of variables in the integral, and so we need to insert a correction factor to account for this.

To see why, recall from calculus that if $g : \mathbb{R} \to \mathbb{R}$ is a monotone and continuously differentiable function on the interval (a, b), and h is integrable on $(g(a), g(b))$, then

$$\int_{g(a)}^{g(b)} h(x)dx = \int_{a}^{b} g'(x)h(g(x))dx \tag{1}$$

(If a or b are $-\infty$ or $+\infty$, one can instead use $\lim_{x\to a} g(x)$ and $\lim_{x\to b} g(x)$ instead of $g(a)$ and $g(b)$ in the limits of integration for the integral on the left.) The crucial part is that in the integral on the right, we must adjust the integrand by scaling by $g'(x)$, the so-called Jacobian of the transform.

In our setting, g is f_c, h is the integrand defining $U(p, d, L)$ and $Z(p, d)$, and a and b are $-\infty$ and $+\infty$. Thus we must adjust the `target` calculation by the Jacobian as well. Since the model block is returning the logarithm of the density, we add $\log(f_c'(x))$ instead of multiplying. The pass will insert a statement for each parameter to add this value to `target` at the end of the `model` block.

Figure 7 shows the result of applying the reparameterization transform to the example from Figure 1b from §1. The left side of the figure reproduces the code of that example and the right hand side shows the output of the transform. On line 5 on the right, we see that the constraint on the parameter `mu` is removed. The uses of `mu` on lines 9 and 12 are remapped according to f_c for $c = (\texttt{lower} = 0.0, \texttt{upper} = 1.0)$. Finally, lines 14-15 add the Jacobian $f_c'(\texttt{mu})$ to the `target` variable.

This example involves just one constrained parameter. If there are multiple constrained parameters, each is re-mapped according to the corresponding f_c. Since each re-mapping is independent of the others, the overall Jacobian is just the sum of the Jacobians for each variable.

To prove the correctness of this pass, we first construct a (T, ϕ)-forward simulation. The mapping T adds the Jacobian for each parameter value, while ϕ applies the *inverse* of f_c for each variable – i.e. it unconstrains the parameter values from the source side of the simulation. The simulation relation R is straightforward: the key invariant is that if pm_s and pm_c are the parameter maps in the source and compiled program states, respectively, then for each variable i with constraint c,

```
1  model {
2    real mu;
3    target += normal_lpdf(alpha | 0,1);
4    target += normal_lpdf(beta | 0,1);
5    for (i in 1:10) {
6      mu = alpha + beta * x[i];
7      target += normal_lpdf(y[i] | mu, 1);
8      target += 3.0;
9    }
10 }
```

```
1  model {
2    real mu;
3    target += normal_lupdf(alpha | 0,1);
4    target += normal_lupdf(beta | 0,1);
5    for (i in 1:10) {
6      mu = alpha + beta * x[i];
7      target += normal_lupdf(y[i] | mu, 1);
8      target += 0.0;
9    }
10 }
```

Fig. 8. Example of before (left) and after (right) the additive constant optimization pass.

we have $pm_s(i) = f_c(pm_c(i))$. Otherwise, the states are essentially the same until the very end of executing the model block, in which we add the Jacobian on the output side.

Applying Theorem 4.3 to this simulation, we are left to prove that the resulting integrals over parameter space are equivalent. This follows by applying the change-of-variables theorem inductively for each parameter variable.

6 ADDITIVE CONSTANT OPTIMIZATION

The Additive Constant optimization pass simplifies the density calculation by removing addition of constants to the `target` variable. For example, under certain conditions it can remove statements of the form `target += c` for a constant c. Similarly, for statements of the form `target += f(...)`, where `f` is a standard library function, it may replace `f` with a more optimized version of the function that omits such constants.

Figure 8 gives an example of the results of applying this pass to a simple example. The original model block, on the left, implements a simple linear regression model. The details of the original model are not important, but we see that it has four `target` statements, three of which increment by the log density of the normal distribution, `normal_lpdf`, and one which adds the constant `3.0`. On the right, the `normal_lpdf` calls have been replaced by `normal_lupdf` and the constant `3.0` has been replaced by `0.0`.[7] The function `normal_lupdf` is a library function that computes the logarithm of the PDF of the normal distribution but with certain normalizing constants dropped. In particular, `normal_lpdf` and `normal_lupdf` compute the following mathematical functions:

$$\texttt{normal\_lpdf}(y|\mu,\sigma) = \log\left(\frac{1}{\sqrt{2\pi}}\cdot\frac{1}{\sigma}\exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)\right)$$

$$\texttt{normal\_lupdf}(y|\mu,\sigma) = \log\left(\frac{1}{\sigma}\exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)\right)$$

so that their difference is $\log(1/\sqrt{2\pi})$.

The optimizations on lines 7 and 8 in this example occur within the body of a loop that runs for 10 iterations. Thus, if the changes on lines 7 and 8 drops constants $c_1 = \log(1/\sqrt{2\pi})$ and $c_2 = 3.0$, respectively, the net effect is to decrease the overall density by $10(c_1 + c_2)$. Crucially, the number of iterations of the loop here does not have any dependency on the parameters of the model. If such a dependency existed, then dropping constants from the body of the loop might nevertheless shift the overall density by a non-constant amount.

[7] Subsequent standard optimization passes in CompCert can then entirely remove this addition of 0.0.

To avoid this kind of control-flow dependency and related problems, the pass implements several checks and restrictions:

(1) The `target` variable must only be used as part of `target += ` statements, otherwise the optimization performs no changes.
(2) Bodies of loops are altered only when the loop bounds are constant and the body does not modify the loop iterator.
(3) Branches of conditional statements are not optimized.

Some of these restrictions could be loosened by doing a more careful dataflow analysis. For example, it should be possible to allow loop bounds to be non-constant so long as they only have dependency on `data` variables.

Why is this optimization pass correct? Recall in §2 that we gave intuition for why by observing that the MCMC loop only cares about the *difference* between the log density of the chain's current state and a candidate's log density. Thus, shifting the log density by addition of a constant would not change this relative difference, and so would preserve the behavior of the program.

We can now also see that such a transformation is semantics preserving from the perspective of the denotational semantics introduced in §3.2. The integral defining a program's probability distribution is scaled by a *normalizing* constant $Z(p, d)$. Thus if we have two programs p_c and p_s with the property that $\mathsf{logdensity}(p_c, d, \gamma) + r = \mathsf{logdensity}(p_s, d, \gamma)$ for all γ, then, assuming the parameters of p_c and p_s are the same, we have:

$$\begin{aligned}
U(p_s, d, L) &= \int_{\mathsf{rect}(p_s)} \big(\lambda l.\ \exp(\mathsf{logdensity}(p_s, d, l)) \cdot [\mathsf{outmap}(p_s, l) \in L]\big) \\
&= \int_{\mathsf{rect}(p_c)} \big(\lambda l.\ \exp(\mathsf{logdensity}(p_c, d, l) + r) \cdot [\mathsf{outmap}(p_c, l) \in L]\big) \\
&= \exp(r) \cdot \int_{\mathsf{rect}(p_c)} \big(\lambda l.\ \exp(\mathsf{logdensity}(p_c, d, l)) \cdot [\mathsf{outmap}(p_c, l) \in L]\big) \\
&= \exp(r) \cdot U(p_c, d, L)
\end{aligned}$$

Similarly, $Z(p_s, d) = \exp(r) \cdot Z(p_c, d)$, so that

$$\Pr(p_s, d, L) = \frac{U(p_s, d, L)}{Z(p_s, d)} = \frac{\exp(r) \cdot U(p_c, d, L)}{\exp(r) \cdot Z(p_c, d)} = \Pr(p_c, d, L)$$

That outlines the denotational proof. For the operational part, we first inductively define a function drop such that for a statement s, $\mathsf{drop}(s)$ is equal to the total constant that would be dropped when running the optimization on s. Then, we construct a (T, ϕ)-simulation with $T(x) = x - r$ and ϕ as the identity. To define the simulation relation R, we define a relation $\mathsf{matchStmt}(s, s', z)$ which holds when (among other things) executing s adds z more to the `target` variable than executing s' does. Similarly, $\mathsf{matchCont}(k, k', z)$ says that executing the remaining statements in the continuation stack k will add z more to `target` than executing the continuation stack k'.

A key part of the invariant captured by the simulation relation R is that if the source state is $\mathsf{Running}\ f\ s\ t\ k\ env\ m\ pm$ and the compiled state is $\mathsf{Running}\ f'\ s'\ t'\ k'\ env'\ m'\ pm'$ then $\mathsf{matchStmt}(s, s', z_1)$ and $\mathsf{matchCont}(k, k', z_2)$ hold for some z_1 and z_2 such that $t - t' = r - z_1 - z_2$. When the program finishes running, before transitioning to Return states, this relation will hold with $z_1 = 0$ and $z_2 = 0$, so that we get $t - t' = r$, as desired.

Comparison with the Stan compiler. The current compiler for Stan does not implement an additive constant optimization pass. Instead, when translating a sample statement `x ~ normal(mu, sigma)`, rather than replacing it with `target += normal_lpdf(x | mu, sigma)` as ProbCompCert does,

Stan translates it to `target += normal_lupdf(x | mu, sigma)`, with the `lupdf` indicating the version of the normal density function that drops the additive constant $\log(1/\sqrt{2\pi})$. Other distributions are similarly compiled to use analogous `lupdf` versions of their density functions. Thus, many of the changes that ProbCompCert's additive constant optimization would perform happen automatically in Stan as part of compiling away the ~ statement.

A key difference is that Stan's compiler will *always* use the `lupdf` version when compiling a ~ statement, whereas ProbCompCert will only replace an `lpdf` with `lupdf` in a `target` increment statement when conditions (1) through (3) mentioned above hold. Currently, the Stan reference manual (version 2.31) claims that although ~ is always compiled using `lupdf` versions of density functions, this will "lead to the same sampling behavior" as using `lpdf` [Stan Development Team 2023, Section 7.4]. However, this is not necessarily true when conditions (1) through (3) are violated.[8] In personal communication with the Stan developers, we have shown them counter-examples to this claim, which they have confirmed, and they have mentioned possible changes to correct the phrasing in the manual.[9]

7 EVALUATION

Proof development size and relative effort. Figure 9 gives a listing of the lines of proof and code for the various components of ProbCompCert. For each pass, we give a breakdown of the lines of proof required for operational reasoning versus denotational reasoning. The Sampling pass's denotational proof is generic for any pass that does not change the logdensity being calculated and is reusable, whereas the other two are specific to the pass.

The operational proofs are typically larger. That is because operational proofs are done by induction over possible steps in the simulation, so their size is roughly proportional to the number of cases in the operational semantics and the extent of changes in the code. Meanwhile, the hard work of the denotational proofs are in the underlying mechanized proofs of integration lemmas, which we either re-use from Coquelicot [Boldo et al. 2015] or proved in a math library.

	Proof/Spec.	**Code**
Sampling	320 (O) + 240 (D)	30
Reparam.	3,160 (O) + 270 (D)	300
Additive Constant	1,500 (O) + 270 (D)	200
Semantics	1,060	–
Math Library	3,000	–

Fig. 9. Lines of proof, specification, and code for verified components. The label "O" is for operational proofs, "D" is for denotational proofs.

Performance evaluation. We evaluate the performance of ProbCompCert's generated density code as compared to that produced by the official Stan compiler (v.2.30.1). In general, comparing the performance of PPLs is subtle, because one must consider both the speed with which samples are generated as well as the rate at which the chain converges to the posterior distribution. For example, a system may generate samples more slowly than another, yet because it uses an algorithm with faster convergence, fewer samples are required overall to produce good statistical estimates. Since ProbCompCert's unverified runtime and proposal generator are quite different from Stan's, this sort of comparison would be difficult to interpret.

To avoid this, we use BridgeStan [Roualdes et al. 2023], a library that exposes a C API for Stan's compiled output. Using BridgeStan, we are able to link Stan's generated density code into ProbCompCert's runtime. We run both ProbCompCert's normal compiled output and the BridgeStan-linked version on a collection of benchmarks and measure the time taken to generate a

[8] Stan does have a warning mode that checks for branching on parameter values, since this can introduce discontinuities in the density function that lead to poor MCMC performance for other reasons. This would flag many such examples.
[9] https://github.com/stan-dev/docs/issues/588.

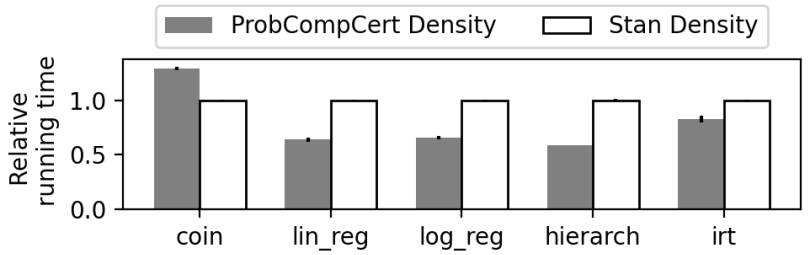

Fig. 10. Average relative running time of standard ProbCompCert and ProbCompCert linked with Stan's density code. Lower is better. Results are normalized so that the Stan-linked version has value 1. Error bars are shown but small, as standard deviation was below 5 percent of the average for each configuration.

fixed number of samples. With this setup, the difference between the two systems is in the density model code, so this enables a more direct assessment of the performance of the density code.

We use five benchmarks: the coin flipping example from the introduction, a linear regression similar to Figure 8, a logistic regression, a hierarchical linear regression, and a simple item response theory (IRT) model. For each experiment, we time how long it takes to run 1 million iterations of the MCMC loop, saving every 10,000th state as a sample. Each experiment is run five times. The models have between 2-30 parameters and 100-400 observed data variables. The experiments were run on a laptop with an Intel i7-6600U CPU @ 2.60GHz processor, running Ubuntu 18.04.

Figure 10 shows the results of these experiments, normalized so that the Stan density version has average value 1 for each benchmark. On 4 of the 5 benchmarks, the ProbCompCert average is less than 85% of the running time of the Stan-linked version. These results should not be interpreted as a definitive claim about the performance of Stan. For example, Stan's generated code and libraries implement various checks for opportunities to apply vectorized operations. In these benchmarks, these just add overhead, but in more sophisticated models they offer opportunities for vast speed-ups. Rather, the conclusion we draw from these results is that for the subset of Stan that ProbCompCert supports, its density code's performance is comparable to that of Stan.

In contrast to the other benchmarks, on the coin flipping benchmark, ProbCompCert has worse performance. The reason is that, as we saw in Figure 7, after Reparameterization, ProbCompCert produces code that re-constrains the `mu` parameter of this model at every use of `mu`, including in the main for loop. In contrast, Stan computes the constrained version once and stores it in a separate variable which is re-used. The repeated re-computation of the constrained value in the loop ends up being a significant fraction of the total work done for this simple example.

We confirmed this by changing the model to add a local variable `mu2`, setting `mu2 = mu`, and replacing uses of `mu` with `mu2`. Then, after Reparameterization, the inserted re-constraining happens only when setting `mu2 = mu`. This change made ProbCompCert's performance about the same as the Stan-linked form. CompCert's common sub-expression elimination cannot make this change automatically, as it does not know that the `expit` function calls computing the constrained value are pure. This could be addressed by adding `expit`, and the other math routines ProbCompCert uses, as intrinsics to CompCert. It should also be possible to add a pass before Reparameterization to introduce variables like we did with `mu2`, without changing Reparameterization's proof.

8 RELATED WORK

Compilation of probabilistic programming languages. Stan's latest OCaml compiler is organized in sequences of transformations between 3 intermediate representations. ProbCompCert uses many

more intermediate passes to ease the task of verification. Besides Stan, several other probabilsitic programming languages come with compilers, as opposed to interpreters, including the one for Infer.net [Kazemi and Poole 2016], Swift [Wu et al. 2016], Gen [Cusumano-Towner et al. 2019], or the Reactive PPL [Baudart et al. 2020]. Some compilers target GPUs [Tristan et al. 2014; Huang et al. 2017] and FPGAs [Banerjee et al. 2019] in order to exploit parallelism in inference.

Bhat et al. [2013] develop a density compiler for the Fun PPL [Borgström et al. 2011] and prove its correctness on paper. Eberl et al. [2015] later produced a machine-checked proof of this result in Isabelle. These works use the term density compiler in a different sense than we have here. There, the compiler takes as input a program describing a generative process and emits a function in a high-level language to calculate the density of that model using a numerical integration primitive. In contrast, in Stan, the model block input to the compiler is already a direct description of how to calculate the density, and the compiler generates low-level code for computing that density.

Semantics of probabilistic programs. Early work on program semantics involving probabilities was focused on modeling uses of randomness for applications like randomized data structures, as opposed to probabilistic programming languages for statistical modeling and inference. Kozen [1981] gave a very early semantics of programs with randomized behavior. A number of domain-theoretic denotational models [Saheb-Djahromi 1980; Jones and Plotkin 1989] were developed for higher-order languages with recursion and probabilistic choice.

More recent work has developed semantics for languages for statistical inference. Ramsey and Pfeffer [2002] present a monadic semantics, using the Giry monad [Giry 1982], to model a stochastic lambda calculus. Staton [2017] gives a denotational semantics for a first-order probabilistic programming language and uses it to prove the correctness of swapping the order of statements in a model. The difficulty of justifying such a seemingly simple transformation hints at the complexity of probabilistic programming languages and their semantics. Culpepper and Cobb [2017] proved a similar result using operational semantics and logical relations, subsequently extended by Wand et al. [2018] to support recursion. Borgström et al. [2016] prove an equivalence between two forms of semantics for probabilistic programming languages, one which ascribes weights to traces of executions, and a more extensional version that describes programs as distributions over return values. Multiple approaches to denotational models for higher-order PPLs have been developed in the past several years [Heunen et al. 2017; Ehrhard et al. 2017; Huang and Morrisett 2016]. Scibior et al. [2018] use such a semantics to prove on paper the correctness of inference procedures for a higher-order language. Lew et al. [2020] extend this approach and develop a type system for safe and provably correct composition of inference procedures.

Gorinova et al. [2019] give a semantics to Stan, and present a Stan-like language called SlicStan that removes some restrictions of Stan, such as the need to segment programs into blocks. They translate SlicStan into Stan, and prove the correctness of this translation. As mentioned, their semantics highly influenced design choices for the semantics in ProbCompCert. ProbCompCert adds semantics of constraints, which is needed to prove the correctness of Reparameterization.

Bugs in PPLs. Dutta et al. [2018] conduct a bug study of PPLs, including Stan, and develop a PPL fuzzer. Their results allow for some assessment of the decision in §3 to use idealized real semantics for the denotational layer. Many bugs they mention appear to be semantics violating even with exact real arithmetic, and hence would be ruled out if the entire system were verified with this semantics. However, they also describe serious bugs related to numerical issues that would not be.

9 CONCLUSION

This paper described ProbCompCert and the verified passes it uses for density compilation. ProbCompCert's density compilation is decomposed into several discrete passes to enable modular

verification. Its layered semantics makes it possible to verify passes by splitting their proofs into operational and denotational parts.

It would be interesting future work to verify other parts of the compiler and address the limitations of ProbCompCert described earlier in §1. The remaining unverified passes of the density compiler can be handled with the techniques described here, particularly because they do not modify the logdensity of the model further. Certain other features, such as supporting nested arrays, could also be handled by the methodology presented here. However, verifying features like Stan's multivariate constraints would require substantial proofs in mechanized multivariate analysis or linear algebra.

A more conceptual challenge lies in verifying an improved proposal generator, such as the one used in Stan itself, or the MCMC algorithm runtime. In both cases, the semantics used so far for ProbCompCert's density compiler should be complementary in specifying the behavior and interactions of these other components.

Finally, it would be interesting to explore verified compilation of probabilistic language features not found in Stan, such as higher-order functions or mixtures of continuous and discrete parameters. Modeling these language features would require a more sophisticated denotational semantics, such as ones used in the work cited in §8. Nevertheless, it may still be possible to use the two-layer approach described here to verify certain parts of compilation even in such languages.

ACKNOWLEDGMENTS

We thank the paper's shepherd, Yizhou Zhang, as well as the anonymous reviewers. The authors are also grateful to the anonymous PLDI artifact evaluators for their careful assessment and feedback on the artifact accompanying this paper. We would further like to thank Brian Ward, Bob Carpenter, Sam Stites, and Xavier Leroy for their help. This material is based upon work supported by the National Science Foundation under Grant No. 2106559.

ARTIFACT AVAILABILITY

An artifact accompanying this paper is available [Tassarotti and Tristan 2023], which includes the source code for ProbCompCert, as well as scripts for reproducing the experiments shown in Figure 10. The source code for ProbCompCert is also available in a public repository [ProbCompCert Development Team 2023], where active development continues.

REFERENCES

Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2019. AcMC 2 : Accelerating Markov Chain Monte Carlo Algorithms for Probabilistic Models. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 515–528. https://doi.org/10.1145/3297858.3304019

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 898–912. https://doi.org/10.1145/3385412.3386009

Matthew R. Becker. 2016. NUTS Sampler Broken (stan-dev/stan issue #2178). https://github.com/stan-dev/stan/issues/2178.

Michael Betancourt. 2018. A Conceptual Introduction to Hamiltonian Monte Carlo. arXiv:1701.02434 [stat.ME]

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 508–522.

Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Math. Comput. Sci.* 9, 1 (2015), 41–62.

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure Transformer Semantics for Bayesian Machine Learning. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 77–96.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 33–46.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 368–392.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 221–236.

Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 574–586.

Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. 2015. A Verified Compiler for Probability Density Functions. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–104.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2017. Measurable Cones and Stable, Measurable Functions: A Model for Probabilistic Higher-Order Programming. *Proc. ACM Program. Lang.* 2, POPL, Article 59 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158147

Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis (Lecture Notes in Mathematics, Vol. 915)*, B. Banaschewski (Ed.). 68–85.

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2019. Probabilistic Programming with Densities in SlicStan: Efficient, Flexible, and Deterministic. *Proc. ACM Program. Lang.* 3, POPL, Article 35 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290348

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science* (Reykjavík, Iceland) *(LICS '17)*. IEEE Press, Article 77, 12 pages.

Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 47 (2014), 1593–1623. http://jmlr.org/papers/v15/hoffman14a.html

Daniel Huang and Greg Morrisett. 2016. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 337–363.

Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. Association for Computing Machinery.

C. Jones and G. Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (Pacific Grove, California, USA). IEEE Press, 186–195.

S. Kazemi and D. Poole. 2016. Knowledge Compilation for Lifted Probabilistic Inference: Compiling to a Low-Level Language. In *KR*.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michale Carbin, and Vikash K. Mansinghka. 2020. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 19 (Jan. 2020), 32 pages.

ProbCompCert Development Team. 2023. *ProbCompCert Git Repository*. https://github.com/jtassarotti/probcompcert

Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL*. 154–165.

Gareth O Roberts, Jeffrey S Rosenthal, and Peter O Schwartz. 1998. Convergence properties of perturbed Markov chains. *Journal of applied probability* 35, 1 (1998), 1–11.

Edward Roualdes, Brian Ward, Seth Axen, and Bob Carpenter. 2023. *BridgeStan: Efficient in-memory access to Stan programs through Python, Julia, and R.* https://doi.org/10.5281/zenodo.7760173

N. Saheb-Djahromi. 1980. Cpo's of measures for nondeterminism. *Theoretical Computer Science* 12, 1 (1980), 19 – 37. https://doi.org/10.1016/0304-3975(80)90003-1

A. Scibior, Ohad Kammar, Matthijs Vákár, S. Staton, H. Yang, Yufei Cai, K. Ostermann, Sean K. Moss, C. Heunen, and Z. Ghahramani. 2018. Denotational validation of higher-order Bayesian inference. *Proceedings of the ACM on Programming Languages* 2 (2018), 1 – 29.

Stan Development Team. 2023. Stan Language Reference Manual (v. 2.31). https://mc-stan.org/docs/2_31/reference-manual/

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag, Berlin, Heidelberg, 855–879. https://doi.org/10.1007/978-3-662-54434-1_32

Joseph Tassarotti and Jean-Baptiste Tristan. 2023. *Verified Density Compilation for a Probabilistic Programming Language (Artifact).* https://doi.org/10.5281/zenodo.7709874

Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. 2014. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Neural Information Processing Systems Foundation.

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. *Proc. ACM Program. Lang.* 2, ICFP, Article 87 (July 2018), 30 pages. https://doi.org/10.1145/3236782

Yi Wu, Lei Li, S. Russell, and Rastislav Bodík. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *IJCAI*.

Received 2022-11-10; accepted 2023-03-31