STAPPL: Statically Typed Probabilistic Programming Language

Jaeho Lee [* 1] **Hyeonseo Yang** [* 1] **Dohyung Kim** [* 1] **Doyoon Lee** [* 1]

Abstract

We have designed and implemented STAPPL, a statically typed probabilistic programming language that compiles to probabilistic graphical models. The static type system of STAPPL can guarantee the safety of programs at compile time, eliminating unsoundness in probabilistic programs. STAPPL compiles a program into a graphical model on which inference can be run.

1. Introduction

Probabilistic programming languages (PPLs) enable one to define rich and sophisticated probabilistic models using high-level programming abstractions (van de Meent et al., 2018). To describe a probabilistic model, PPLs provide special constructs for defining random variables, sampling from distributions, and conditioning on observed data, along with conventional programming constructs. Because probabilistic programs are built from familiar programming building blocks, such as functions, variables, and conditionals, it is advantageous to understand and extend probabilistic models containing complex dependencies between random variables.

First-order probabilistic programming languages can represent any probabilistic model that can be represented as a graphical model (van de Meent et al., 2018). Therefore, it is possible to compile a deterministic probabilistic program without higher-order functions into a graphical model, which can be used for inference.

Unfortunately, PPLs often lack the safety guarantees, as they are typically embedded into existing dynamic languages like Python (Bingham et al., 2019) and Clojure (Wood et al., 2014). While dynamic typing provides flexibility and ease of use for rapid prototyping, it can lead to runtime errors and performance bottlenecks in complex probabilistic models. Languages with dynamic typing typically makes it very difficult to reason about the correctness of the program and cause unexpected bugs at runtime.

Statically typed languages such as OCaml and Haskell, on the other hand, offer safe and expressive type systems that allow programmers to represent complex properties through types, effectively preventing runtime type errors (Pierce, 2002). When a programming language is carefully designed along with a static type system, a guarantee of no runtime errors can be achieved for well-typed programs. Moreover, static type systems can provide a basis for efficient compilation and optimization strategies.

While static type systems may burden the programmer when explicit type annotations are required for every variable and function, they can be alleviated by type reconstruction[1] algorithms that automatically deduce the types of expressions in a program. These type systems build upon the principles of Hindley-Milner type system (Damas & Milner, 1982), which enables the compiler to automatically deduce the types of expressions in a program. Popular functional programming languages like OCaml and Haskell are equipped with Hindley-Milner type system, which allows the programmer to write programs without explicit type annotations while still enjoying the benefits of static typing. This shows that properly designed static type systems can be both expressive and ergonomic.

We propose a statically typed probabilistic programming language STAPPL that aims to provide a concise and expressive programming constructs for defining probabilistic models and performing inference, as well as a type checking and reconstruction algorithm that helps verifying the model at compile-time. This will enable users to write complex probabilistic models with confidence, knowing that the compiler will catch many common errors before execution.

2. The problem

In this section, we will discuss the common pitfalls of probabilistic programming languages with dynamic typing, from simple type errors to more complex issues, and how STAPPL eliminates these issues at compile time.

[*]Equal contribution [1]Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea. Correspondence to: Jaeho Lee <jhlee@ropas.snu.ac.kr>.

M1522.001300 Probabilistic Graphical Models, Final project. Copyright 2024 by the author(s).

[1]Type reconstruction is more often referred as *type inference* in the programming language community. We however avoid using the term to avoid the confusion with *probabilistic inference.*

2.1. Simple type errors

Consider the following STAPPL program that represents a two-component Bernoulli mixture model.

```
let z = sample(bernoulli(0.5)) in
let p = if !z then 0.1 else 0.9 in
let d = bernoulli(p) in
observe(d, 1.5);
z
```

This program contains a type error in the `observe` statement, which the STAPPL compiler correctly detects and reports it to the user before running the program. Here, `d` is a Bernoulli distribution object of type `bool dist` which expects a Boolean observed value, yet the observed value `1.5` has type `real`. One possible fix for the above program is to change the observed value to a Boolean value `true`.

Note that we have not annotated any type information in the program, yet the STAPPL compiler is able the reconstruct the types of expressions and detect type errors at compile time. This contrasts with other statically typed languages, where type reconstruction is often an afterthought—as exemplified by Java and C/C++—or where the entire static type system was retrofitted, as in TypeScript and Python's type-hinting.

While it not be a huge issue for small programs like above to crash at runtime—which is the behavior of the majority of programming languages with dynamic type-checking—, it is an undesirable behavior for probabilistic models that may take a long time to execute. Such type errors should ideally be handled at compile-time.

2.2. Subtle type errors

Now we modify the above program slightly to introduce a more subtle type error.

```
let z = sample(bernoulli(0.5)) in
let p = if !z then 0.1 else 0.9 in
let d = bernoulli(p) in
observe(d, z);
z
```

The observed value is the random variable `z` itself, which is of type `bool`. This may not be even be a type error in rudimentary type systems, but it is clearly a mistake in the context of probabilistic programming. Observed values should be closed—`z` is not closed as it is a random variable.

STAPPL's type system is designed to catch such subtle type errors that may not be immediately obvious to the programmer. This is achieved by introducing a *stamp* system that distinguishes between random variables and values, which is attached to appropriate types.

In the above program, the type of `z` is deduced as `bool rv` (random variable), while `observe` expects it to be of type `bool val` (value). Again, the STAPPL compiler will catch this at compile time and report it to the user.

Furthermore, STAPPL is able to catch an error in the following program as well.

```
let z = sample(bernoulli(0.5)) in
let p = if !z then 0.1 else 0.9 in
let d = bernoulli(p) in
let v = if !z then true else false in
observe(d, v);
z
```

STAPPL properly figures out that the observed value `v` has type `bool rv`, as the `rv` stamp is propagated through the conditional expression from `z`.

These examples demonstrate the power of the STAPPL's type system in preventing subtle errors before the program is executed.

3. Syntax and semantics

3.1. Syntax

We now present the syntax of STAPPL.

$$
\begin{array}{lcl}
P & ::= & e \mid \texttt{fun}\ f(\overline{x})\ \{e\}\ P \\
e & ::= & c \mid x \mid e \oplus e \mid \ominus e \mid e\texttt{;}\ e \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \\
 & \mid & f(\overline{e}) \mid p(\overline{e}) \mid \texttt{sample}(e) \mid \texttt{observe}(e\texttt{,}\ e) \\
c & ::= & \texttt{()} \mid b \mid i \mid r \\
b & ::= & \texttt{true} \mid \texttt{false} \\
i & \in & \mathbb{N} \\
r & \in & \mathbb{R}
\end{array}
$$

STAPPL consists of top-level first-order functions f's—which are defined by a function name, a list of arguments, and a body expression—and a single top-level expression. The expression is largely standard, with constants c's, variables x's, binary operators $\oplus$, unary operator $\ominus$, sequencing operator `;`, conditional expression `if` e `then` e `else` e, and let-binding `let` x `=` e `in` e. It also includes function calls f's, primitive calls p's, sampling `sample(`e`)`, and observing `observe(`e_1`,` e_2`)`. Primitive calls are used to call built-in distributions to create distribution objects. Constants include unit `()`, Boolean `true` and `false`, integer i's, and real r's.

3.2. Compilation

STAPPL programs are compiled into a probabilistic graphical model using the rules in Figure 1. These rules are largely standard as described in the literature (van de Meent et al., 2018).

The graph is represented as a tuple $(V, A, \mathscr{P}, \mathscr{Y})$, where V is the set of random variables, A is the set of dependencies between random variables, $\mathscr{P}$ maps random variables to probability distribution and mass functions, and $\mathscr{Y}$ maps observed variables to their values.

The compilation rule is defined as a relation $\rho, \phi \vdash e \Downarrow (V, A, \mathscr{P}, \mathscr{Y}), E$, where ρ contains function definitions and ϕ records the flow control context that contains the predicate that represents the path we followed from the conditionals. E is a deterministic expression that does not contain any sampling or observing statements. We can read $\rho, \phi \vdash e \Downarrow (V, A, \mathscr{P}, \mathscr{Y}), E$ as "under the function definitions ρ and the flow control context ϕ, the expression e evaluates to the graph $(V, A, \mathscr{P}, \mathscr{Y})$ and the deterministic expression E".

3.3. Statics

Our STAPPL is empowered with a sound type checker that rejects wrong-typed programs. The objective of typing is to check if the program fits own position of values, random variables, and distributions. Note that our type system strictly divides integer and float expressions as in OCaml.

$$
\begin{array}{lcl}
\tau^\bullet & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{real} \mid \alpha^\bullet \\
\tau & ::= & \tau^{\bullet\dagger} \mid \tau^\bullet\ \mathsf{dist} \mid \alpha \\
\tau^{\rightarrowtail} & ::= & (\overline{\tau}) \to \tau \\
\dagger & ::= & \mathsf{val} \mid \mathsf{rv} \mid \dagger \circ \dagger \mid \beta \\
\sigma & ::= & \forall \overline{\alpha}. \forall \overline{\beta}. \tau^{\rightarrowtail} \\
\Gamma & ::= & \{\overline{x : \tau}\} \\
\Sigma & ::= & \{\overline{f : \sigma}\}
\end{array}
$$

What differentiates our type system from typical other type systems is that it distinguishes random variables to values with a special *stampping* mechanism. The stamp $\dagger$ is used to distinguish between random variables with `rv` and values with `val`, and β is a stamp variable. These stamps can be combined with the operator $\circ$ to represent the composition of stamps, where $\dagger \circ \dagger$ is used to represent the composition of stamps. $\dagger \circ \dagger' =$ `val` only if both $\dagger$'s are `val`, and $\dagger \circ \dagger' =$ `rv` otherwise. Data types are represented with τ, where $\tau^\bullet$ is a base type and α is a type variable. Function types are represented with $\tau^{\rightarrowtail}$. The monomorphic type environment Γ is a mapping from variables to types, and the polymorphic type environment Σ is a mapping from functions to types.

Our typing rules are given in Figure 2. Each type formula is written in form $\Sigma, \Gamma \vdash e : \tau$ which means that expression e matches with type τ under a polymorphic type environment Σ and a monomorphic type environment Γ.

The corresponding type reconstruction algorithm is given in Figure 3. This is based on the Hindley-Milner type reconstruction algorithm. We have implemented the algorithm in OCaml in a type-safe manner using *Generalized Algebraic Data Types*, or *GADTs*.

4. Sampling and inference

In this section, we discuss how inference of the query expression be done in STAPPL. First, we formulate the inference problem as the follows:

$$
\begin{array}{c}
\text{Evaluate deterministic expression } E \\
\text{in the context of } G(V, A, \mathscr{P}, \mathscr{Y})
\end{array}
$$

To solve the problem, we utilize Metropolis-Hastings algorithm with Gibbs sampling to infer the the distribution of query expression. In detail, we use the following algorithm to infer the distribution of the query expression:

Algorithm 1 Gibbs Sampling Algorithm

```
ctx ← empty dictionary
query ← deterministic query expression
observed_variables ← corresponding values
unobserved_variables ← 0
repeat
  for all unobserved variable $X$ do
    $x$ ← current value of $X$
    expr ← pmdf expression of $X$
    $\ell_p, x'$ ← eval_pmdf(ctx, expr)
    ctx' ← ctx $\oplus x'$
    $\ell'_p$, _ ← eval_pmdf(ctx', expr)
    $\ell_\alpha \leftarrow \ell'_p - \ell_p$
    for all variables affected by the variable $Y$ do
      $y$ ← current value of $Y$
      prob_w_cand, _ ← eval_pmdf(ctx, $y$)
      ctx ← ctx $\oplus x$
      prob_wo_cand, _ ← eval_pmdf(ctx, $y$)
      ctx ← ctx $\oplus x'$
      $\ell_\alpha \leftarrow \ell_\alpha$+ prob_w_cand - prob_wo_cand
    end for
    $\alpha \leftarrow \exp(\ell_\alpha)$ {Calculate acceptance probability}
    if Uniform[0,1] > $\alpha$ then
      ctx ← ctx ⊕ current value {Accept the candidate}
    else
      Reject the candidate
    end if
  end for
  if Sufficient bootstrapping then
    Evaluate query in the context of ctx
  end if
until Sufficient number of samples collected
Save the resulting histogram as an image
```

The algorithm consists of the following steps:

1. **Initialization** We initialize the context with an empty

Compilation of expressions. $\boxed{\rho, \phi \vdash e \Downarrow (V, A, \mathscr{P}, \mathscr{Y}), E}$

$$\textsc{Const}\ \frac{}{\rho, \phi \vdash c \Downarrow G_\varnothing, c} \qquad \textsc{Var}\ \frac{}{\rho, \phi \vdash x \Downarrow G_\varnothing, x} \qquad \textsc{Let}\ \frac{\rho, \phi \vdash e_1 \Downarrow G_1, E_1 \quad \rho, \phi \vdash e_2[E_1/v] \Downarrow G_2, E_2}{\rho, \phi \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow G_1 \cup G_2, E_2}$$

$$\textsc{If}\ \frac{\rho, \phi \vdash e_1 \Downarrow G_1, E_1 \quad \rho, \phi \wedge E_1 \vdash e_2 \Downarrow G_2, E_2 \quad \rho, \phi \wedge \neg E_1 \vdash e_3 \Downarrow G_3, E_3}{\rho, \phi \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow G_1 \cup G_2 \cup G_3, \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3}$$

$$\textsc{Sample}\ \frac{\rho, \phi \vdash e \Downarrow (V, A, \mathscr{P}, \mathscr{Y}), E \quad V \vdash \mathsf{fresh}\ x \quad Z = \mathsf{fv}(E) \quad F = \mathsf{score}(E, x)}{\rho, \phi \vdash \texttt{sample}\ e \Downarrow (V \cup \{x\}, A \cup \{(z, x) \mid z \in Z\}, \mathscr{P}[x \mapsto F], \mathscr{Y}), E}$$

$$\textsc{Observe}\ \frac{\begin{array}{c}\rho, \phi \vdash e_1 \Downarrow G_1, E_1 \quad \rho, \phi \vdash e_2 \Downarrow G_2, E_2 \quad G_1 \cup G_2 = (V, A, \mathscr{P}, \mathscr{Y}) \\ V \vdash \mathsf{fresh}\ x \quad F_1 = \mathsf{score}(E_1, x) \quad F = (\!|\ \phi\ ?\ F_1 : \mathbf{1}\ |\!) \\ Z = \mathsf{fv}(F) - \{x\} \quad \mathsf{fv}(E_2) = \varnothing \quad B = \{(z, x) \mid z \in Z\}\end{array}}{\rho, \phi \vdash \texttt{observe}\ (e_1\texttt{,}\ e_2) \Downarrow (V \cup \{x\}, A \cup B, \mathscr{P}[x \mapsto F], \mathscr{Y}[x \mapsto E_2]), E_2}$$

$$\textsc{PrimCall}\ \frac{(\rho, \phi \vdash e_i \Downarrow G_i, E_i)_{i=1}^n}{\rho, \phi \vdash \texttt{c}(e_1\texttt{,}\ \ldots\texttt{,}\ e_n) \Downarrow \bigcup_{i=1}^n G_i, \texttt{c}(E_1\texttt{,}\ \ldots\texttt{,}\ E_n)}$$

$$\textsc{FunCall}\ \frac{(\rho, \phi \vdash e_i \Downarrow G_i, E_i)_{i=1}^n \quad \rho(f) = \texttt{fun}\ f(x_1\texttt{,}\ldots\texttt{,}x_n)\texttt{\{}e\texttt{\}} \quad \rho, \phi \vdash e[E_i/x_i]_{i=1}^n \Downarrow G, E}{\rho, \phi \vdash \texttt{f}(e_1\texttt{,}\ \ldots\texttt{,}\ e_n) \Downarrow \bigcup_{i=1}^n G_i \cup G, \mathrm{E}}$$

Figure 1. Compilation rules of STAPPL.

dictionary that maps variable names to their current values. We initialize the values of the observed variables with the corresponding values, and the unobserved variables with 0.

2. **Sampling** For each unobserved variable, we evaluate the logarithm of probability mass / density function (pmdf) and a new candidate value of the variable.

3. **Propagation** For each variable affected by the variable, we evalutate the logarithm of pmdf with and without the candidate value, and the logarithms of pmdf are used to calculate the acceptance rate α.

4. **Calculating acceptance rate** We calculate the Metropolis-Hastings acceptance rate α to decide whether to accept or reject the candidate value.

5. **Updating** We sample a random number from the uniform distribution between 0 and 1, and compare it with the acceptance rate α to decide whether to accept or reject the candidate value. If the candidate value is accepted, we update the context with the new value.

6. **Bootstrapping** We repeat the process above for sufficient steps to finish bootstrapping.

7. **Evaluation** After the bootstrapping is finished, we start evaluating the query expression with the context calculated to generate the samples of the query expression.

8. **Termination** We repeat the sampling process until the sufficient number of samples are collected.

9. **Visualization** Finally, we save the histogram built as an image.

This algorithm efficiently estimates the probability distribution of the query expression using Gibbs sampling and the Metropolis-Hastings algorithm.

5. Experiments

Our STAPPL type checker, compiler, and the inference engine is implemented with OCaml, bridged with a plotting library Seaborn in Python for visualization. You can find the source code of STAPPL at `https://github.com/Zeta611/stappl`.

We give three examples of probabilistic models written in STAPPL in Figures 4 to 6.

6. Conclusion and further work

We presented STAPPL, a statically typed probabilistic programming language. This language compiles into a deterministic graphical model, which can be efficiently used for inference. The compiled graphical model can be fed to our inference engine, which uses Metropolis-Hastings algorithm with Gibbs sampling to infer the distribution of the query

Typing of expressions. $\boxed{\Sigma, \Gamma \vdash e : \tau}$

$$\frac{}{\Sigma, \Gamma \vdash () : \mathsf{unit}^{\mathsf{val}}}\ \textsc{T-unit} \qquad \frac{}{\Sigma, \Gamma \vdash b : \mathsf{bool}^{\mathsf{val}}}\ \textsc{T-bool} \qquad \frac{}{\Sigma, \Gamma \vdash i : \mathsf{int}^{\mathsf{val}}}\ \textsc{T-int} \qquad \frac{}{\Sigma, \Gamma \vdash r : \mathsf{real}^{\mathsf{val}}}\ \textsc{T-real} \qquad \frac{\Gamma(x) = \tau}{\Sigma, \Gamma \vdash x : \tau}\ \textsc{T-var}$$

$$\frac{\Sigma, \Gamma \vdash e_1 : \tau_1^{\bullet\dagger_1} \qquad \Sigma, \Gamma \vdash e_2 : \tau_2^{\bullet\dagger_2} \qquad \oplus : \sigma \qquad \sigma \succ \left(\left(\tau_1^{\bullet\dagger_1}, \tau_2^{\bullet\dagger_2}\right) \to \tau^{\bullet(\dagger_1 \circ \dagger_2)}\right)}{\Sigma, \Gamma \vdash e_1 \oplus e_2 : \tau^{\bullet(\dagger_1 \circ \dagger_2)}}\ \textsc{T-bop}$$

$$\frac{\Sigma, \Gamma \vdash e_1 : \tau_1^{\bullet\dagger_1} \qquad \ominus : \sigma \qquad \sigma \succ \left(\tau_1^{\bullet\dagger_1} \to \tau^{\bullet\dagger_1}\right)}{\Sigma, \Gamma \vdash \ominus\ e_1 : \tau^{\bullet\dagger_1}}\ \textsc{T-uop} \qquad \frac{\left(\Sigma, \Gamma \vdash e_i : \tau\right)_{i=1}^{n}}{\Sigma, \Gamma \vdash [\overline{e_i}]_{i=1}^{n} : \tau\ \mathsf{list}}\ \textsc{T-list} \qquad \frac{\Sigma, \Gamma \vdash e_1 : \tau_1 \qquad \Sigma, \Gamma \vdash e_2 : \tau_2}{\Sigma, \Gamma \vdash e_1\texttt{;}\ e_2 : \tau_2}\ \textsc{T-seq}$$

$$\frac{\Sigma, \Gamma \vdash e_p : \mathsf{bool}^{\dagger} \qquad \Sigma, \Gamma \vdash e_c : \tau \qquad \Sigma, \Gamma \vdash e_a : \tau}{\Sigma, \Gamma \vdash \texttt{if}\ e_p\ \texttt{then}\ e_c\ \texttt{else}\ e_a : \tau}\ \textsc{T-cond} \qquad \frac{\Sigma, \Gamma \vdash e_x : \tau_x \qquad \Sigma, \Gamma\{x : \tau_x\} \vdash e : \tau}{\Sigma, \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_x\ \texttt{in}\ e : \tau}\ \textsc{T-let}$$

$$\frac{\Sigma(f) \succ \left(\left(\overline{\tau_i}\right)_{i=1}^{n} \to \tau\right) \qquad \left(\Sigma, \Gamma \vdash e_i : \tau_i\right)_{i=1}^{n}}{\Sigma, \Gamma \vdash f(\overline{e_i})_{i=1}^{n} : \tau}\ \textsc{T-fapp} \qquad \frac{p : \sigma \qquad \sigma \succ \left(\overline{\tau_i^{\bullet\dagger_i}}\right)_{i=1}^{n} \to \tau^{\bullet}\ \mathsf{dist} \qquad \left(\Sigma, \Gamma \vdash e_i : \tau_i^{\bullet\dagger_i}\right)_{i=1}^{n}}{\Sigma, \Gamma \vdash p(\overline{e_i})_{i=1}^{n} : \tau^{\bullet}\ \mathsf{dist}}\ \textsc{T-papp}$$

$$\frac{\Sigma, \Gamma \vdash e : \tau^{\bullet}\ \mathsf{dist}}{\Sigma, \Gamma \vdash \texttt{sample}(e) : \tau^{\bullet\mathsf{rv}}}\ \textsc{T-sample} \qquad \frac{\Sigma, \Gamma \vdash e_1 : \tau^{\bullet}\ \mathsf{dist} \qquad \Sigma, \Gamma \vdash e_2 : \tau^{\bullet\mathsf{val}}}{\Sigma, \Gamma \vdash \texttt{observe}(e_1\texttt{,}\ e_2) : \mathsf{unit}}\ \textsc{T-observe}$$

Typing of functions. $\boxed{\Sigma, \Gamma \vdash P : \tau^{\bullet\mathsf{rv}}}$

$$\frac{\Sigma, \Gamma\left\{\overline{x_i : \tau_i}\right\}_{i=1}^{n} \vdash e : \tau \qquad \Sigma\left\{f : \mathsf{Gen}_{\Gamma}\left(\left(\overline{\tau_i}\right)_{i=1}^{n} \to \tau\right)\right\}, \Gamma \vdash P : \tau_P^{\bullet\mathsf{rv}}}{\Sigma, \Gamma \vdash \mathsf{fun}\ f(\overline{x_i})_{i=1}^{n}\{e\}\ P : \tau_P^{\bullet\mathsf{rv}}}\ \textsc{T-fun}$$

Figure 2. Typing rules of STAPPL.

$\boxed{\mathcal{M} : \mathsf{FTyEnv} \times \mathsf{VTyEnv} \times (\mathsf{Expr} + \mathsf{Prog}) \times \mathsf{Type} \to \mathsf{Subst}}$

```
$\mathcal{M}(\Sigma, \Gamma, (), \tau) = \mathcal{U}(\tau, \mathsf{unit}^{\mathsf{val}})$
$\mathcal{M}(\Sigma, \Gamma, b, \tau) = \mathcal{U}(\tau, \mathsf{bool}^{\mathsf{val}})$
$\mathcal{M}(\Sigma, \Gamma, i, \tau) = \mathcal{U}(\tau, \mathsf{int}^{\mathsf{val}})$
$\mathcal{M}(\Sigma, \Gamma, r, \tau) = \mathcal{U}(\tau, \mathsf{real}^{\mathsf{val}})$
$\mathcal{M}(\Sigma, \Gamma, x, \tau) = \mathcal{U}(\tau, \Gamma(x))$
$\mathcal{M}(\Sigma, \Gamma, e_1 \oplus e_2, \tau) =$
    $\wr \oplus : \forall \beta_1 \beta_2 . (\tau_1^{\bullet \beta_1}, \tau_2^{\bullet \beta_2}) \to \tau^{\bullet (\beta_1 \circ \beta_2)} \int$
    let $\beta_1', \beta_2'$ = freshStamps() in
    let $s_1 = \mathcal{M}(\Sigma, \Gamma, e_1, \tau_1^{\bullet \beta_1'})$ in
    let $s_2 = \mathcal{M}(\Sigma, \Gamma s_1, e_2, \tau_2^{\bullet \beta_2'} s_2)$ in
    $s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, \ominus\ e_1, \tau) = \wr \ominus : \forall \beta_1 . \tau_1^{\bullet \beta_1} \to \tau^{\bullet \beta_1} \int$
    let $\beta_1'$ = freshStamp() in
    $\mathcal{M}(\Sigma, \Gamma, e_1, \tau_1^{\bullet \beta_1'})$
$\mathcal{M}(\Sigma, \Gamma, [\overline{e}], \tau) =$ let $\alpha$ = freshTyVar() in
    let $s = \mathcal{U}(\tau, \alpha\ \mathsf{list})$ in
    foldRight $[\overline{e}]$ $s$
        $(\lambda e . \lambda s . s \mathcal{M}(\Sigma, \Gamma s, e, \alpha s))$
$\mathcal{M}(\Sigma, \Gamma, e_1 \texttt{;}\ e_2, \tau) =$ let $\alpha_1$ = freshTyVar() in
    let $s_1 = \mathcal{M}(\Sigma, \Gamma, e_1, \alpha_1)$ in
    let $s_2 = \mathcal{M}(\Sigma, \Gamma s_1, e_2, \tau s_1)$ in
    $s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, \texttt{if}\ e_p\ \texttt{then}\ e_c\ \texttt{else}\ e_a, \tau) =$
    let $\alpha_1$ = freshTyVar() in
    let $s_1 = \mathcal{M}(\Sigma, \Gamma, e_1, \alpha_1)$ in
    let $s_2 = \mathcal{M}(\Sigma, \Gamma s_1, e_2, \tau s_1)$ in
    $s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, \texttt{let}\ x\ \texttt{=}\ e_x\ \texttt{in}\ e, \tau) =$
    let $\alpha_x$ = freshTyVar() in
    let $s_1 = \mathcal{M}(\Sigma, \Gamma, e_x, \alpha_x)$ in
    let $s_2 =$
     $\mathcal{M}(\Sigma, \Gamma s_1\{x : \alpha_x s_1\}, e, \tau s_1)$ in
    $s_1 s_2$

$\mathcal{M}(\Sigma, \Gamma, f(\overline{e}), \tau) = \wr \Sigma(f) = \forall \overline{\alpha} . \forall \overline{\beta} . \tau_f^{\to} \int$
    let $\overline{\alpha'}$ = freshTyVars() in
    let $\overline{\beta'}$ = freshStamps() in
    let $\tau_f'^{\to} = \tau_f^{\to} [\overline{\alpha'/\alpha}] [\overline{\beta'/\beta}]$ in
    let $\overline{\alpha_e}$ = freshTyVars() in
    let $s_1 = \mathcal{U}((\overline{\alpha_e}) \to \tau, \tau_f'^{\to})$ in
    let $s_2 = \mathcal{M}(\Sigma, \Gamma s_1, e_2, \tau s_1)$ in
    $s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, p(\overline{e}), \tau) = \wr p : \forall \overline{\beta} . \left(\overline{\tau_i^{\bullet \beta_i}}\right) \to \tau^{\bullet}\ \mathsf{dist} \int$
    let $\overline{\beta'}$ = freshStamps() in
    let $\tau_p = \left(\overline{\tau_i^{\bullet \beta_i'}}\right) \to \tau^{\bullet}\ \mathsf{dist}$ in
    let $\overline{\alpha_e}$ = freshTyVars() in
    let $s = \mathcal{U}((\overline{\alpha_e}) \to \tau, \tau_p)$ in
    foldRight $[\overline{e}]$ $s$
        $(\lambda e . \lambda s . s \mathcal{M}(\Sigma, \Gamma s, e, \tau s))$
$\mathcal{M}(\Sigma, \Gamma, \texttt{sample}(e), \tau) =$ let $\alpha$ = freshDTyVar() in
    let $s_1 = \mathcal{U}(\tau, \alpha^{\bullet \mathsf{rv}})$ in
    let $s_2 =$
     $\mathcal{M}(\Sigma, \Gamma s_1, e, (\alpha^{\bullet}\ \mathsf{dist}) s_1)$ in
    let $s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, \texttt{observe}(e_1\texttt{,}\ e_2), \tau) =$
    let $s = \mathcal{U}(\tau, \mathsf{unit}^{\mathsf{val}})$ in
    let $\alpha$ = freshDTyVar() in
    let $s_1 =$
     $\mathcal{M}(\Sigma, \Gamma s, e_1, (\alpha^{\bullet}\ \mathsf{dist}) s)$ in
    let $s_2 =$
     $\mathcal{M}(\Sigma, \Gamma s s_1, e_2, (\alpha^{\bullet \mathsf{val}}) s s_1)$ in
    $s s_1 s_2$
$\mathcal{M}(\Sigma, \Gamma, \texttt{fun}\ f(\overline{x})\{e\}\ P, \tau) =$
    let $\overline{\alpha}, \alpha'$ = freshTyVars() in
    let $s_1 =$
     $\mathcal{M}(\Sigma, \Gamma\{\overline{x : \alpha}\}, e, \alpha')$ in
    let $s_2 = \mathcal{M}($
       $\Sigma\{f : \mathsf{Gen}_{\Sigma}((\overline{\alpha}) \to \alpha')\},$
       $\Gamma, P, \tau s_1)$ in
    $s_1 s_2$
```

Figure 3. Type reconstruction algorithm of STAPPL.

```
let z = sample(bernoulli(0.5)) in
let mu = if !z then ~-.1.0 else 1.0 in
let d = normal(mu, 1.0) in
let y = 0.5 in
observe(d, y);
z
```

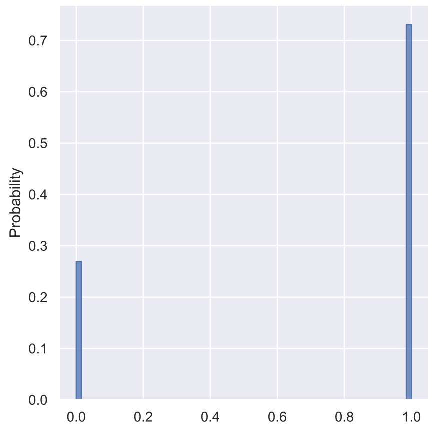

Figure 4. A simple two-component Gaussian mixture model in STAPPL.

```
fun main() {
  let x = sample(normal(0.0, 1.0)) in
  let y = bernoulli(if (x >. 1.0) then 0.9 else
   ↪  0.1) in
  observe(y, true);
  x
}

main()
```

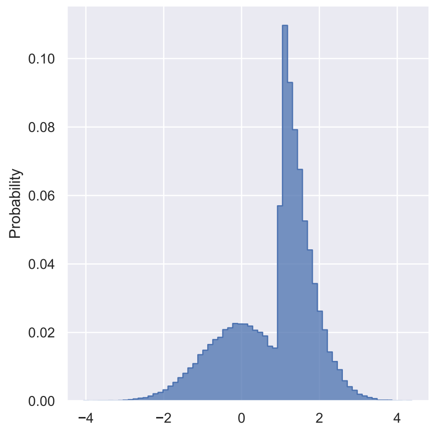

Figure 5. A simple two-component Bernoulli mixture model in STAPPL.

```
fun determine_grade(difficult, smart) {
  if difficult & smart then 0.8
    else if difficult & !smart then 0.3
    else if !difficult & smart then 0.95
    else 0.5
}

let difficult = sample(bernoulli(0.4)) in
let smart = sample(bernoulli(0.3)) in
let grade = bernoulli(determine_grade(difficult,
 ↪  smart)) in
let sat = bernoulli(
  if smart then 0.94 +. 0.01
    else 0.2
) in
observe(grade, false);
observe(sat, true);
smart
```

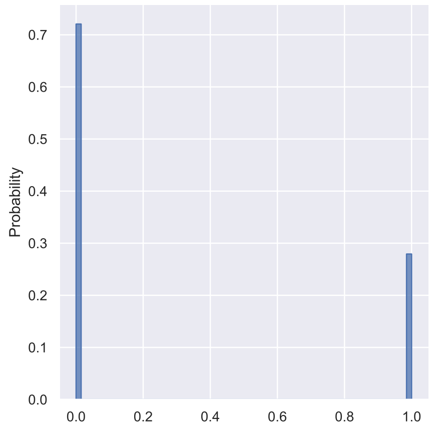

Figure 6. The Student model, adapted from (van de Meent et al., 2018), in STAPPL.

expression.

Using this statically typed language, researchers can write complex probabilistic models with confidence, knowing that the compiler will catch many common errors before execution. We believe that STAPPL will be a valuable tool for researchers in the field of probabilistic programming. Unlike previous approaches that rely on dynamic typing, STAPPL provides a safe and efficient way to define and execute probabilistic models.

Though STAPPL supports almost all kinds of probabilistic models discussed in graduate level courses, there are still some limitations. For example, STAPPL does not support higher-order functions, which are essential for some advanced and dynamic probabilistic models.

Future work will focus on extending the language to support higher-order functions and other advanced features like list and records, which were omitted in this version for simplicity. More complex but efficient inference engines can be implemented also, to support faster inference on more complex probabilistic models.

We believe that our approach that combines the safety and efficiency of static typing with the flexibility and expressiveness of probabilistic programming will be a valuable contribution to the field of probabilistic programming.

While STAPPL acts well as desired—compiling a probabilistic graph model—it could be enhanced further by adding other language features in language grammar. For instance, STAPPL is a first-order language and does not allow recursive functions. In the sense of dealing with PGM, such features are not necessary and sometimes, might even cause compilation failure. However, in the sense of dealing with probabilistic programming language, such features could be utilized well for sophisticated programming.

STAPPL could be also extended by separation of PGM and query. While STAPPL reads the whole program as target for PGM compilation and query to evaluate, for broad usage, this process could be divided into two separate input. STAPPL could store the compiled PGM beforehand, and re-use this PGM multiple times with different queries. Furthermore, if generated PGM is quite large that it takes time to compile, STAPPL could store it as an external file and upload as user's desire.

References

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, January 2019. ISSN 1532-4435.

Damas, L. and Milner, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pp. 207–212, New York, NY, USA, 1982. Association for Computing Machinery. ISBN 0897910656. doi: 10.1145/582153.582176. URL https://doi.org/10.1145/582153.582176.

Pierce, B. C. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An introduction to probabilistic programming. 2018.

Wood, F., van de Meent, J. W., and Mansinghka, V. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pp. 1024–1032, 2014.