

Watertight Probabilistic Abstractions in Python

Guillaume Baudart Avraham Shinnar Martin Hirzel Louis Mandel
IBM Research, USA
guillaume.baudart@ibm.com, shinnar@us.ibm.com, hirzel@us.ibm.com, lmandel@us.ibm.com

Abstract

There are various probabilistic modeling and inference packages for Python. Unfortunately, they either put probabilistic models in Python strings and thus lack integration benefits, or have leaky abstractions and thus are hard to code or debug. This paper introduces YAPS, which overcomes these issues by reinterpreting Python syntax to give it Stan semantics.

1 Introduction

A *probabilistic model* is a mathematical model for explaining real-world observations as being generated from latent distributions [3]. Probabilistic models can be used for machine learning, and compared to alternative approaches, have the potential to make uncertainty more overt, require less labeled training data, and improve interpretability [1]. The key *abstractions* for writing probabilistic models are *sampling* of latent variables and observations and *inference* of latent variables [4]. Probabilistic modeling is supported by several stand-alone domain-specific programming languages, e.g., Stan [2]. On the other hand, machine-learning is supported by many Python-based packages. To capitalize on Python’s packages and familiarity, PyStan [2] and other efforts [9, 11, 12] embed probabilistic abstractions into Python.

In programming, a *watertight abstraction* provides a basis for coding or debugging without *leaking* information about lower-level abstractions that it builds upon. Unfortunately, probabilistic abstractions in Python offered by existing efforts [9, 11, 12] are not watertight [1]. To code with those packages, one must also use lower-level packages such as NumPy, PyTorch, or TensorFlow. Furthermore, bugs such as tensor dimension mismatches often manifest at those lower levels and cannot be reasoned about at the probabilistic level alone.


Example. This paper introduces YAPS (Yet Another Pythonic Stan), a watertight embedding of Stan into Python. Figure 1 shows an example in a Jupyter notebook [7]. Cell 1 imports our library. Cell 2 Line 1 uses the `@yaps.model` decorator to indicate that the following function, while being syntactically Python, should be semantically reinterpreted as Stan. Since the code is reinterpreted, its original Python interpretation is no longer available, and thus, does not leak abstractions. Line 2:2 declares `coin` as a probabilistic model with one observed variable `x`, representing ten coin tosses, each of which is either tails (0) or heads (1). The

PROBPROG’18, October 4–6, 2018, Boston, MA, USA
2018.

```
In [1]: 1 import yaps
        2 from yaps.lib import *

In [2]: 1 @yaps.model
        2 def coin(x: int(lower=0, upper=1)[10]):
        3     theta: real(lower=0, upper=1) <~ uniform(0, 1)
        4     for i in range(10):
        5         x[i] <~ bernoulli(theta)

In [3]: 1 coin.graph

Out[3]: 

In [4]: 1 print(coin)

data {
  int<lower=0,upper=1> x[10];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ uniform(0,1);
  for (i in 1:10)
    x[i] ~ bernoulli(theta);
}

In [5]: 1 flips = [0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
        2 posterior = yaps.infer(coin(x=flips), iter=1000)
        3 print("theta: {:.3f}".format(posterior.theta.mean()))

INFO:pystan:COMPILING THE C++ CODE FOR MODEL coin_1d59
d81653ce56f864b633a5388cb394 NOW.

theta: 0.249
```

Figure 1. Coin example in YAPS.

type `int(lower=0, upper=1)[10]` comes from Stan, where it is used for informative error messages and compiler optimizations. Line 2:3 declares a latent variable `theta` for the unknown bias of the coin. The initialization `<~ uniform(0, 1)` samples `theta` with the prior belief that any bias is equally likely. Lines 2:4–2:5 indicate that each of the coin tosses `x[i]` is sampled from a Bernoulli distribution with the same latent `theta` parameter.

To reinterpret Python syntax, we must first parse it, and once parsed, we can easily visualize its dependencies. Cell 3 yields a visual rendering of the graphical model. YAPS compiles to Stan code, and Cell 4 prints the result, showing Stan’s explicit code blocks for observations (`data`), latent variables (`parameters`), and the actual model. In Python, we opted for a more concise (but equally watertight) syntax with implicit blocks inspired by SlicStan [5].

While watertight abstractions are essential for modeling, interaction with the host language is essential for inference.

Cell 5 Line 1 contains concrete observed coin flips. Line 5:2 calls inference, passing observed data as an actual argument to the model to infer a joint posterior distribution. YAPS creates a posterior object with fields for latent variables such as `theta`. Line 5:3 prints the results: the inferred posterior belief is that the coin is biased towards tails (`theta` is 0.249).

Related Work. Compared to PyStan [2], which reads Stan code from a file or from multi-line Python strings, YAPS is more deeply embedded. Compared to PyMC3 [9], Edward [11], or Pyro [12], YAPS is more watertight. Those efforts use *lazy evaluation*: they overload operators that appear to do eager computation to instead generate a computational graph to be evaluated during inference. Lazy evaluation is a popular approach for embedding domain-specific languages into general-purpose host languages [6, 8]. Unfortunately, lazy evaluation in Python does not track local variable names, does not cleanly isolate the embedded language, and leads to verbose syntax that is less similar to stand-alone probabilistic languages. In contrast, the reinterpretation approach of YAPS avoids those disadvantages. Both approaches (lazy evaluation and reinterpretation) have the advantage of working in pure Python without any separate preprocessor.

2 Design

Language design and rationale. The design of YAPS follows the principle “Stan-like for probabilistic features, Python-like for everything else”. YAPS uses familiar Python syntax for non-probabilistic features such as for loops, type declarations, or function declarations. Stan-specific features are expressed with syntactically-valid Python syntax that resembles the original syntax: `<~` for sampling or `x.T[a, b]` for truncated distribution. Since observed variables are free during modeling but bound during inference, YAPS make them formal arguments of the model (Figure 1 Cell 2 Line 2) and actual arguments of the inference call (Cell 5 Line 2). This is one place where reinterpretation enabled more intuitive syntax. Whereas Stan requires users to place functions, parameters, transformed parameters, model, transformed data, and generated quantities into separate code blocks, YAPS uses SlicStan-style program analysis [5] to let users place all of these at the top-level in the function. This syntax is more concise and flexible and SlicStan shows it can even improve modularity. For users who prefer explicit blocks, YAPS offers a syntax based on Python with statements.

Implementation difficulties and solutions. Writing our own embedded Python parser would have been cumbersome, but fortunately, the Python standard library modules `inspect` and `ast` solved that for us. Our `@yaps.model` decorator uses those modules to replace the Python function by an intermediate representation suitable for visualization, compilation to Stan, and inference. One difficulty was to implement a suitable syntax for sampling. We first tried `=~`, where `=` is

```
In [2]: 1 @yaps.model
        2 def coin(x: int(lower=0, upper=1)[10]):
        3     for i in range(10):
        4         x[i] <- bernoulli(theta)

In [3]: 1 flips = [0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
        2 posterior = yaps.infer(coin(x=flips), iter=1000)

ror message:\n{}".format(model_name, msg)
--> 134     raise ValueError(error_msg)
      135     elif result['status'] == 0: # SUCCESS_RC is 0
      136         logger.debug("Successfully parsed Stan model '{}'"
                          "'.format(model_name))

ValueError: Failed to parse Stan model 'coin_8d88be73678b3bb643e
64f513233b9a0'. Error message:
SYNTAX ERROR, MESSAGE(S) FROM PARSER:

variable "theta" does not exist.
error in '<ipython-input-2-60cc75e8a33e>' at line 4, column 27

-----
1: @yaps.model
2: def coin(x: int(lower=0, upper=1)[10]):
3:     for i in range(10):
4:         x[i] <- bernoulli(theta)
                                     ^
-----
```

Figure 2. Error reporting example in YAPS.

Python’s assignment, but that was insufficient, because the left-hand side of Stan’s sampling is more expressive than that of Python’s assignment. Hence we settled on `<~` instead, which we substitute with `is` before parsing, since that does not occur in Stan and has a low precedence in Python.

Another difficulty was name resolution. Identifiers in YAPS code refer to Stan types, functions, and distributions; for watertight abstractions, they should not be resolved to Python entities. The Python interpreter does not attempt to resolve names in YAPS function bodies. The Python interpreter does report errors for unknown names in function signatures, e.g., `def model2(N: int, y: real[N])`. To avoid that, YAPS provides stubs (e.g., `int`, `real`) and enables users to declare other required identifiers (e.g. `N = yaps.dependent_type_var()`).

Finally, building a robust interface between Python and the Stan compiler would also have been cumbersome, but fortunately, PyStan solved that for us. The only difficulty was that PyStan error messages refer to locations in generated Stan code. We implemented a reverse source location mapping and used that to make error messages refer to source locations in YAPS code instead. Figure 2 shows an example error message, obtained by mapping an error from PyStan back into the original Python syntax.

Results. In addition to the YAPS compiler, we implemented a tool that generates YAPS syntax from Stan source code. We were thus able to test our prototype on 62 complete examples extracted from the Stan manual [10].

3 Conclusion

This paper introduces YAPS and demonstrates that it is tightly embedded in Python and watertight. It is possible to write a rigorous and self-contained specification for the language of YAPS models, and in fact, we are planning to do that in future work.

References

- [1] Guillaume Baudart, Martin Hirzel, and Louis Mandel. 2018. Deep Probabilistic Programming Languages: A Qualitative Study. (2018). <https://arxiv.org/abs/1804.06458>
- [2] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–37. <https://www.jstatsoft.org/article/view/v076i01>
- [3] Zoubin Ghahramani. 2015. Probabilistic Machine Learning and Artificial Intelligence. *Nature* 521, 7553 (May 2015), 452–459. <https://www.nature.com/articles/nature14541>
- [4] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sri-ram K. Rajamani. 2014. Probabilistic Programming. In *ICSE track on Future of Software Engineering (FOSE)*. 167–181. <https://doi.org/10.1145/2593882.2593900>
- [5] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2018. SlicStan: Improving Probabilistic Programming using Information Flow Analysis. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems (PPS)*. <https://pps2018.soic.indiana.edu/files/2017/12/SlicStanPPS.pdf>
- [6] Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *International Conference on Software Reuse (ICSR)*. 134–142. <https://doi.org/10.1109/ICSR.1998.685738>
- [7] Fernando Pérez. 2014. Project Jupyter. (2014). <http://jupyter.org/> (Retrieved July 2018).
- [8] Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Communications of the ACM (CACM)* 55 (2012), 121–130. Issue 6. <https://doi.org/10.1145/2184319.2184345>
- [9] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2015. Probabilistic Programming in Python Using PyMC3. (2015). <https://arxiv.org/abs/1507.08050>
- [10] Stan Development Team. 2017. *Stan Modeling Language, User's Guide and Reference Manual*. <https://github.com/stan-dev/stan/releases/download/v2.17.0/stan-reference-2.17.0.pdf> Stan Version 2.17.0.
- [11] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/1701.03757>
- [12] Uber. 2017. Pyro. (2017). <http://pyro.ai/> (Retrieved July 2018).