# INSOMNIA: Types and Modules for Probabilistic Programming [*]

Extended Abstract

Aleksey Kliger
akliger@gmail.com

Sean Stromsten
BAE Systems
sean.stromsten@baesystems.com

## 1. Introduction

Probabilistic programming languages partition the challenge of Bayesian machine learning into the separate tasks of modeling and inference [1, 2, 5, 7, 10]. Users of such languages focus primarily on modeling, leaving inference to the language developers. Unfettered by concerns about inference, modelers will want to create big, complex probabilistic programs, which will present familiar problems: Models may become too big to comprehend, debug or validate as wholes. Users building large applications will need to divide and coordinate work among several people and will need to reuse common elements of models without rebuilding them. We hypothesize that model creation will benefit from two linguistic tools that have proven helpful in deterministic programming: types and modules. We have combined probabilistic modeling, types, and modules in a design called INSOMNIA.

INSOMNIA is a statically typed language inspired by Standard ML [8]. INSOMNIA's models generalize ML's modules, and INSOMNIA uses "model types" (a generalization of ML signatures) to classify probabilistic models and parts thereof. To formalize common modeling patterns, INSOMNIA can abstract over a model described by a model type, just as a Standard ML functor can abstract over a module described by a signature.

The benefits of types and modules are well known. In the probabilistic setting, we are especially interested in using static types to distinguish deterministic computations (denoted by modules) from probabilistic ones (denoted by models). We are also interested in enabling programmers to build deterministic structures that have probabilistic components and vice versa. In this abstract we show only some very simple, contrived examples. The real benefits of types and modules are in larger problems such as tracking, where we might want to develop models for vehicle dynamics and for sensors separately, or to swap in different models of each type.

In Standard ML, a module denotes a collection of *components*. A component may be a core-language value, a type, or a submodule. INSOMNIA adds *models*; a model denotes a *joint distribution* over its components. An INSOMNIA model has a principal type which can be inferred. Explicit model types can be ascribed in order to ease the cognitive burden of understanding large models. The model type can describe all components of the model, but a programmer may also ascribe a smaller model type (a supertype) to show what subset of components are intended to be observed or queried.

An INSOMNIA submodel itself denotes a distribution, so a model that contains a submodel denotes a distribution of distributions. We use this idiom to represent a probabilistic experiment that can be repeated an arbitrary number of times. The outcome of such an experiment, when observed, provides evidence for the posterior distribution of latent parameters of the outer model. The idiom avoids conflating such an experiment with a sequence or a function—the type system guarantees that arbitrarily many outcomes can be observed, and that conditioned on their parameters, they are independent and identically distributed.

We have defined INSOMNIA's semantics by elaboration into $F_\omega^{\mathsf{meas}}$, which is $F_\omega$ extended with a distribution monad [11]. Our elaboration extends the approach of Rossberg, Dreyer and Russo [12] for the fragment of ML with only generative functors. Elaboration yields a typed monadic probabilistic program, which at present is type-erased and then translated to Gamble [3], a probabilistic programming language in the Racket family [4]. Other back ends are left to future work.

## 2. Probabilistic modeling with INSOMNIA

INSOMNIA brings modules and static types to probabilistic programming. Our original idea was to extend the deterministic ML module system with probabilistic components. An ordinary ML module has two species of components: type components and value components. We explored adding a third species of component: random-variable components. The idea was that the denotation of a module would include a joint distribution over all its random-variable components. We hope to explore the tradeoff between usability and the more complex elaboration of this design into $F_\omega^{\mathsf{meas}}$ in future work. In the present version of Insomnia we add distributions to the module level:

- A *module* is like an ML module: it has type components, value components, and submodules, and it is described by a *signature* (also called *module type*). The denotation of its value part is a record of values.
- A *model* is syntactically similar to a module. It has type components, value components, and submodel, and it is described by a *model type*. But its denotation is a distribution over modules.

A pleasing consequence of this design is a submodel represents a family of exchangeable experiments.

### 2.1 Models as distributions over modules

A *module* groups core-language types, values and submodules. Each module may be ascribed one or more *signatures* of varying specificity. A signature, which serves as an interface to the module, specifies the kinds (and optionally the definitions) of core-language types, the types of core-language values, and the signatures of submodules. For example, the signature `COIN_BIAS` promises two

[*] This material is based upon work sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-14-C-0002. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

core-language values `coin1_bias` and `coin2_bias`, both real numbers.

```
COIN_BIAS = module type {
  sig coin1_bias : Real
  sig coin2_bias : Real
}
```

Module `CoinBias` implements `COIN_BIAS`.

```
CoinBias : COIN_BIAS = module {
  val coin1_bias = 0.5
  val coin2_bias = 0.9
}
```

A *model* denotes a distribution over modules. Just as a module's value component is a record of values, a model's value component is a joint distribution over such records. The model `TwoCoins` denotes a joint distribution over two coins with different biases:[1]

```
TwoCoins = model {
  val coin1 ~ flip CoinBias.coin1_bias
  val coin2 ~ flip CoinBias.coin2_bias
}
```

The ~ syntax (from BUGS [6]) shows that the binding of `coin1` is monadic: on the right, the `flip` expression has type "distribution of Boolean," but the bound variable `coin1` has type Boolean.

The monadic bind may also be used to compose models. In the next example we first bind a submodule `C` drawn from the model `TwoCoins`. We then define `r`, whose distribution is uniform over a range conditioned on the results of the flips of `C.coin1` and `C.coin2`.

```
M = model {
  C ~ TwoCoins
  val r ~ uniform (if C.coin1 then -1.0 else -2.0)
                  (if C.coin2 then 1.0 else 2.0)
}
```

The resulting model denotes a joint distribution over modules with three components: two booleans and a real.

### 2.2 Nested models as exchangeable experiments

Quite often, probabilistic programmers wish to infer a posterior distribution from a set of observations whose size is not known in advance. The process producing such observations needs to describe a probabistic experiment that can be repeated arbitrarily many times. If the result of such an experiment has type $\tau$, the experiment can be represented as an infinite sequence of values of type $\tau$, or as a function from natural numbers (or some other index set) to $\tau$. But these encodings don't capture a key property of such experiments: the results are *exchangeable* and in fact each result is independent of the position of the result within the indexed set. In INSOMNIA, we want to capture this property in the type system, and we already have a suitable tool: the ability to conduct a probabilistic experiment arbitrarily many times is denoted by a *distribution* over values of type $\tau$. Distributions are expressed by models, and we therefore encode such experiments as submodels.

As an example, we develop a model of the following scenario:

1. A fair coin $A$ is flipped.
2. Based on the outcome of $A$, one of two biased coins $B_1$ or $B_2$ is chosen and is called $B$.
3. The outcomes of repeatedly flipping the chosen coin $B$ are observed.

[1] Because `TwoCoins` is not explicitly ascribed a model type, the INSOMNIA compiler infers its principal model type.

Steps 1 and 2 are modeled by `FlipAChooseB`:

```
FlipAChooseB = model {
  val a_outcome ~ flip 0.5
  val b_bias = if a_outcome then 0.9 else 0.1
}
```

`FlipAChooseB` denotes a model, which is a distribution over modules, but our experiment is parameterized over a single value drawn from this distribution, which is expressed using signature `B_BIAS`:

```
B_BIAS = module type {  sig b_bias : Real  }
```

Now we represent step 3 using the `FlipBCoin` functor, which returns a model:

```
FlipBCoin = (X : B_BIAS) -> model {
  val outcome ~ flip X.b_bias
}
```

Finally the model `Experiment`, which has model type `EXPERIMENT`, forms the joint distribution of the $A$ coin and a nested model of an exchangeable sequence of $B$ experiments:

```
EXPERIMENT = model type {
  A      : module type { sig a_outcome : Bool }
  BFlips : model  type { sig outcome   : Bool }
}
Experiment : EXPERIMENT = model {
  A ~ FlipAChooseB
  BFlips = FlipBCoin (A)
}
```

`BFlips` is not a single module sampled from the application of the $B$ flip functor; it is the model itself. Each sample of the submodel would be a new coin flip with its own outcome.

## 3. Conclusions and Future Work

We believe that static types can clarify the structure of models, and modules will help probabilistic programmers build large models compositionally from reusable model fragments. We expect programmers to compose large models from smaller models, and from functors that return models. We expect model types both to lighten the cognitive load of understanding models and their components, and also to guarantee that composition succeeds.

An alternative approach is to embed probabilistic programming into a larger setting that already provides static types, and perhaps a module or object system. Languages such as Figaro [10], which is embedded in Scala, and Hakaru [9], which is embedded in Haskell, embody this approach.

Our early results show that the ML module system may be extended with *models*. A model is a kind of monadic module that can be given meaning by elaboration into the polymorphic stochastic lambda calculus using an extension of the approach of Rossberg et al. [12]. With INSOMNIA we hope to provide the best of two worlds: to help with the "democratization of machine learning," we provide a language with abstractions for modular development of complex probabilistic programs. To help with analysis and understanding of models' properties, we define INSOMNIA by elaboration into a small core calculus whose semantics may be studied further. With these basics in place, future work may explore how best to leverage and extend the module system to account for peculiarities of probabilistic programming.

## Acknowledgments

The original design of INSOMNIA, in which each module had three species of components, owes a great deal to Norman Ramsey. For help with the work and the paper, we thank Norman Ramsey, Mitch Wand, Andrew Cobb, Ryan Culpepper, Greg Sullivan, and Theo Giannakopoulos.

## References

[1] *Stan Modeling Language Users Guide and Reference Manual, Version 2.8.0*, 2015. URL http://mc-stan.org/.

[2] J. Borgström, A. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for bayesian machine learning. In G. Barthe, editor, *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 77–96. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19717-8. DOI:10.1007/978-3-642-19718-5_5. URL http://dx.doi.org/10.1007/978-3-642-19718-5_5.

[3] R. Culpepper. *Gamble*, 2015. URL https://rmculpepper.github.io/gamble/.

[4] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/.

[5] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *In UAI*, pages 220–229, 2008.

[6] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistics in medicine*, 28(25): 3049, 2009.

[7] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*, Mar. 2014.

[8] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, MA, USA, 1997. ISBN 9780262631815.

[9] P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. *Probabilistic inference by program transformation in Hakaru (system description)*, 2015.

[10] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[11] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'02, pages 154–165, 2002. DOI:10.1145/503272.503288.

[12] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, 24:529–607, 2014. ISSN 1469-7653. DOI:10.1017/S0956796814000264. URL http://journals.cambridge.org/article_S0956796814000264.