HOL Theorem Prover Case Study: Verifying Probabilistic Programs

Joe Hurd
joe.hurd@cl.cam.ac.uk

University of Cambridge
The HOL Theorem Prover

- Developed by Mike Gordon’s Hardware Verification Group in Cambridge, first release was HOL88.
- Implements classical Higher-Order Logic with Hindley-Milner polymorphism.
- Sprung from the Edinburgh LCF project, so has a small logical kernel to ensure soundness.
- Links to external proof tools, either as oracles (e.g., SAT solvers) or by translating their proofs (e.g., Gandalf).
- Comes with a large library of theorems contributed by many users over the years, including theories of lists, real analysis, groups etc.
To verify a probabilistic algorithm in HOL:

\[ \text{E} \vdash P(\text{B}1) \]

\[ P \vdash \text{E!} \]

\[ \text{R} \]

and model the probabilistic algorithm in the logic:

\[ \text{prob}_\text{program} : \text{N!B}1!f\text{success, failure}^g\text{B}1 \]

\[ \text{success} \]

\[ \text{failure} \]

\[ \text{ns} \]

\[ \text{fst} \]

\[ \text{prove} \]

\[ \text{that the algorithm satisfies its specification.} \]
To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;

\[ \mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^\infty)), \quad P : \mathcal{E} \rightarrow \mathbb{R} \]
To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;
  \[ \mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^\infty)), \quad P : \mathcal{E} \rightarrow \mathbb{R} \]

- and model the probabilistic algorithm in the logic;
  \[ \text{prob}_\text{program} : \mathbb{N} \rightarrow \mathbb{B}^\infty \rightarrow \{\text{success, failure}\} \times \mathbb{B}^\infty \]
To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;

\[ \mathcal{E} : \mathcal{P}(\mathbb{B}^\infty), \quad \mathbb{P} : \mathcal{E} \rightarrow \mathbb{R} \]

- and model the probabilistic algorithm in the logic;

\[ \text{prob}_{\text{program}} : \mathbb{N} \rightarrow \mathbb{B}^\infty \rightarrow \{\text{success}, \text{failure}\} \times \mathbb{B}^\infty \]

- then finally prove that the algorithm satisfies its specification.

\[ \vdash \forall n. \mathbb{P} \{ s \mid \text{fst (prob}_{\text{program}} n s) = \text{failure} \} \leq 2^{-n} \]
Formalizing Probability

• Need to construct a probability space of Bernoulli($\frac{1}{2}$) sequences, to give meaning to such terms as

$$\mathbb{P} \{ s \mid \text{fst (prob_program n s)} = \text{failure} \}$$

• To ensure soundness, would like it to be a purely definitional extension of HOL (no axioms).

• Use measure theory, and end up with a set $\mathcal{E}$ of events and a probability function $\mathbb{P}$:

$$\mathcal{E} = \{ S \subset \mathbb{B}^\infty \mid S \text{ is a measurable set} \}$$

$$\mathbb{P}(S) = \text{the probability measure of } S \text{ (for } S \in \mathcal{E})$$
Suppose a probabilistic ‘function’:
\[ \hat{f} : \alpha \rightarrow \beta \]

Model \( \hat{f} \) with a higher-order logic function
\[ f : \alpha \rightarrow \mathbb{B}^\infty \rightarrow \beta \times \mathbb{B}^\infty \]
that passes around ‘an infinite sequence of coin-flips.’

The probability that \( \hat{f}(a) \) meets a specification
\[ B : \beta \rightarrow \mathbb{B} \] can then be formally defined as
\[ \mathbb{P}\{s \mid B(\text{fst} (f \ a \ s))\} \]
Modelling Probabilistic Algorithms

- Can use state-transformer monadic notation to express HOL models of probabilistic algorithms:

  \[
  \text{unit } a \equiv \lambda s. (a, s) \\
  \text{bind } f \; g \equiv \lambda s. \text{let } (x, s') \leftarrow f(s) \text{ in } g \; x \; s'
  \]

- For example, if \( \text{dice} \) is a program that generates a dice throw from a sequence of coin flips, then

  \[
  \text{two\_dice} = \text{bind } \text{dice} (\lambda x. \text{bind } \text{dice} (\lambda y. \text{unit} (x + y)))
  \]

  generates the sum of two dice.
Example: The Binomial Distribution

- Definition of a sampling algorithm for the binomial distribution:

\[
\begin{align*}
\triangleright & \quad \text{bit} = \lambda s. \ (\text{if shd} \ s \ \text{then} \ 1 \ \text{else} \ 0, \ \text{stl} \ s) \\
\triangleright & \quad \text{bin} \ 0 = \text{unit} \ 0 \ \land \\
\forall \ n. & \quad \text{bin} \ (\text{suc} \ n) = \\
& \quad \text{bind} \ \text{bit} \ (\lambda x. \ \text{bind} \ (\text{bin} \ n) \ (\lambda y. \ \text{unit} \ (x + y)))
\end{align*}
\]

- Correctness theorem:

\[
\triangleright \quad \forall n, r. \ \mathbb{P} \{ s \mid \text{fst} \ (\text{bin} \ n \ s) = r \} = \binom{n}{r} \left(\frac{1}{2}\right)^n
\]
Example: A Dice Program

A dice program, due to Knuth (1976):

dice =
coin_flip
(prob_repeat
  (coin_flip
   (coin_flip
    (coin_flip
      (unit none)
      (unit (some 1))))
    (mmap some
     (coin_flip
      (unit 2)
      (unit 3))))))
  (prob_repeat
   (coin_flip
     (mmap some
      (coin_flip
       (unit 4)
       (unit 5))))
    (coin_flip
     (unit (some 6))
     (unit none))))

Comparison: Prism Model Checker

- Prism is a probabilistic model checker developed by Kwiatkowska et. al. at the University of Birmingham.
- Prism version of dice program:

```
probabilistic
module dice
    s : [0..7] init 0;  // local state
    d : [0..6] init 0;  // value of the dice
    [] s=0 -> 0.5 : s'=1 + 0.5 : s'=2;
    [] s=1 -> 0.5 : s'=3 + 0.5 : s'=4;
    [] s=2 -> 0.5 : s'=5 + 0.5 : s'=6;
    [] s=3 -> 0.5 : s'=1 + 0.5 : s'=7 & d'=1;
    [] s=4 -> 0.5 : s'=7 & d'=2 + 0.5 : s'=7 & d'=3;
    [] s=5 -> 0.5 : s'=7 & d'=4 + 0.5 : s'=7 & d'=5;
    [] s=6 -> 0.5 : s'=2 + 0.5 : s'=7 & d'=6;
    [] s=7 -> s'=7;
endmodule
```
Prism automatically evaluates the result probabilities in less than a second:

$$P \left[ \text{true U s=7 & d=k} \right] = 0.166666...$$

For each $$k = 1, \ldots, 6$$, result accurate to 6 decimal places.

HOL correctness theorem spans $$\sim 100$$ lines of interactive proof script:

$$\vdash \forall n. \mathbb{P} \{ s \mid \text{fst (dice s)} = n \} = \text{if } 1 \leq n \leq 6 \text{ then } \frac{1}{6} \text{ else } 0$$
Comparison: Prism Model Checker

This program calculates the sum of two dice.

**HOL:** large term, clumsy

**Prism:** concise, automatic

\[ \forall n. \]
\[ \mathbb{P}\{s \mid \text{fst (two_dice } s) = n\} = \]
\[ \text{if } n = 2 \lor n = 12 \text{ then } \frac{1}{36} \]
\[ \text{else if } n = 3 \lor n = 11 \text{ then } \frac{2}{36} \]
\[ \text{else if } n = 4 \lor n = 10 \text{ then } \frac{3}{36} \]
\[ \text{else if } n = 5 \lor n = 9 \text{ then } \frac{4}{36} \]
\[ \text{else if } n = 6 \lor n = 8 \text{ then } \frac{5}{36} \]
\[ \text{else if } n = 7 \text{ then } \frac{6}{36} \]
\[ \text{else } 0 \]
Comparison: Prism Model Checker

- Probabilistic model checkers (such as Prism)
  - have automatic operation,
  - but can only verify probabilistic finite state automata.
  - Perhaps better suited as an embedded verification tool, in a compiler or program synthesizer?

- Theorem provers (such as HOL)
  - require interactive proof,
  - but can represent any probabilistic program.
  - Perhaps better suited for ‘one-off’ verifications of textbook probabilistic algorithms?
Example: Miller-Rabin Primality Test

The Miller-Rabin algorithm is a probabilistic primality test, used by commercial software such as Mathematica.

Can verify the test using our HOL model of probabilistic programs:

$$\forall n, t, s. \text{ prime } n \Rightarrow \text{fst } (\text{miller } n \ t \ s) = \top$$
$$\forall n, t. \text{ ~prime } n \Rightarrow 1 - 2^{-t} \leq \mathbb{P} \{ s \mid \text{fst } (\text{miller } n \ t \ s) = \bot \}$$

Here $n$ is the number to test for primality, and $t$ is the maximum number of iterations allowed.

Took $\sim 1000$ lines of interactive proof script.
Comparison: Coq Theorem Prover

- Coq theorem prover for constructive logic, developed by Barras et. al. at INRIA, France.
- Recent work by Paulin, Audebaud and Lassaigne allows probabilistic programs to be formalized in Coq.
- Model uses the probability distribution monad

\[ \hat{\tau} = (\tau \rightarrow [0, 1]) \rightarrow [0, 1] \]

\[
\text{flip}: \hat{\mathbb{B}} := \lambda f : \mathbb{B} \rightarrow [0, 1]. \ f(\top)/2 + f(\bot)/2
\]

\[
x +_p y : \hat{\tau} := \lambda f : \tau \rightarrow [0, 1]. \ p(x(f)) + (1 - p)(y(f))
\]

\[
\text{random}(n): \hat{\mathbb{Z}} := \lambda f : \mathbb{Z} \rightarrow [0, 1]. \ \sum_{1 \leq i \leq n} f(i)/n
\]
Comparison: Coq Theorem Prover

Can model the Miller-Rabin test in Coq:

\[
\text{witness } n \ a := a^s \equiv 1 \pmod{n} \lor \exists j. 0 \leq j < r \land a^{2^j s} \equiv -1 \pmod{n}
\]

(where \( n - 1 = 2^r s \), and \( s \) odd)

\[
\text{miller } n \ t := \begin{cases} 
\text{unit} & \text{if } n = 0 \\
\text{else} & \text{bind (bind (random } n \ 1) (\lambda a. \text{unit } (\text{witness } n \ a)) \text{) } (\lambda b. \text{if } b \text{ then } \text{miller } n \ (t - 1) \text{ else } \text{unit} \bot) 
\end{cases}
\]

Meta-language evaluation of \text{miller 9 3} shows that the probability that 9 is declared composite is 98.4375%.
Comparison: Coq Theorem Prover

- The Coq theorem prover
  - can execute probabilistic programs using fast meta-level evaluation,
  - but measure theory is hard in constructive logic.
  - Perhaps better suited for high-assurance calculations of probabilities and expectations?

- The HOL theorem prover
  - is slow to execute programs inside the logic,
  - but contains a formalized measure theory ready to verify probabilistic programs.
  - Perhaps better suited for outright verification of probabilistic programs?
And Finally

Slides for this talk available at:

http://www.cl.cam.ac.uk/~jeh1004/research/talks/