# 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.

# The HOL Theorem Prover

To verify a probabilistic algorithm in HOL:

# The HOL Theorem Prover

To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;

$$\mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^\infty)), \quad \mathbb{P} : \mathcal{E} \to \mathbb{R}$$

# The HOL Theorem Prover

To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;

$$\mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^\infty)), \quad \mathbb{P} : \mathcal{E} \to \mathbb{R}$$

- and model the probabilistic algorithm in the logic;

$$\mathsf{prob\_program} : \mathbb{N} \to \mathbb{B}^\infty \to \{\mathsf{success}, \mathsf{failure}\} \times \mathbb{B}^\infty$$

# The HOL Theorem Prover

To verify a probabilistic algorithm in HOL:

- Must be able to formalize its probabilistic specification;

$$\mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^\infty)), \quad \mathbb{P} : \mathcal{E} \to \mathbb{R}$$

- and model the probabilistic algorithm in the logic;

$$\mathsf{prob\_program} : \mathbb{N} \to \mathbb{B}^\infty \to \{\mathsf{success}, \mathsf{failure}\} \times \mathbb{B}^\infty$$

- then finally prove that the algorithm satisfies its specification.

$$\vdash \forall n.\ \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{prob\_program}\ n\ s) = \mathsf{failure}\} \le 2^{-n}$$

# Formalizing Probability

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

$$\mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{prob\_program}\ n\ s) = \mathsf{failure}\}$$

- 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 $\mathbb{P}$:

$$\begin{aligned} \mathcal{E} &= \{S \subset \mathbb{B}^\infty \mid S \text{ is a measurable set}\} \\ \mathbb{P}(S) &= \text{the probability measure of } S \text{ (for } S \in \mathcal{E}) \end{aligned}$$

# Modelling Probabilistic Algorithms

- Suppose a probabilistic 'function':

$$\hat{f} : \alpha \to \beta$$

- Model $\hat{f}$ with a higher-order logic function

$$f : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

that passes around 'an infinite sequence of coin-flips.'

- The probability that $\hat{f}(a)$ meets a specification $B : \beta \to \mathbb{B}$ can then be formally defined as

$$\mathbb{P}\,\{s \mid B(\mathsf{fst}\ (f\ a\ s))\}$$

# Modelling Probabilistic Algorithms

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

$$\begin{aligned} \mathsf{unit}\ a &= \lambda\, s.\ (a, s) \\ \mathsf{bind}\ f\ g &= \lambda\, s.\ \mathsf{let}\ (x, s') \leftarrow f(s)\ \mathsf{in}\ g\ x\ s' \end{aligned}$$

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

$$\mathsf{two\_dice} = \mathsf{bind\ dice}\ (\lambda\, x.\ \mathsf{bind\ dice}\ (\lambda\, y.\ \mathsf{unit}\ (x + y)))$$

generates the sum of two dice.

# Example: The Binomial Distribution

- Definition of a sampling algorithm for the binomial distribution:

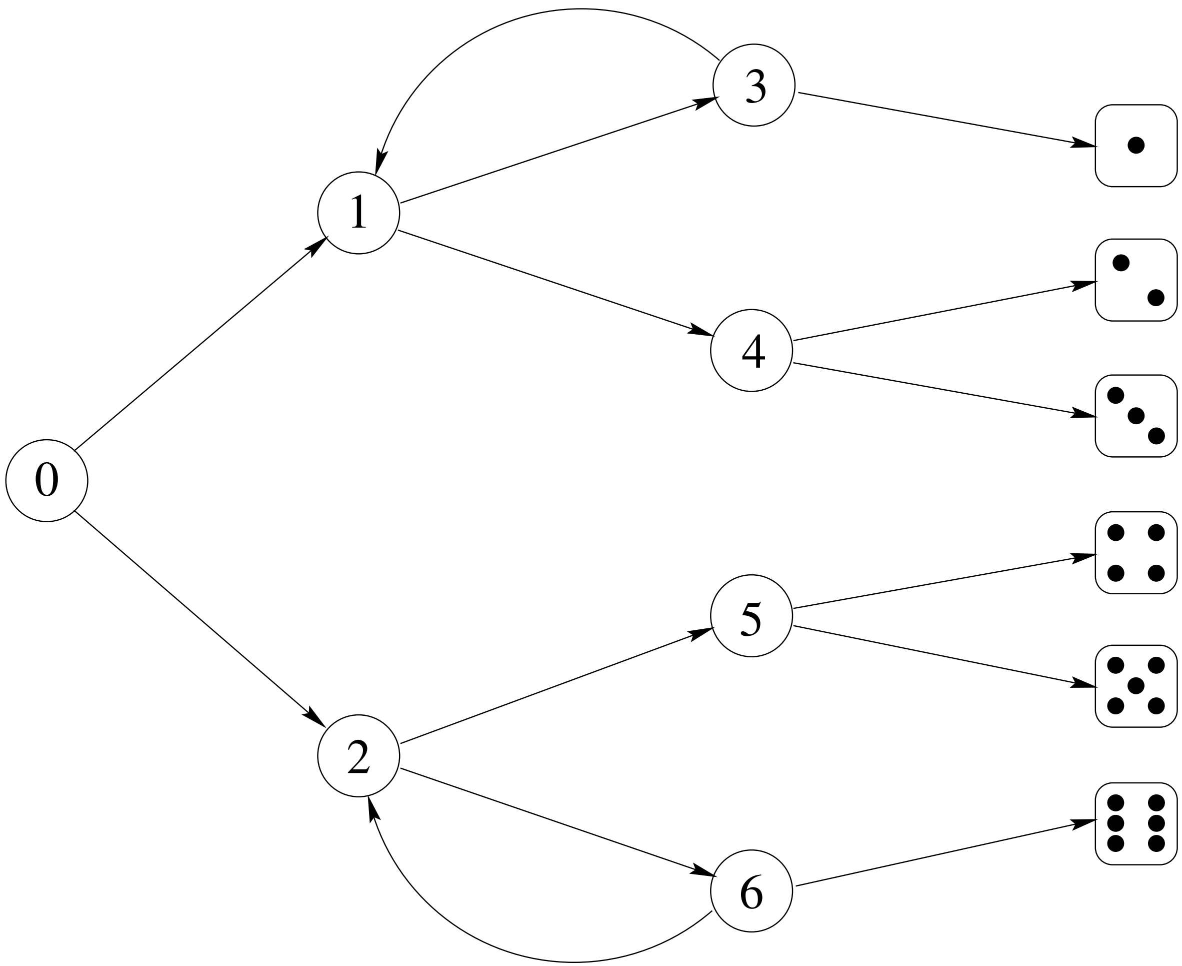
$$
\begin{array}{ll}
\vdash & \mathsf{bit} = \lambda s.\ (\mathsf{if\ shd}\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0,\ \mathsf{stl}\ s) \\
\vdash & \mathsf{bin}\ 0 = \mathsf{unit}\ 0\ \wedge \\
& \forall\, n. \\
& \quad \mathsf{bin}\ (\mathsf{suc}\ n) = \\
& \quad \mathsf{bind\ bit}\ (\lambda\, x.\ \mathsf{bind}\ (\mathsf{bin}\ n)\ (\lambda\, y.\ \mathsf{unit}\ (x + y)))
\end{array}
$$

- Correctness theorem:

$$\vdash\ \forall\, n, r.\ \mathbb{P}\, \{s \mid \mathsf{fst}\ (\mathsf{bin}\ n\ s) = r\} = \binom{n}{r} \left(\tfrac{1}{2}\right)^n$$

# Example: A Dice Program

A dice program, due to Knuth (1976):




```
dice  =
coin_flip
(prob_repeat
  (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 probabalistic 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
```

# Comparison: Prism Model Checker

Prism automatically evaluates the result probabilities in less than a second:

```
P [ true U s=7 & d=k ] = 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:

$$\vdash \ \forall n.\, \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{dice}\ s) = n\} = \mathsf{if}\ 1 \le n \le 6\ \mathsf{then}\ \tfrac{1}{6}\ \mathsf{else}\ 0$$

# Comparison: Prism Model Checker

This program calculates the sum of two dice.

**HOL:** large term, clumsy

**Prism:** concise, automatic

$$\vdash \quad \forall n.$$

$$\mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{two\_dice}\ s) = n\} =$$

$$\text{if } n = 2 \lor n = 12 \text{ then } \tfrac{1}{36}$$

$$\text{else if } n = 3 \lor n = 11 \text{ then } \tfrac{2}{36}$$

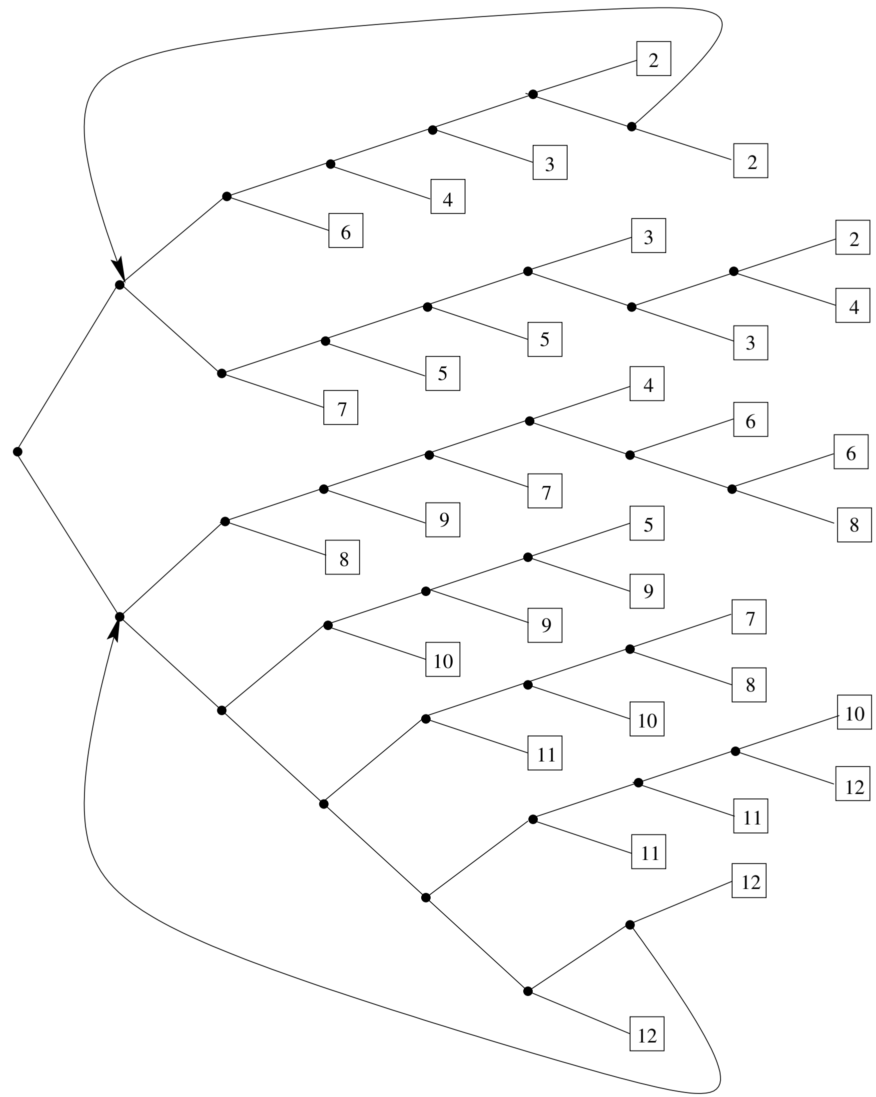
$$\text{else if } n = 4 \lor n = 10 \text{ then } \tfrac{3}{36}$$

$$\text{else if } n = 5 \lor n = 9 \text{ then } \tfrac{4}{36}$$

$$\text{else if } n = 6 \lor n = 8 \text{ then } \tfrac{5}{36}$$

$$\text{else if } n = 7 \text{ then } \tfrac{6}{36}$$

$$\text{else } 0$$



# 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:

$$\vdash \ \forall n, t, s.\ \mathsf{prime}\, n \ \Rightarrow\ \mathsf{fst}\,(\mathsf{miller}\, n\ t\ s) = \top$$

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

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 $\hat{\tau} = (\tau \to [0,1]) \to [0,1]$:

$$\begin{aligned}
\texttt{flip} : \hat{\mathbb{B}} &:= \lambda f : \mathbb{B} \to [0,1].\ f(\top)/2 + f(\bot)/2 \\
x +_p y : \hat{\tau} &:= \lambda f : \tau \to [0,1].\ p(x(f)) + (1-p)(y(f)) \\
\texttt{random}(n) : \hat{\mathbb{Z}} &:= \lambda f : \mathbb{Z} \to [0,1].\ \sum_{1 \leq i \leq n} f(i)/n
\end{aligned}$$

# Comparison: Coq Theorem Prover

Can model the Miller-Rabin test in Coq:

$$\texttt{witness}\ n\ a$$
$$:= \quad a^s \equiv 1 \pmod{n} \ \vee\ \exists j.\ 0 \le j < r \ \wedge\ a^{2^j s} \equiv -1 \pmod{n}$$

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

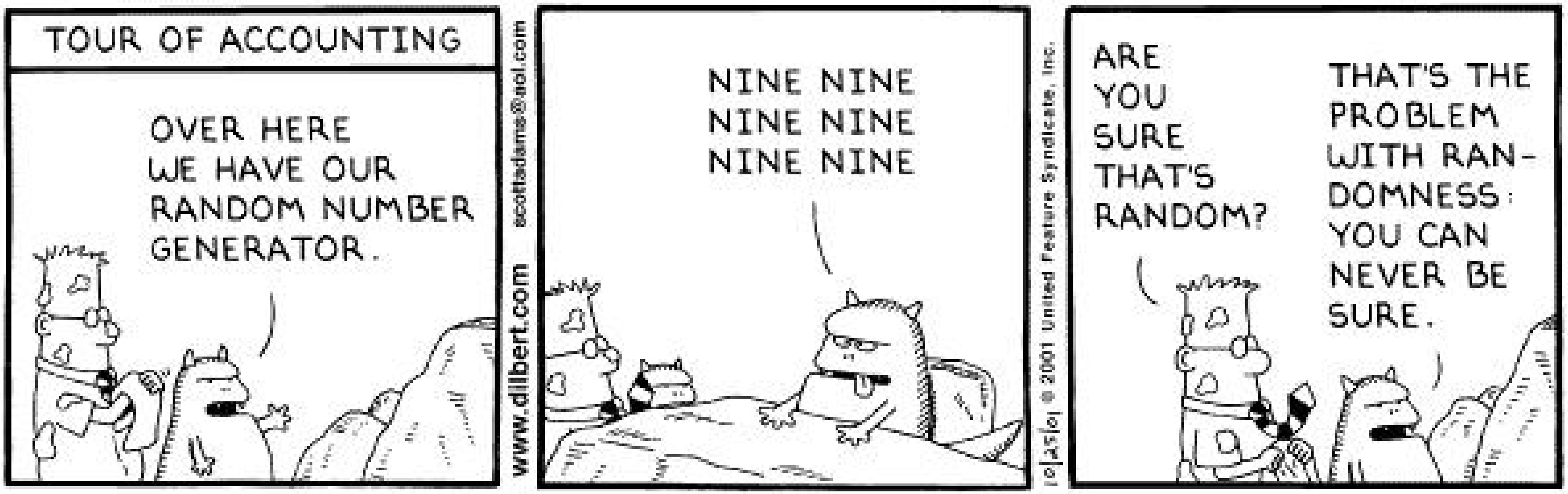
$$\texttt{miller}\ n\ t$$
$$:= \quad \texttt{if}\ n = 0\ \texttt{then}\ \texttt{unit}\ \top$$
$$\texttt{else}$$
$$\texttt{bind}\ (\texttt{bind}\ (\texttt{random}\ (n-1))\ (\lambda a.\ \texttt{unit}\ (\texttt{witness}\ n\ a)))$$
$$(\lambda b.\ \texttt{if}\ b\ \texttt{then}\ \texttt{miller}\ n\ (t-1)\ \texttt{else}\ \texttt{unit}\ \bot)$$

Meta-language evaluation of $\texttt{miller}\ 9\ 3$ 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/`