Verifying Probabilistic Programs using the HOL Theorem Prover

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

University of Cambridge
Contents

- Introduction
- Formalizing Probability
- Modelling Probabilistic Programs
- Example Verifications
- Conclusion
Introduction

- Quicksort Algorithm (Hoare, 1962):

```haskell
fun quicksort elements = 
  if length elements <= 1 then elements 
  else 
    let
      val pivot = choose_pivot elements 
      val (left, right) = partition pivot elements 
    in 
      quicksort left @ [pivot] @ quicksort right 
    end;
```

- Usually $O(n \log n)$ comparisons, unless choice of pivot interacts badly with data.
Introduction

- Example of bad behaviour when pivot is first element:

  input: [5, 4, 3, 2, 1]
  pivot 5: [4, 3, 2, 1]--5--[]
  pivot 4: [3, 2, 1]--4--[]
  pivot 3: [2, 1]--3--[]
  pivot 2: [1]--2--[]
  output: [1, 2, 3, 4, 5]

- Lists in reverse order take $O(n^2)$ comparisons.
- So do lists that are in the right order!
Introduction

- Solution: Introduce randomization into the algorithm itself.
- Pick pivots uniformly at random from the list of elements.
- Every list has exactly the same performance profile:
  - Expected number of comparisons is $O(n \log n)$.
  - Small class $C \subset S_n$ of lists with guaranteed bad performance has been replaced with a small probability $|C|/n!$ of bad performance on any input.
Introduction

- Broken procedure for choosing a pivot:

```haskell
fun choose_pivot elements = 
  if length elements = 1 orelse coin_flip ()
  then hd elements
  else choose_pivot (tl elements);
```

- Not a uniform distribution when length of elements > 2.

- Actually reinstates a bad class of input lists taking $O(n^2)$ (expected) comparisons.

- Would like to verify probabilistic programs in a theorem prover.
Motivation

---

Copyright © 2001 United Feature Syndicate, Inc.
Contents

- Introduction
- Formalizing Probability
- Modelling Probabilistic Programs
- Example Verifications
- Conclusion
The HOL Theorem Prover

- Developed by Mike Gordon’s Hardware Verification Group in Cambridge, first release was HOL88.
- Latest release in mid-2002 called HOL4, developed jointly by Cambridge and Utah.
- 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 program in HOL:

- Must be able to formalize its probabilistic specification:
  \[ \mathcal{E} : \mathcal{P}(\mathbb{B}^\infty), \quad \mathbb{P} : \mathcal{E} \to \mathbb{R} \]
- and model the probabilistic program in the logic:
  \[ \text{prob\_program} : \mathbb{N} \to \mathbb{B}^\infty \to \{\text{success, failure}\} \times \mathbb{B}^\infty \]
- then finally prove that the program satisfies its specification.

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

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

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

- 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 $P$:

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

$$P(S) = \text{the probability measure of } S \text{ (for } S \in \mathcal{E})$$
Formalizing Probability

- Formalized some general measure theory in HOL, including Carathéodory’s extension theorem.
- Next defined the measure of prefix sets (or cylinders):
  \[ \forall l. \mu \{ s_0s_1s_2 \cdots | [s_0, \ldots, s_{n-1}] = l \} = 2^{-\text{length } l} \]
- Finally extended this measure to a \( \sigma \)-algebra:
  \[ \mathcal{E} = \sigma(\text{prefix sets}) \]
  \[ \mathbb{P} = \text{Carathéodory extension of } \mu \text{ to } \mathcal{E} \]
- Similar to the definition of Lebesgue measure.
Contents

- Introduction
- Formalizing Probability
  - Modelling Probabilistic Programs
- Example Verifications
- Conclusion
Modelling Probabilistic Programs

• Given 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 Programs

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

  \[
  \text{unit } a \ = \ \lambda s. (a, s) \\
  \text{bind } f \ g \ = \ \lambda s. \ \text{let} \ (x, s') \leftarrow f(s) \text{ in } g x s' \\
  \text{coin_flip } f \ g \ = \ \lambda s. \ (\text{if shd } s \text{ then } f \text{ else } g, \ \text{stl } s)
  \]

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

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

  generates the sum of two dice.
Example: The Binomial$(n, \frac{1}{2})$ Distribution

- Definition of a sampling algorithm for the Binomial$(n, \frac{1}{2})$ distribution:

  \[ \vdash \text{bit} = \text{coin\_flip (unit 1)} (\text{unit 0}) \]
  \[ \vdash \text{binomial 0} = \text{unit 0} \land \forall n. \]
  \[ \quad \text{binomial (suc } n) = \]
  \[ \quad \text{bind bit } (\lambda x. \text{bind (binomial } n) (\lambda y. \text{unit } (x + y))) \]

- Correctness theorem:

  \[ \vdash \forall n, r. \mathbb{P} \{ s \mid \text{fst (binomial } n s) = r \} = \binom{n}{r} \left(\frac{1}{2}\right)^n \]
The Binomial(\(n, \frac{1}{2}\)) sampling algorithm is guaranteed to terminate within \(n\) coin-flips.

The following algorithm generates dice throws from coin-flips (Knuth and Yao, 1976):

- The backward loops introduce the possibility of looping forever.
- But the probability of this happening is 0.
- Probabilistic termination: the program terminates with probability 1.
Probabilistic Termination

- Probabilistic termination is more expressive than guaranteed termination.
- No coin-flip algorithm that is guaranteed to terminate can sample from the following distributions:
  - Uniform(3): choosing one of 0, 1, 2 each with probability $\frac{1}{3}$.
  - Geometric($\frac{1}{2}$): choosing $n \in \mathbb{N}$ with probability $(\frac{1}{2})^{n+1}$. *The index of the first head in a sequence of coin-flips.*
- We model probabilistic termination in HOL using a probabilistic while loop:

\[
\vdash \forall c, b, a. \quad \text{while } c \ b \ a = \text{if } c(a) \ \text{then bind } (b \ a) \ \text{(while } c \ b) \ \text{else unit } a
\]
Contents

- Introduction
- Formalizing Probability
- Modelling Probabilistic Programs
- Example Verifications
- Conclusion
Example: The \textbf{Uniform}(3) Distribution

- First make a raw definition of \texttt{unif3}:

  \begin{align*}
  \not\vdash \texttt{unif3} &= \\
  &\text{while } (\lambda n. \: n = 3) \\
  &\text{ (coin\_flip \texttt{(coin\_flip \texttt{(unit 0) (unit 1)}) (coin\_flip \texttt{(unit 2) (unit 3))}) } 3
  \end{align*}

- Next prove \texttt{unif3} satisfies probabilistic termination.
- Then independence must follow, and we can use this to derive a more elegant definition of \texttt{unif3}:

  \begin{align*}
  \not\vdash \texttt{unif3} &= \texttt{coin\_flip \texttt{(coin\_flip \texttt{(unit 0) (unit 1)}) (coin\_flip \texttt{(unit 2) unif3})}
  \end{align*}

- The correctness theorem also follows:

  \begin{align*}
  \not\vdash \forall n. \: \mathbb{P} \{ s \mid \text{fst \texttt{(unif3 s)}} = n \} &= \text{if } n < 3 \text{ then } \frac{1}{3} \text{ else } 0
  \end{align*
Example: Optimal Dice

A probabilistic finite state automaton:

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))))

Verifying Probabilistic Programs using the HOL Theorem Prover – Joe Hurd – p.21/32
Example: Optimal Dice

- Correctness theorem:

$$\forall n. \mathbb{P}\{s \mid \text{fst (dice } s\text{)} = n\} = \begin{cases} \text{if } 1 \leq n \land n \leq 6 \text{ then } \frac{1}{6} \text{ else } 0 \end{cases}$$

- The dice program takes $3\frac{2}{3}$ coin flips (on average) to output a dice throw.

- Knuth and Yao (1976) show this to be optimal.

- To generate the sum of two dice throws, is it possible to do better than $7\frac{1}{3}$ coin flips?
Example: Optimal Dice

On average, this program takes $4 \frac{7}{18}$ coin flips to produce a result, and this is also optimal.

\[
\begin{align*}
\forall n. \quad & P\{s \mid \text{fst (two_dice s)} = n\} = \\
& \text{if } n = 2 \lor n = 12 \text{ then } \frac{1}{36} \\
& \text{else if } n = 3 \lor n = 11 \text{ then } \frac{2}{36} \\
& \text{else if } n = 4 \lor n = 10 \text{ then } \frac{3}{36} \\
& \text{else if } n = 5 \lor n = 9 \text{ then } \frac{4}{36} \\
& \text{else if } n = 6 \lor n = 8 \text{ then } \frac{5}{36} \\
& \text{else if } n = 7 \text{ then } \frac{6}{36} \\
& \text{else } 0
\end{align*}
\]
Example: Random Walk

- A drunk exits a pub at point \( n \), and lurches left and right with equal probability until he hits home at point 0.

> HOME 0 1 \( i-1 \) \( i \) \( i+1 \) \( n \) \( \ldots \)

- Will the drunk always get home?
Example: Random Walk

- We can formalize the random walk as a probabilistic program:

  \[ \forall n. \text{lurch } n = \text{coin\_flip (unit } (n + 1)) (\text{unit } (n - 1)) \]
  \[ \forall f, b, a, k. \text{cost } f \ b \ (a, k) = \text{bind } (b(a)) (\lambda a'. \text{unit } (a', f(k))) \]
  \[ \forall n, k. \]

  \[
  \text{walk } n \ k = \\
  \text{bind } (\text{while } (\lambda (n, _). \ 0 < n) (\text{cost suc lurch}) (n, k)) \]

  \[
  (\lambda (_, k). \text{unit } k)
  \]

- “Will the drunk always get home?”
  is equivalent to
  “Does \text{walk} satisfy probabilistic termination?”
Example: Random Walk

- Perhaps surprisingly, the drunk **does** always get home.
- We formalize the proof of this in HOL
  - This shows the probabilistic termination of \textit{walk}.
  - And as usual, independence immediately follows.
- Then we can derive a more natural definition of \textit{walk}:
  \[
  \forall n, k. \\
  \text{walk } n \ k = \\
  \text{if } n = 0 \text{ then unit } k \text{ else} \\
  \text{coin\_flip } (\text{walk } (n+1) \ (k+1)) \ (\text{walk } (n-1) \ (k+1))
  \]
- And prove some neat properties:
  \[
  \forall n, k. \forall^* s. \text{even } (\text{fst } (\text{walk } n \ k \ s)) = \text{even } (n + k)
  \]
Example: Random Walk

- Can extract walk to ML and simulate it.
  - Use high-quality random bits from `/dev/random`.
- A typical sequence of results from random walks starting at level 1:
  
  
  57, 1, 7, 173, 5, 49, 1, 3, 1, 11, 9, 9, 1, 1, 1547, 27, 3, 1, 1, 1, ...

- Record breakers:
  - 34th simulation yields a walk with 2645 steps
  - 135th simulation yields a walk with 603787 steps
  - 664th simulation yields a walk with 1605511 steps
- Expected number of steps to get home is infinite!
Example: Miller-Rabin Primality Test

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

We formalize the test as a HOL function \texttt{miller}, and prove:

\[ \forall n, t, s. \text{prime } n \Rightarrow \text{fst (miller } n \ t \ s) = \top \]
\[ \forall n, t. \neg \text{prime } n \Rightarrow 1 - 2^{-t} \leq \Pr \{ s \mid \text{fst (miller } n \ t \ s) = \bot \} \]

Here \( n \) is the number to test for primality, and \( t \) is the maximum number of iterations allowed.
Example: Miller-Rabin Primality Test

- Can define a pseudo-random number generator in HOL, and interpret \texttt{miller} in the logic to prove numbers composite:

  \[ \neg \text{prime}(2^{26} + 1) \land \neg \text{prime}(2^{27} + 1) \land \neg \text{prime}(2^{28} + 1) \]

- Or can manually extract \texttt{miller} to ML, and execute it using \texttt{/dev/random} and calls to GMP:

<table>
<thead>
<tr>
<th>bits</th>
<th>$E_{l,n}$</th>
<th>MR</th>
<th>Gen time</th>
<th>MR$_1$ time</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>99424</td>
<td>99458</td>
<td>0.0443</td>
<td>0.2498</td>
</tr>
<tr>
<td>1000</td>
<td>99712</td>
<td>99716</td>
<td>0.0881</td>
<td>0.7284</td>
</tr>
<tr>
<td>2000</td>
<td>99856</td>
<td>99852</td>
<td>0.3999</td>
<td>4.2910</td>
</tr>
</tbody>
</table>
Contents

- Introduction
- Formalizing Probability
- Modelling Probabilistic Programs
- Example Verifications
- Conclusion
Conclusion

- Feasible to verify probabilistic programs in a theorem prover, ‘just like deterministic programs.’
- Requires much interactive proof to verify each algorithm, with heavy use of automatic proof tools.
- ... but once verified, probabilistic programs can then be used as building blocks in higher-level ones.
- Fixing on coin-flips creates a distinction between guaranteed termination and probabilistic termination.
- Aim for a library of verified probabilistic programs, with ML extractions available.
- Also need more theory: randomized quicksort (and many others) will require expectation.
Related Work

- Probabilistic model checking, Kwiatkowska, Norman, Segala and Sproston, 2000.
- Probabilistic predicate transformers, Morgan, McIver, Seidel and Sanders, 1994–
  - *Proof Rules for Probabilistic Loops*, Morgan, 1996