A Formal Approach to Probabilistic Termination

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

University of Cambridge

Contents

- **Introduction**
- Modelling Probabilistic Programs
- Probabilistic While Loop
- Random Walk
- Conclusion

Introduction

- Quicksort Algorithm (Hoare, 1962):

```
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

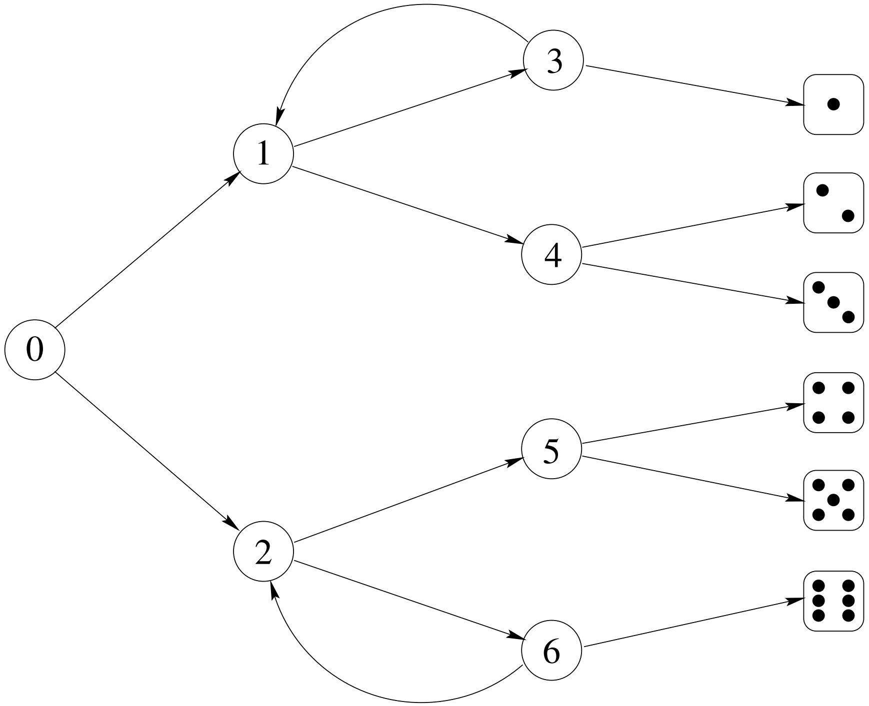
- Broken procedure for choosing a pivot:

```
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.

Introduction

- The (broken) `choose_pivot` program is guaranteed to terminate within $length(\texttt{elements})$ 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.

Introduction

- 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.*
- But how can probabilistic termination be modelled in a logic of total functions?
  - *What should* Geometric$(\frac{1}{2})$ *return for the all-tails sequence?*

Contents

- Introduction
- **Modelling Probabilistic Programs**
- Probabilistic While Loop
- Random Walk
- 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.

Verification in HOL

To verify a probabilistic program 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 program 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 program satisfies its specification.

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

Modelling Probabilistic Programs

- Given 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 Programs

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

$$\begin{aligned} \mathsf{unit}\ a &= \lambda\, s.\ (a, s) \\ \mathsf{bind}\ f\ g &= \lambda\, s.\ \mathsf{let}\ (x, s') \leftarrow f(s)\ \mathsf{in}\ g\ x\ s' \\ \mathsf{coin\_flip}\ f\ g &= \lambda\, s.\ (\mathsf{if\ shd}\ s\ \mathsf{then}\ f\ \mathsf{else}\ g,\ \mathsf{stl}\ s) \end{aligned}$$

- For example, if 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$(n, \frac{1}{2})$ Distribution

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

$$
\begin{array}{ll}
\vdash & \text{bit} = \text{coin\_flip}\ (\text{unit}\ 1)\ (\text{unit}\ 0) \\
\vdash & \text{binomial}\ 0 = \text{unit}\ 0\ \wedge \\
& \forall\, n. \\
& \quad \text{binomial}\ (\text{suc}\ n) = \\
& \quad \text{bind bit}\ (\lambda\, x.\ \text{bind}\ (\text{binomial}\ n)\ (\lambda\, y.\ \text{unit}\ (x + y)))
\end{array}
$$

- Correctness theorem:

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

Contents

- Introduction
- Modelling Probabilistic Programs
- **Probabilistic While Loop**
- Random Walk
- Conclusion

Probabilistic While Loop

- Consider the following *bounded* probabilistic while loop:

$$
\begin{aligned}
\vdash\ & \forall\, c, b, n, a. \\
& \quad \mathsf{while\_cut}\ c\ b\ 0\ a = \mathsf{unit}\ a\ \ \wedge \\
& \quad \mathsf{while\_cut}\ c\ b\ (\mathsf{suc}\ n)\ a = \\
& \quad \mathsf{if}\ c(a)\ \mathsf{then\ bind}\ (b(a))\ (\mathsf{while\_cut}\ c\ b\ n)\ \mathsf{else\ unit}\ a
\end{aligned}
$$

  - $a : \alpha$ is the loop state.
  - $c : \alpha \to \mathbb{B}$ is the loop condition.
  - $b : \alpha \to \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ is the loop body.
  - $n : \mathbb{N}$ is a cut-off parameter, ensuring that the loop always terminates within n iterations.
- The bounded while loop is guaranteed to terminate.

Probabilistic While Loop

- We can now define an *unbounded* probabilistic while loop as follows:

$$
\begin{array}{ll}
\vdash & \forall\, c, b, a, s.\\
& \quad \mathsf{while}\ c\ b\ a\ s =\\
& \quad \mathsf{if}\ \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{while\_cut}\ c\ b\ n\ a\ s))\ \mathsf{then}\\
& \quad\quad \mathsf{while\_cut}\ c\ b\\
& \quad\quad (\mathsf{minimal}\ (\lambda\, n.\ \neg c(\mathsf{fst}\ (\mathsf{while\_cut}\ c\ b\ n\ a\ s))))\ a\ s\\
& \quad \mathsf{else\ arb}
\end{array}
$$

- For a given starting state (c, b, a, s):
  - if the loop would naturally terminate after n iterations then it does so;
  - otherwise it returns the arbitrary value arb.

Probabilistic While Loop

- We can advance the probabilistic while loop:

$$\begin{aligned} \vdash\ & \forall\, c, b, a. \\ & \quad \mathsf{while}\ c\ b\ a = \\ & \quad \mathsf{if}\ c(a)\ \mathsf{then}\ \mathsf{bind}\ (b(a))\ (\mathsf{while}\ c\ b)\ \mathsf{else}\ \mathsf{unit}\ a \end{aligned}$$

- For a desirable independence property to hold, the following must be true of c and b:

$$\forall\, a.\ \forall^* s.\ \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{while\_cut}\ c\ b\ n\ a\ s))$$

  - $\forall^* s.\ \phi(s)$ means $\{s \mid \phi(s)\} \in \mathcal{E}\ \wedge\ \mathbb{P}\,\{s \mid \phi(s)\} = 1$.

- Can see this as a probabilistic termination condition.
  - Equivalent to the 0-1 law of Hart, Sharir and Pnueli.

Example: The $\mathsf{Uniform}(3)$ Distribution

- First make a raw definition of unif3:

$$\vdash \quad \mathsf{unif3} = \\ \mathsf{while}\ (\lambda\, n.\ n = 3) \\ (\mathsf{coin\_flip}\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 0)\ (\mathsf{unit}\ 1))\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 2)\ (\mathsf{unit}\ 3)))\ 3$$

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

$$\vdash \quad \mathsf{unif3} = \mathsf{coin\_flip}\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 0)\ (\mathsf{unit}\ 1))\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 2)\ \mathsf{unif3})$$

- The correctness theorem also follows:

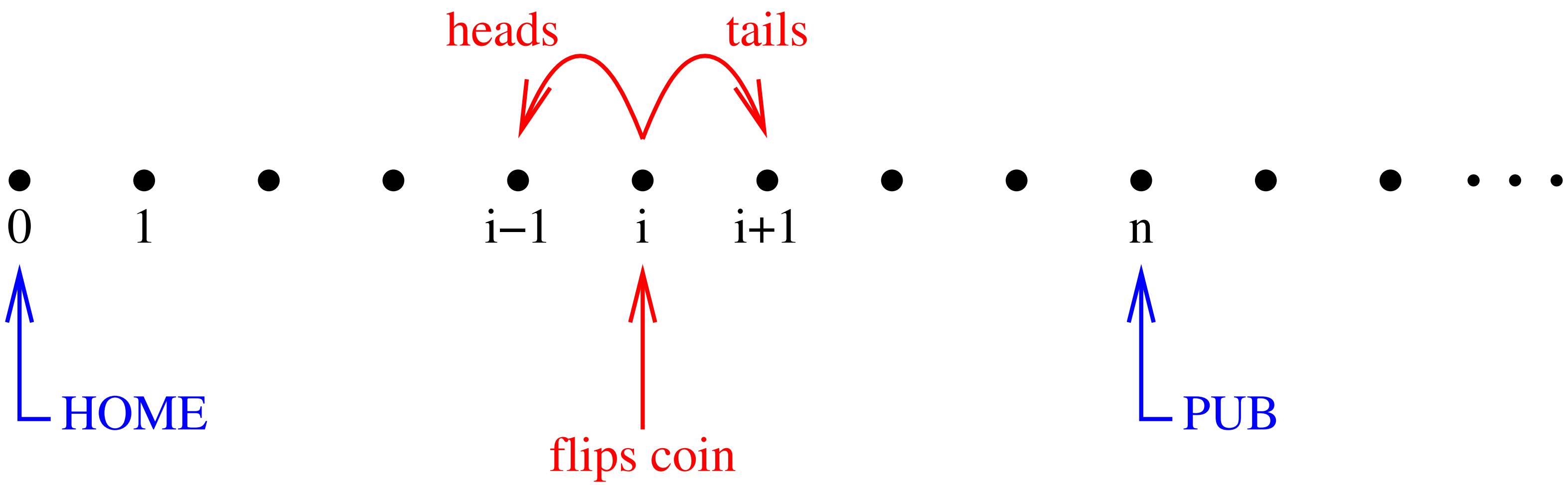
$$\vdash \quad \forall\, n.\ \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{unif3}\ s) = n\} = \mathsf{if}\ n < 3\ \mathsf{then}\ \tfrac{1}{3}\ \mathsf{else}\ 0$$

Contents

- Introduction
- Modelling Probabilistic Programs
- Probabilistic While Loop
- **Random Walk**
- Conclusion

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.



- Will the drunk always get home?

Random Walk

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

$$\vdash \quad \forall\, n.\ \mathsf{lurch}\ n = \mathsf{coin\_flip}\ (\mathsf{unit}\ (n+1))\ (\mathsf{unit}\ (n-1))$$

$$\vdash \quad \forall\, f, b, a, k.\ \mathsf{cost}\ f\ b\ (a, k) = \mathsf{bind}\ (b(a))\ (\lambda\, a'.\ \mathsf{unit}\ (a', f(k)))$$

$$\vdash \quad \forall\, n, k.$$

$$\mathsf{walk}\ n\ k =$$

$$\mathsf{bind}\ (\mathsf{while}\ (\lambda\, (n, \_).\ 0 < n)\ (\mathsf{cost\ suc\ lurch})\ (n, k))$$

$$(\lambda\, (\_, k).\ \mathsf{unit}\ k)$$

- *"Will the drunk always get home?"*
  is equivalent to
  "Does walk *satisfy probabilistic termination?"*

Random Walk

- Perhaps surprisingly, the drunk does always get home.
- To see this, let π_{ij} be the probability that a drunk starting at point i will eventually hit point j.
- The first property of π_{ij} that we prove is

  Translation Invariance: $\vdash \; \forall\, i, j, n.\; \pi_{ij} = \pi_{(i+n)(j+n)}$
- This is used to prove the all-important

  Multiplicative Property: $\vdash \; \forall\, i.\; \pi_{i0} = \pi_{10}^{i}$
- So if $\pi_{10} = 1$, then probabilistic termination is assured: the drunk gets home from every pub.

Random Walk

- By the definition of the random walk, we have:

$$\pi_{10} = \tfrac{1}{2}\pi_{20} + \tfrac{1}{2}\pi_{00}$$

- Applying the Multiplicative Property again:

$$\pi_{10} = \tfrac{1}{2}\pi_{10}^{2} + \tfrac{1}{2}$$

- And this can be rearranged to

$$(\pi_{10} - 1)^2 = 0$$

- The only solution of this equation is:

$$\pi_{10} = 1$$

Random Walk

- As usual, independence is a consequence of probabilistic termination.
- This allows us to derive a more natural definition:

$$
\begin{array}{l}
\vdash \quad \forall\, n, k. \\
\qquad \mathsf{walk}\ n\ k = \\
\qquad \mathsf{if}\ n = 0\ \mathsf{then\ unit}\ k\ \mathsf{else} \\
\qquad\quad \mathsf{coin\_flip}\ (\mathsf{walk}\ (n{+}1)\ (k{+}1))\ (\mathsf{walk}\ (n{-}1)\ (k{+}1))
\end{array}
$$

- And prove some neat properties:

$$\vdash \quad \forall\, n, k.\ \forall^* s.\ \mathsf{even}\ (\mathsf{fst}\ (\mathsf{walk}\ n\ k\ s)) = \mathsf{even}\ (n + k)$$

Random Walk

- Can also 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:

$$57, 1, 7, 173, 5, 49, 1, 3, 1, 11, 9, 9, 1, 1, 1547, 27, 3, 1, 1, 1, \ldots$$

- 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!

Contents

- Introduction
- Modelling Probabilistic Programs
- Probabilistic While Loop
- Random Walk
- **Conclusion**

Conclusion

- Fixing on coin-flips creates a distinction between guaranteed termination and probabilistic termination.
    - Functions that are guaranteed to terminate have better logical properties, and can bound the number of random bits that they will require.
    - But many interesting algorithms require probabilistic termination to be defined.
- Could define some program schemes to help prove probabilistic termination.
    - But there will always be programs such as the random walk that don’t fit into any scheme because their termination argument is too subtle.

Future Work

- Directly support recursive definitions of probabilistic programs (TFL-like behaviour):
  - User inputs intended recursion equations.
  - System makes a definition.
  - Sytem derives the recursive equations and induction theorem, with probabilistic termination condition as an assumption.
  - User proves this condition (perhaps using auxiliary function).

Related Work

- *Semantics of Probabilistic Programs*, Kozen, 1979.
- *Termination of Probabilistic Concurrent Processes*, Hart, Sharir and Pnueli, 1983.
- Probabilistic predicate transformers, Morgan, McIver, Seidel and Sanders, 1994–
    - *Notes on the Random Walk: an Example of Probabilistic Temporal Reasoning*, 1996
    - *Proof Rules for Probabilistic Loops*, Morgan, 1996