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):

```
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.

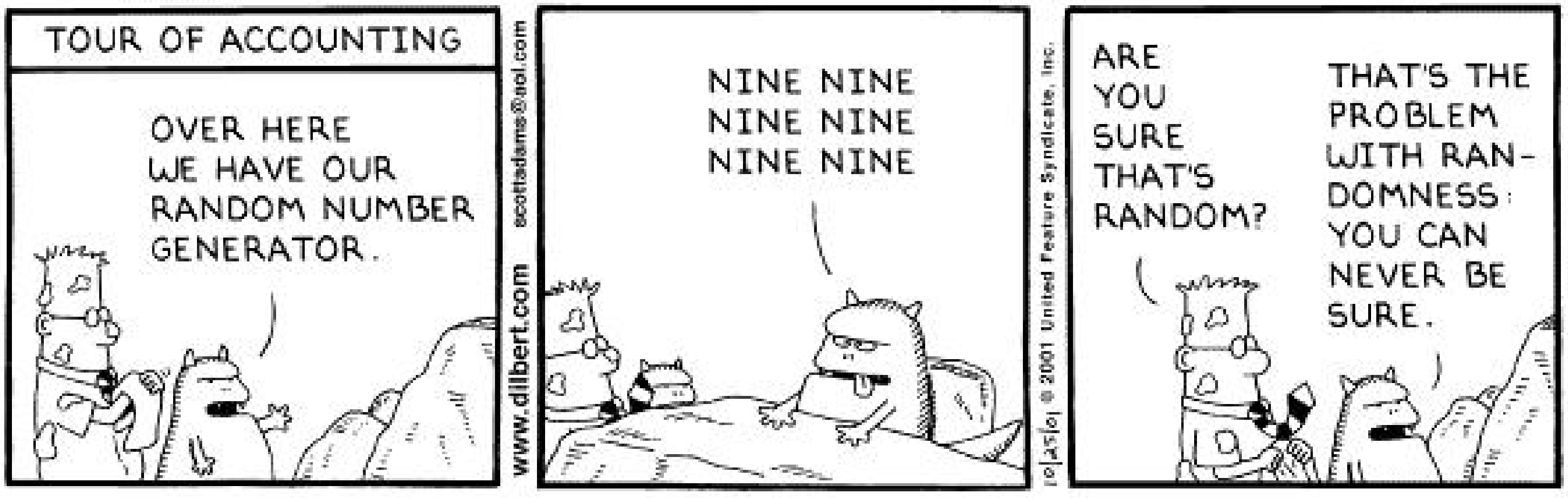
Introduction

- 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.

Motivation


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
10/25/01 © 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

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.

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.

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

Formalizing Probability

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

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

- 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 $\mathbb{P}$:

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

Formalizing Probability

- Formalized some general measure theory in HOL, including Carathéodory's extension theorem.
- Next defined the measure of prefix sets (or cylinders):

$$\forall l.\ \mu \{s_0 s_1 s_2 \cdots \mid [s_0, \ldots, s_{n-1}] = l\} = 2^{-(\mathsf{length}\ l)}$$

- Finally extended this measure to a σ-algebra:

$$\begin{array}{rcl} \mathcal{E} & = & \sigma(\text{prefix sets}) \\ \mathbb{P} & = & \text{Carathéodory extension of } \mu \text{ to } \mathcal{E} \end{array}$$

- Similar to the definition of Lebesgue measure.

Contents

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

Modelling Probabilistic Programs

- Given a probabilistic ‘function’:

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

- 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:

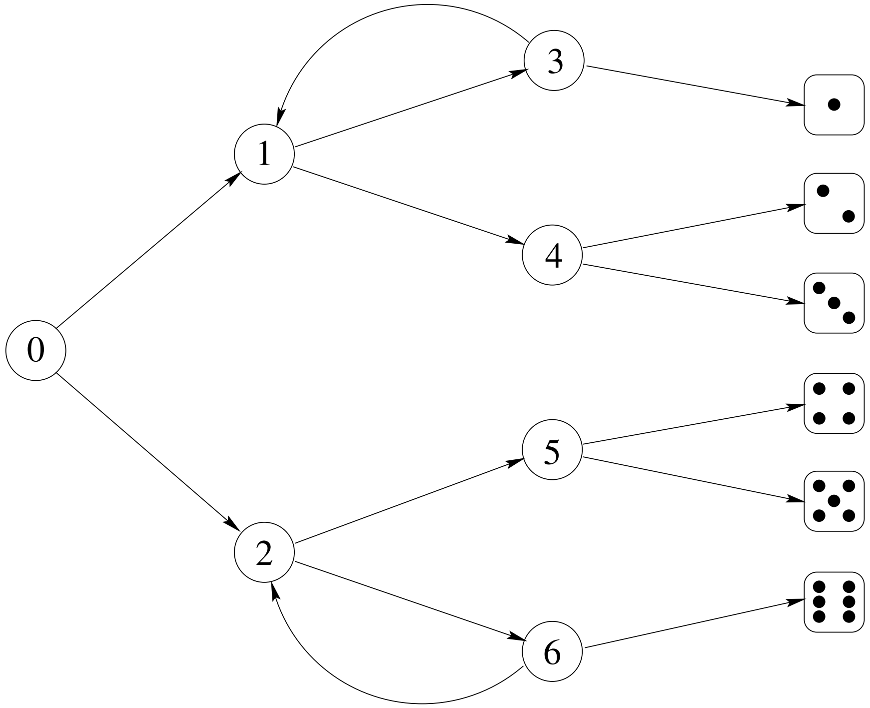
$$
\begin{array}{ll}
\vdash & \mathsf{bit} = \mathsf{coin\_flip}\ (\mathsf{unit}\ 1)\ (\mathsf{unit}\ 0) \\
\vdash & \mathsf{binomial}\ 0 = \mathsf{unit}\ 0\ \wedge \\
 & \forall\, n. \\
 & \quad \mathsf{binomial}\ (\mathsf{suc}\ n) = \\
 & \quad \mathsf{bind}\ \mathsf{bit}\ (\lambda\, x.\ \mathsf{bind}\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

- Correctness theorem:

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

Probabilistic Termination

- 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:

$$\begin{array}{ll} \vdash & \forall\, c, b, a. \\ & \text{while } c\ b\ a = \text{if } c(a) \text{ then bind } (b\ a)\ (\text{while } c\ b) \text{ else unit } a \end{array}$$

Contents

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

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

- First make a raw definition of unif3:

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

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

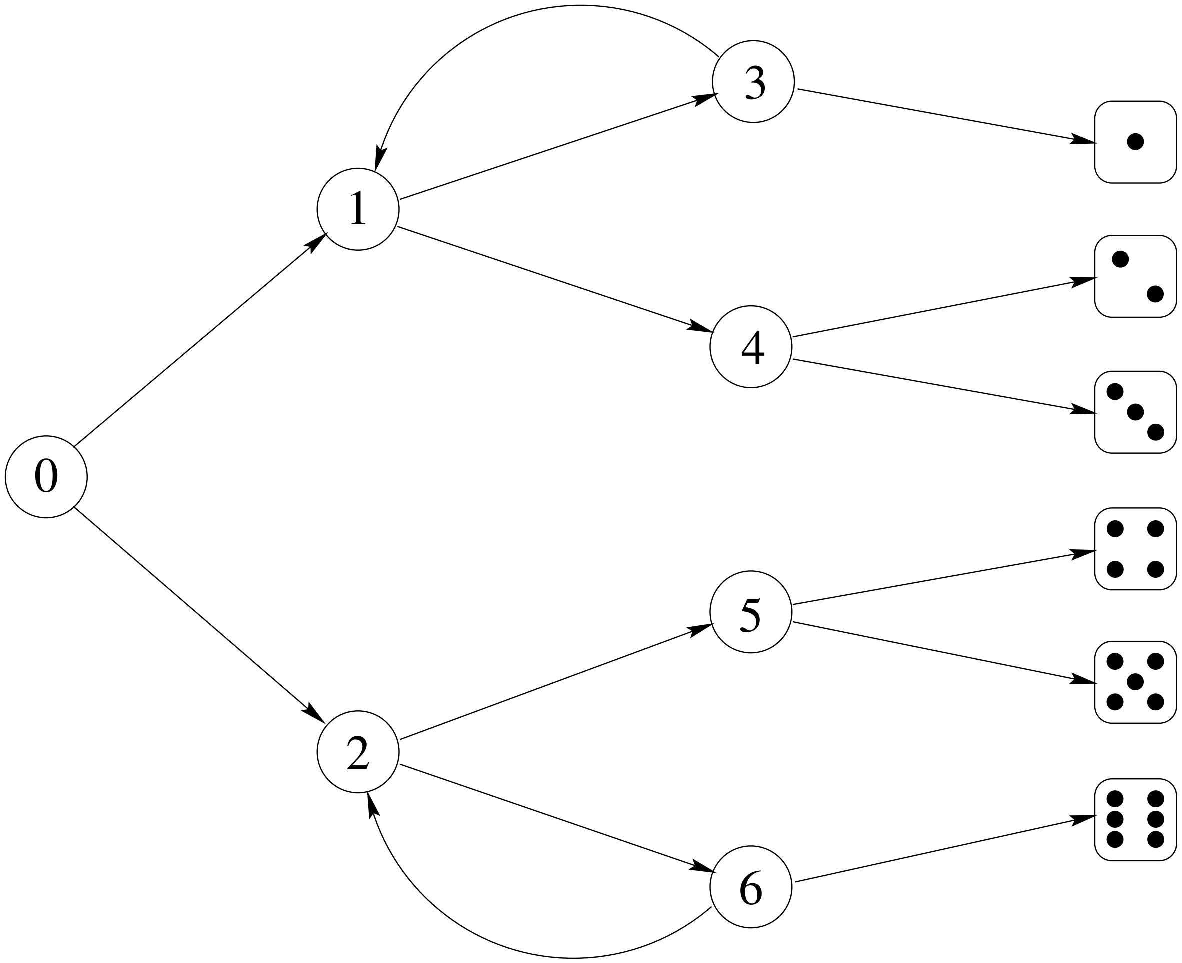
$$\vdash \quad \mathsf{unif3} = \mathsf{coin\_flip}\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 0)\ (\mathsf{unit}\ 1))\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 2)\ \mathsf{unif3})$$

- The correctness theorem also follows:

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

Example: Optimal Dice

A probabilistic finite state automaton:



```
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))))
```

Example: Optimal Dice

- Correctness theorem:

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

- 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.

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

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

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

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

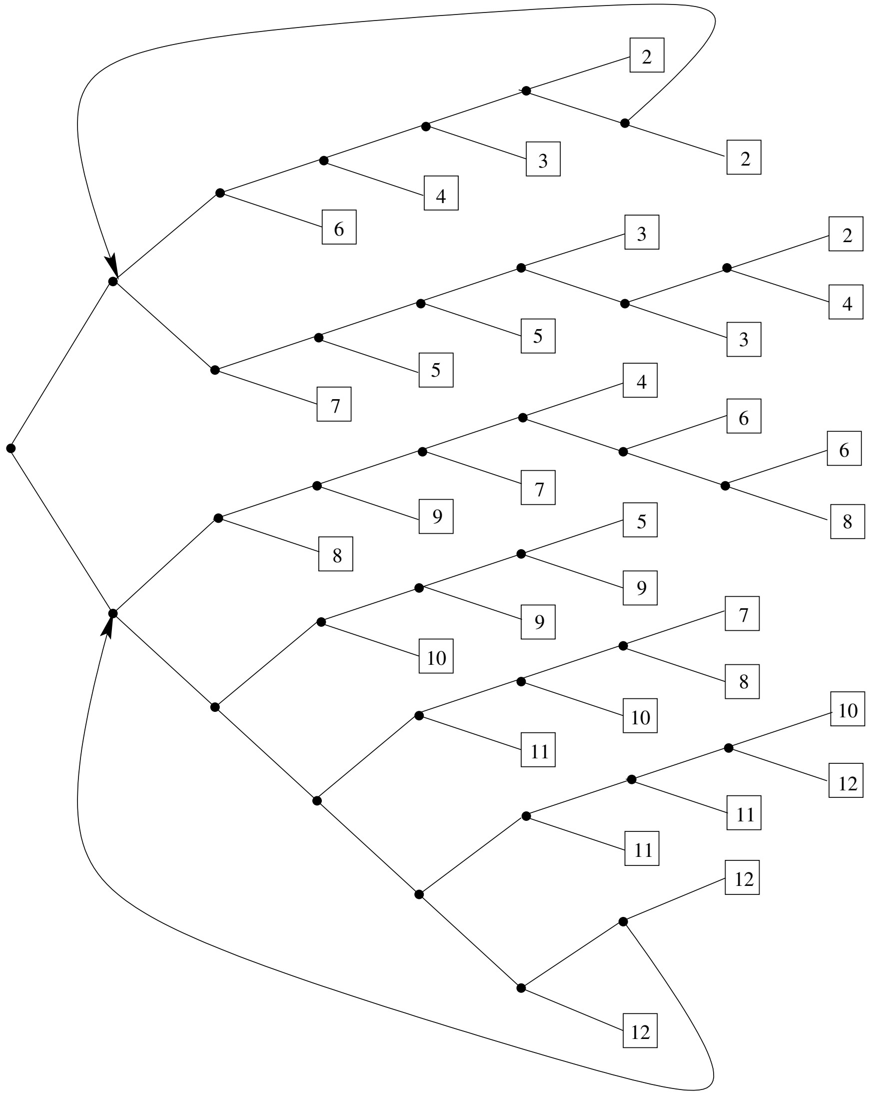
$$\text{else if } n = 4 \lor n = 10 \text{ then } \tfrac{3}{36}$$

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

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

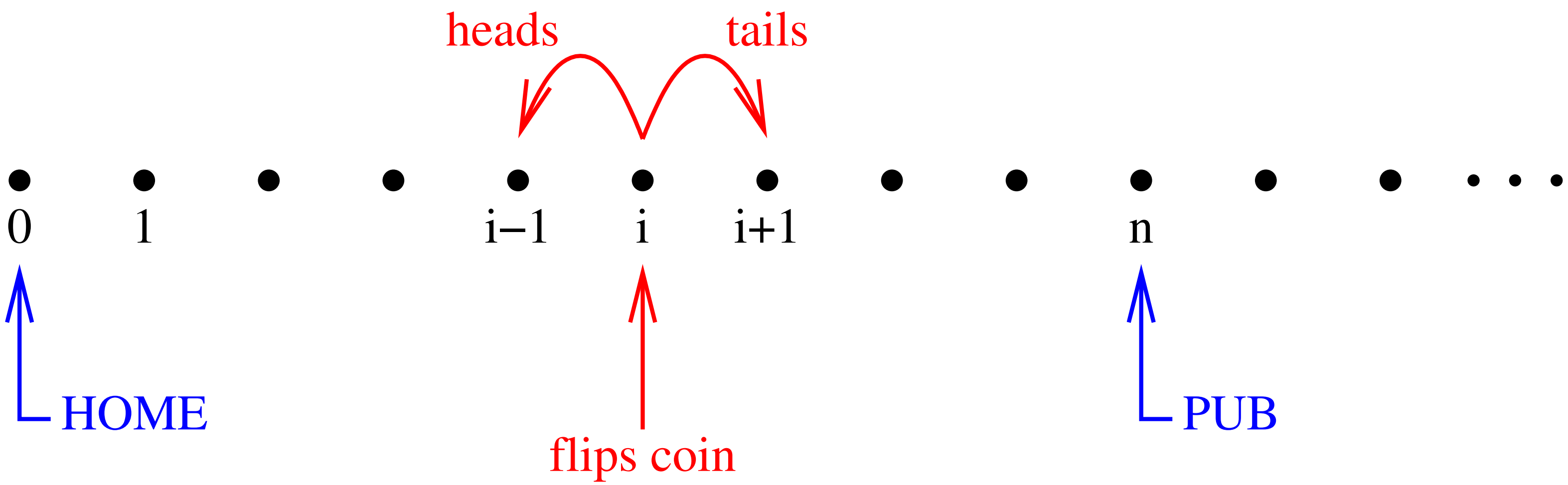
$$\text{else if } n = 7 \text{ then } \tfrac{6}{36}$$

$$\text{else } 0$$



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.



- Will the drunk always get home?

Example: Random Walk

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

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

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

$$\begin{aligned} \vdash \quad & \forall\, n, k. \\ & \quad \mathsf{walk}\ n\ k = \\ & \quad \mathsf{bind}\ (\mathsf{while}\ (\lambda\, (n, \_).\ 0 < n)\ (\mathsf{cost}\ \mathsf{suc}\ \mathsf{lurch})\ (n, k)) \\ & \quad (\lambda\, (\_, k).\ \mathsf{unit}\ k) \end{aligned}$$

- *“Will the drunk always get home?”*
  is equivalent to
  “Does 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 walk.
  - And as usual, independence immediately follows.
- Then we can derive a more natural definition of walk:

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

- And prove some neat properties:

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

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:

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

- 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!

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 miller, and prove:

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

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

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

Example: Miller-Rabin Primality Test

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

$$\vdash \neg \mathsf{prime}(2^{2^6}+1) \wedge \neg \mathsf{prime}(2^{2^7}+1) \wedge \neg \mathsf{prime}(2^{2^8}+1)$$

- Or can manually extract miller to ML, and execute it using `/dev/random` and calls to GMP:

bits	$\mathbb{E}_{l,n}$	MR	Gen time	MR$_1$ time
500	99424	99458	0.0443	0.2498
1000	99712	99716	0.0881	0.7284
2000	99856	99852	0.3999	4.2910

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

- *Semantics of Probabilistic Programs*, Kozen, 1979.
- Probabilistic model checking, Kwiatkowska, Norman, Segala and Sproston, 2000.
- *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