Formal Verification of Probabilistic Programs: Two Approaches

Joe Hurd

joe.hurd@cl.cam.ac.uk

University of Cambridge

Joint work with Annabelle McIver (Macquarie University) and Carroll Morgan (University of New South Wales)

Contents

Introduction

Probabilistic programs are useful for many applications:

- Symmetry breaking
  - Rabin's mutual exclusion algorithm
- Eliminating pathological cases
  - Miller-Rabin primality test
- Algorithm complexity
  - Sorting nuts and bolts
- Defeating a powerful adversary
  - Mixed strategies in game theory
- Solving a problem in an extremely simple way
  - Finding minimal cuts

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.

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, Utah and ANU.
- 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.

Contents

- Introduction
- **Approach 1: Monads**
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - wlp Verification Condition Generator
  - Example Verifications
- Conclusion

Introduction: Monads

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.

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

Contents

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 $\mathbb{P}$:

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

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
- Approach 1: Monads
  - Formalizing Probability
  - **Modelling Probabilistic Programs**
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - wlp Verification Condition Generator
  - 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 $\text{Binomial}(n, \frac{1}{2})$ Distribution

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

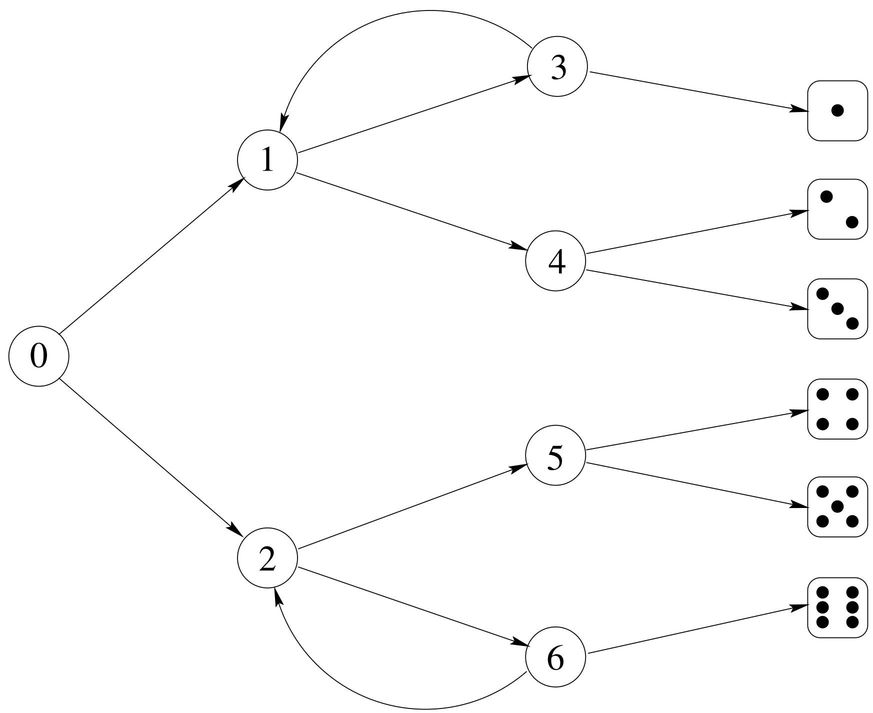
$$
\begin{array}{ll}
\vdash & \text{bit} = \text{coin\_flip}\ (\text{unit}\ 1)\ (\text{unit}\ 0) \\
\vdash & \text{binomial}\ 0 = \text{unit}\ 0\ \ \wedge \\
& \forall\, n.\\
& \quad \text{binomial}\ (\text{suc}\ n) = \\
& \quad \text{bind bit}\ (\lambda\, x.\ \text{bind}\ (\text{binomial}\ n)\ (\lambda\,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

- Correctness theorem:

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

Probabilistic Termination

- The $\mathsf{Binomial}(n, \frac{1}{2})$ 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:

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

Contents

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.
- This allows us to derive a recursive definition of unif3:

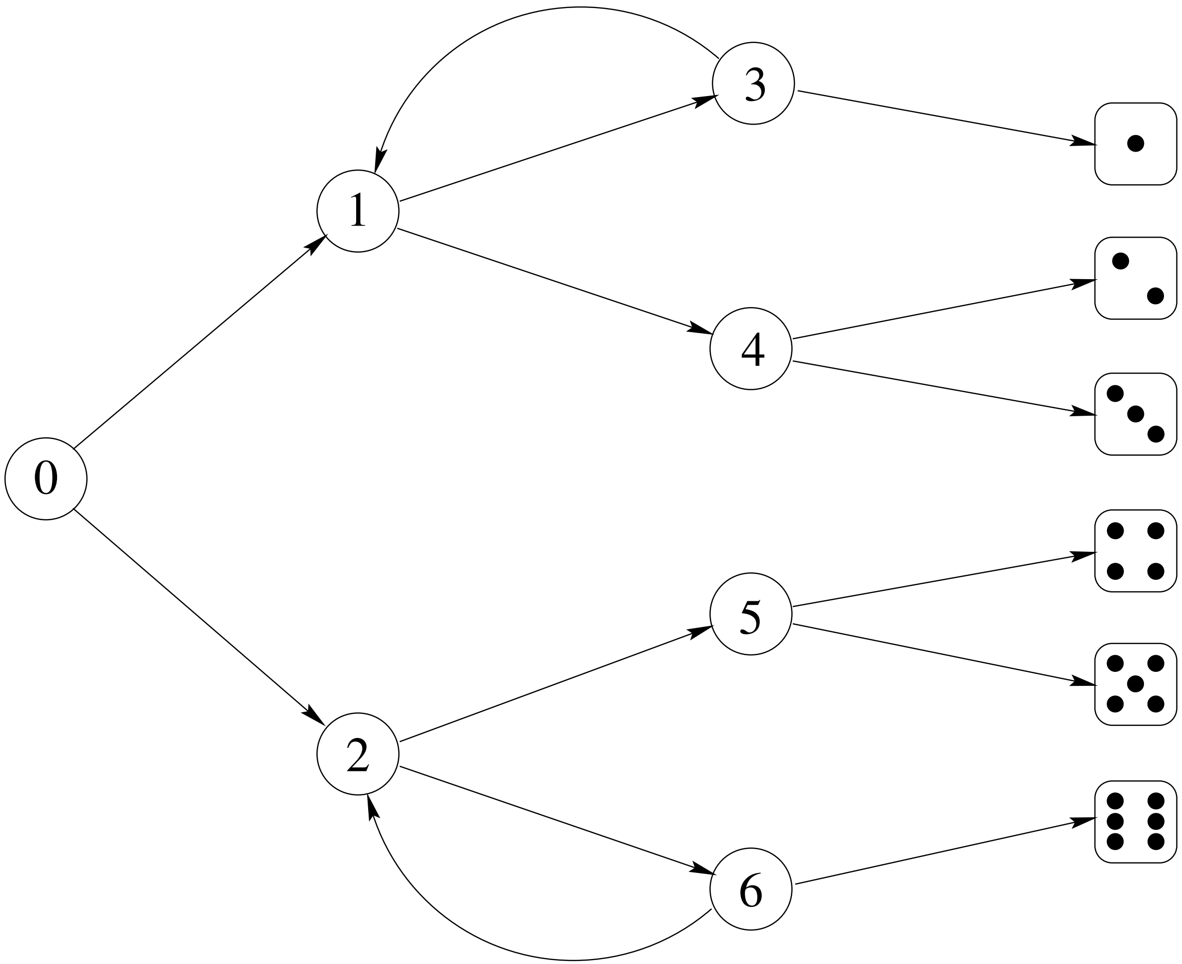
$$\vdash \quad \mathsf{unif3} = \mathsf{coin\_flip}\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 0)\ (\mathsf{unit}\ 1))\ (\mathsf{coin\_flip}\ (\mathsf{unit}\ 2)\ \mathsf{unif3})$$

- 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:

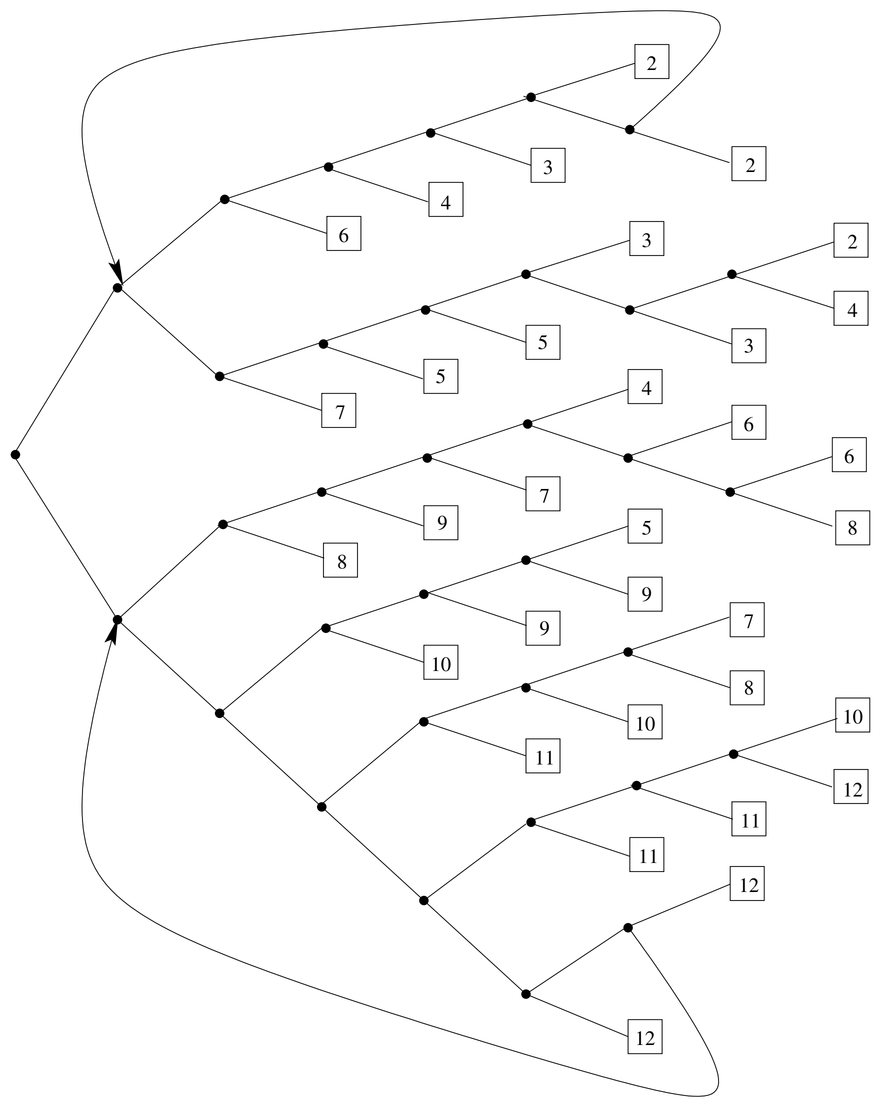
$$\vdash \quad \forall n.\, \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{dice}\ s) = n\} = \mathsf{if}\ 1 \le n \land n \le 6\ \mathsf{then}\ \tfrac{1}{6}\ \mathsf{else}\ 0$$

- 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

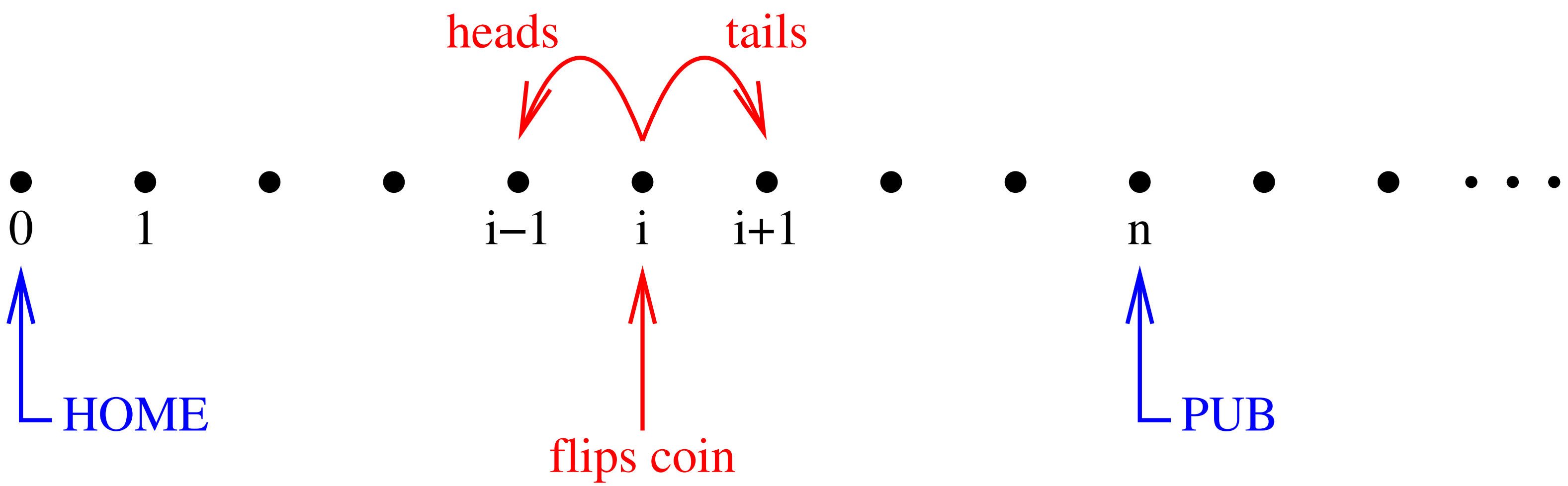
On average, this program takes $4\frac{7}{18}$ coin flips to produce a result, and this is also optimal.

$$\begin{array}{ll}\vdash & \forall n.\\ & \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{two\_dice}\ s) = n\} = \\ & \mathsf{if}\ n = 2 \lor n = 12\ \mathsf{then}\ \frac{1}{36} \\ & \mathsf{else\ if}\ n = 3 \lor n = 11\ \mathsf{then}\ \frac{2}{36} \\ & \mathsf{else\ if}\ n = 4 \lor n = 10\ \mathsf{then}\ \frac{3}{36} \\ & \mathsf{else\ if}\ n = 5 \lor n = 9\ \mathsf{then}\ \frac{4}{36} \\ & \mathsf{else\ if}\ n = 6 \lor n = 8\ \mathsf{then}\ \frac{5}{36} \\ & \mathsf{else\ if}\ n = 7\ \mathsf{then}\ \frac{6}{36} \\ & \mathsf{else}\ 0\end{array}$$



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

- Perhaps surprisingly, the drunk does always get home.
  - We formalize the proof of this in HOL.
  - Thus the formalized random walk satisfies probabilistic termination.
- This allows us to derive a natural definition of walk:

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

- 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
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- **Approach 2: pGCL**
  - Formalizing Probabilistic Guarded Commands
  - wlp Verification Condition Generator
  - Example Verifications
- Conclusion

Introduction: pGCL

- pGCL stands for probabilistic Guarded Command Language.
- It's Dijkstra's GCL extended with probabilistic choice

$$c_1 \;{}_p\!\oplus\; c_2$$

- Like GCL, the semantics is based on weakest preconditions.
- Important: retains demonic choice

$$c_1 \sqcap c_2$$

- Developed by Morgan et al. in the Programming Research Group, Oxford, 1994–

Contents

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - **Formalizing Probabilistic Guarded Commands**
  - wlp Verification Condition Generator
  - Example Verifications
- Conclusion

pGCL Semantics

- Given a standard program C and a postcondition Q, let P be the weakest precondition that satisfies

$$[P]C[Q]$$

- Precondition P is weaker than P' if $P' \Rightarrow P$.
- Such a P will always exist and be unique, so think of C as a function that transforms postconditions into weakest preconditions.
- pGCL generalizes this to probabilistic programs:
    - Conditions $\alpha \rightarrow \mathbb{B}$ become expectations $\alpha \rightarrow \mathsf{posreal}$.
    - Expectation P is weaker than P' if $P' \sqsubseteq P$.
    - Think of programs as *expectation transformers*.

pGCL Commands

Model pGCL commands with a HOL datatype:

$$
\begin{array}{lcl}
\mathsf{command} & \equiv & \mathsf{Assert\ of\ (state \to posreal) \times command} \\
& | & \mathsf{Abort} \\
& | & \mathsf{Skip} \\
& | & \mathsf{Assign\ of\ string \times (state \to \mathbb{Z})} \\
& | & \mathsf{Seq\ of\ command \times command} \\
& | & \mathsf{Demon\ of\ command \times command} \\
& | & \mathsf{Prob\ of\ (state \to posreal) \times command \times command} \\
& | & \mathsf{While\ of\ (state \to \mathbb{B}) \times command}
\end{array}
$$

Note: the probability in Prob can depend on the state.

Derived Commands

Define the following *derived commands* as syntactic sugar:

$$
\begin{array}{rcl}
v := e & \equiv & \mathsf{Assign}\ v\ e \\
c_1\ ;\ c_2 & \equiv & \mathsf{Seq}\ c_1\ c_2 \\
c_1\ \sqcap\ c_2 & \equiv & \mathsf{Demon}\ c_1\ c_2 \\
c_1\ {}_p\!\oplus\ c_2 & \equiv & \mathsf{Prob}\ (\lambda s.\ p)\ c_1\ c_2 \\
\mathsf{Cond}\ b\ c_1\ c_2 & \equiv & \mathsf{Prob}\ (\lambda s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ c_1\ c_2 \\
v := \{e_1, \ldots, e_n\} & \equiv & v := e_1\ \sqcap\ \cdots\ \sqcap\ v := e_n \\
v := \langle e_1, \cdots, e_n \rangle & \equiv & v := e_1\ {}_{1/n}\!\oplus\ v := \langle e_2, \ldots, e_n \rangle \\
p_1 \to c_1 \mid \cdots \mid p_n \to c_n & \equiv &
\end{array}
$$

$$
\begin{cases}
\mathsf{Abort} & \text{if none of the } p_i \text{ hold on the current state} \\
\prod_{i \in I} c_i & \text{where } I = \{i \mid 1 \le i \le n \wedge p_i \text{ holds}\}
\end{cases}
$$

In addition, we write $v := n + 1$ instead of "v" $:= \lambda s.\ s$ "n" $+ 1$.

Weakest Preconditions

Define weakest preconditions (wp) directly on commands:

$$
\begin{aligned}
\vdash \quad & (\mathsf{wp}\ (\mathsf{Assert}\ p\ c) = \mathsf{wp}\ c) \\
& \wedge (\mathsf{wp}\ \mathsf{Abort} = \lambda r.\ \mathsf{Zero}) \\
& \wedge (\mathsf{wp}\ \mathsf{Skip} = \lambda r.\ r) \\
& \wedge (\mathsf{wp}\ (\mathsf{Assign}\ v\ e) = \lambda r, s.\ r\ (\lambda w.\ \mathsf{if}\ w = v\ \mathsf{then}\ e\ s\ \mathsf{else}\ s\ w)) \\
& \wedge (\mathsf{wp}\ (\mathsf{Seq}\ c_1\ c_2) = \lambda r.\ \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ r)) \\
& \wedge (\mathsf{wp}\ (\mathsf{Demon}\ c_1\ c_2) = \lambda r.\ \mathsf{Min}\ (\mathsf{wp}\ c_1\ r)\ (\mathsf{wp}\ c_2\ r)) \\
& \wedge (\mathsf{wp}\ (\mathsf{Prob}\ p\ c_1\ c_2) = \\
& \qquad \lambda r, s.\ \mathsf{let}\ x \leftarrow [p\ s]_{\leq 1}\ \mathsf{in}\ x(\mathsf{wp}\ c_1\ r\ s) + (1 - x)(\mathsf{wp}\ c_2\ r\ s)) \\
& \wedge (\mathsf{wp}\ (\mathsf{While}\ b\ c) = \\
& \qquad \lambda r.\ \mathsf{expect\_lfp}\ (\lambda e, s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ \mathsf{wp}\ c\ e\ s\ \mathsf{else}\ r\ s))
\end{aligned}
$$

Weakest Preconditions: Example

- The goal is to end up with variables i and j containing the same value:

$$\textit{post} \;\equiv\; \text{if } i = j \text{ then } 1 \text{ else } 0.$$

- First program:

$$\begin{aligned} \mathsf{pd} \;&\equiv\; i := \langle 0, 1 \rangle \;;\; j := \{0, 1\} \\ &\vdash\; \mathsf{wp}\ \mathsf{pd}\ \textit{post} = \mathsf{Zero} \end{aligned}$$

- Second program:

$$\begin{aligned} \mathsf{dp} \;&\equiv\; j := \{0, 1\} \;;\; i := \langle 0, 1 \rangle \\ &\vdash\; \mathsf{wp}\ \mathsf{dp}\ \textit{post} = \lambda s.\ 1/2. \end{aligned}$$

Contents

Weakest Liberal Preconditions

Weakest liberal conditions (wlp) model partial correctness.

$$
\begin{aligned}
\vdash \quad & (\mathsf{wlp}\ (\mathsf{Assert}\ p\ c) = \mathsf{wlp}\ c) \\
& \wedge (\mathsf{wlp}\ \mathsf{Abort} = \lambda r.\ \mathsf{Magic}) \\
& \wedge (\mathsf{wlp}\ \mathsf{Skip} = \lambda r.\ r) \\
& \wedge (\mathsf{wlp}\ (\mathsf{Assign}\ v\ e) = \lambda r, s.\ r\ (\lambda w.\ \mathsf{if}\ w = v\ \mathsf{then}\ e\ s\ \mathsf{else}\ s\ w)) \\
& \wedge (\mathsf{wlp}\ (\mathsf{Seq}\ c_1\ c_2) = \lambda r.\ \mathsf{wlp}\ c_1\ (\mathsf{wlp}\ c_2\ r)) \\
& \wedge (\mathsf{wlp}\ (\mathsf{Demon}\ c_1\ c_2) = \lambda r.\ \mathsf{Min}\ (\mathsf{wlp}\ c_1\ r)\ (\mathsf{wlp}\ c_2\ r)) \\
& \wedge (\mathsf{wlp}\ (\mathsf{Prob}\ p\ c_1\ c_2) = \\
& \qquad \lambda r, s.\ \mathsf{let}\ x \leftarrow [p\ s]_{\leq 1}\ \mathsf{in}\ x(\mathsf{wlp}\ c_1\ r\ s) + (1 - x)(\mathsf{wlp}\ c_2\ r\ s)) \\
& \wedge (\mathsf{wlp}\ (\mathsf{While}\ b\ c) = \\
& \qquad \lambda r.\ \mathsf{expect\_gfp}\ (\lambda e, s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ \mathsf{wlp}\ c\ e\ s\ \mathsf{else}\ r\ s))
\end{aligned}
$$

Weakest Liberal Preconditions: Example

- We illustrate the difference between wp and wlp on the simplest infinite loop:

$$\mathsf{loop} \;\equiv\; \mathsf{While}\ (\lambda s.\ \top)\ \mathsf{Skip}$$

- For any postcondition *post*, we have

$$\vdash\ \mathsf{wp}\ \mathsf{loop}\ \textit{post} = \mathsf{Zero}\ \wedge\ \mathsf{wlp}\ \mathsf{loop}\ \textit{post} = \mathsf{Magic}$$

- These correspond to the Hoare triples

$$[\bot]\ \mathsf{loop}\ [\textit{post}] \qquad \{\top\}\ \mathsf{loop}\ \{\textit{post}\}$$

as we would expect from an infinite loop.

Calculating wlp Lower Bounds

- Suppose we have a pGCL command c and a postcondition q.
- We wish to derive a lower bound on the weakest liberal precondition.
- Can think of this as the first-order query $P \sqsubseteq \text{wlp}\ c\ q$.
- Idea: use a Prolog interpreter to solve for the variable P.

Calculating wlp: Rules

Example Rules:

- $\mathsf{Magic} \sqsubseteq \mathsf{wlp}\ \mathsf{Abort}\ Q$
- $Q \sqsubseteq \mathsf{wlp}\ \mathsf{Skip}\ Q$
- $R \sqsubseteq \mathsf{wlp}\ C_2\ Q\ \wedge\ P \sqsubseteq \mathsf{wlp}\ C_1\ R\ \Rightarrow$
  $P \sqsubseteq \mathsf{wlp}\ (\mathsf{Seq}\ C_1\ C_2)\ Q$
- $P_1 \sqsubseteq \mathsf{wlp}\ C_1\ Q\ \wedge\ P_2 \sqsubseteq \mathsf{wlp}\ C_2\ Q\ \Rightarrow$
  $\mathsf{Min}\ P_1\ P_2 \sqsubseteq \mathsf{wlp}\ (\mathsf{Demon}\ C_1\ C_2)\ Q$

Note: the Prolog interpreter automatically calculates the 'middle condition' in a Seq command.

Calculating wlp: While Loops

- We use the following theorem about While loops:

$$\begin{array}{ll} \vdash & \forall P, Q, b, c. \\ & \quad P \sqsubseteq \mathsf{If}\ b\ (\mathsf{wlp}\ c\ P)\ Q \Rightarrow P \sqsubseteq \mathsf{wlp}\ (\mathsf{While}\ b\ c)\ Q \end{array}$$

- Cannot use in this form, because of the repeated occurrence of P in the premise.
- Instead, provide a rule that requires an assertion:
  - $R \sqsubseteq \mathsf{wlp}\ C\ P \ \wedge\ P \sqsubseteq \mathsf{If}\ b\ R\ Q \ \Rightarrow$ $P \sqsubseteq \mathsf{wlp}\ (\mathsf{Assert}\ P\ (\mathsf{While}\ b\ c))\ Q$
- The second premise generates a *verification condition* as an extra subgoal.
- It is left to the user to provide a useful loop invariant in the Assert around the while loop.

Contents

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - wlp Verification Condition Generator
  - **Example Verifications**
- Conclusion

Example: Monty Hall

$$
\begin{array}{l}
\textsf{contestant}\ \textit{switch}\ \equiv \\
\quad pc := \{1, 2, 3\}\ ; \\
\quad cc := \langle 1, 2, 3 \rangle\ ; \\
\qquad \begin{array}{cccc}
 & pc \neq 1 \land cc \neq 1 & \rightarrow & ac := 1 \\
| & pc \neq 2 \land cc \neq 2 & \rightarrow & ac := 2 \\
| & pc \neq 3 \land cc \neq 3 & \rightarrow & ac := 3\ ;
\end{array} \\
\quad \textsf{if}\ \neg\textit{switch}\ \textsf{then Skip else} \\
\qquad cc := (\textsf{if}\ cc \neq 1 \land ac \neq 1\ \textsf{then}\ 1 \\
\qquad\qquad\qquad \textsf{else if}\ cc \neq 2 \land ac \neq 2\ \textsf{then}\ 2\ \textsf{else}\ 3)
\end{array}
$$

The postcondition is simply the desired goal of the contestant, i.e.,

$$\textsf{win}\ \equiv\ \textsf{if}\ cc = pc\ \textsf{then}\ 1\ \textsf{else}\ 0.$$

Example: Monty Hall

- Verification proceeds by:
  1. Rewriting away all the syntactic sugar.
  2. Expanding the definition of wp.
  3. Carrying out the numerical calculations.
- After 22 seconds and 250536 primitive inferences in the logical kernel:

$$\vdash \ \mathsf{wp}\ (\mathsf{contestant}\ \textit{switch})\ \mathsf{win} = \lambda s.\ \mathsf{if}\ \textit{switch}\ \mathsf{then}\ 2/3\ \mathsf{else}\ 1/3$$

- In other words, by switching the contestant is twice as likely to win the prize.
- Not trivial to do by hand, because the intermediate expectations get rather large.

Example: Rabin Mutual Exclusion

- Suppose N processors are executing concurrently, and from time to time some of them need to enter a critical section of code.
- The mutual exclusion algorithm of Rabin (1982, 1992) works by electing a leader who is permitted to enter the critical section:
  1. Each of the waiting processors repeatedly tosses a fair coin until a head is shown
  2. The processor that required the largest number of tosses wins the election.
  3. If there is a tie, then have another election.
- Could implement the coin tossing using

  $n := 0\ ;\ b := 0\ ;$ While $(b = 0)\ (n := n + 1\ ;\ b := \langle 0, 1 \rangle)$

Example: Rabin Mutual Exclusion

For our verification, we do not model i processors concurrently executing the above voting scheme, but rather the following data refinement of that system:

1. Initialize i with the number of processors waiting to enter the critical section who have just picked a number.
2. Initialize n with 1, the lowest number not yet considered.
3. If $i = 1$ then we have a unique winner: return SUCCESS.
4. If $i = 0$ then the election has failed: return FAILURE.
5. Reduce i by eliminating all the processors who picked the lowest number n (since certainly none of them won the election).
6. Increment n by 1, and jump to Step 3.

Example: Rabin Mutual Exclusion

The following pGCL program implements this data refinement:

$$
\begin{aligned}
\text{rabin} \quad \equiv \quad & \text{While } (1 < i) \ (\\
& \quad n := i \ ; \\
& \quad \text{While } (0 < n) \\
& \quad\quad (d := \langle 0, 1 \rangle \ ; \ i := i - d \ ; \ n := n - 1) \\
&)
\end{aligned}
$$

The desired postcondition representing a unique winner of the election is

$$\textit{post} \ \equiv \ \text{if } i = 1 \text{ then } 1 \text{ else } 0$$

Example: Rabin Mutual Exclusion

- The precondition that we aim to show is

$$\textit{pre} \;\equiv\; \text{if } i = 1 \text{ then } 1 \text{ else if } 1 < i \text{ then } 2/3 \text{ else } 0$$

"For any positive number of processors wanting to enter the critical section, the probability that the voting scheme will produce a unique winner is $2/3$, except for the trivial case of one processor when it will always succeed."

- Surprising: The probability of success is independent of the number of processors.
- We formally verify the following statement of partial correctness:

$$\textit{pre} \sqsubseteq \text{wlp rabin } \textit{post}$$

Example: Rabin Mutual Exclusion

- Need to annotate the While loops with invariants.
- The invariant for the outer loop is simply *pre*.
- For the inner loop we used

$$\text{if } 0 \le n \le i \text{ then } (2/3) * \text{invar1 } i\ n + \text{invar2 } i\ n \text{ else } 0$$

where

$$\begin{aligned}
\text{invar1 } i\ n \ &\equiv \\
&1 - (\text{if } i = n \text{ then } (n+1)/2^n \text{ else if } i = n+1 \text{ then } 1/2^n \text{ else } 0) \\
\text{invar2 } i\ n \ &\equiv \ \text{if } i = n \text{ then } n/2^n \text{ else if } i = n+1 \text{ then } 1/2^n \text{ else } 0
\end{aligned}$$

- Coming up with these was the hardest part of the verification.

Example: Rabin Mutual Exclusion

The verification proceeded as follows:

1. Create the annotated program annotated_rabin.
2. Prove $\mathsf{wlp}\ \mathsf{rabin} = \mathsf{wlp}\ \mathsf{annotated\_rabin}$
3. Use this to reduce the goal to

$$\mathit{pre} \sqsubseteq \mathsf{wlp}\ \mathsf{annotated\_rabin}\ \mathit{post}$$

4. This is in the correct form to apply the VC generator.
5. Finish off the VCs with 58 lines of HOL-4 proof script.

```
|- Leq (\s. if s"i" = 1 then 1
            else if 1 < s"i" then 2/3 else 0)
   (wlp rabin (\s. if s"i" = 1 then 1 else 0))
```

Contents

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - wlp Verification Condition Generator
  - Example Verifications
- **Conclusion**

Conclusion

Advantages of Monad Approach

- Grounded in measure theory.
  - Probabilities more than real numbers.
- More suitable for verifying functional programs.
  - Simple to lift verified HOL functions to ML.
- Can reason about the distinction between probabilistic and guaranteed termination.
  - Practical difference: operating systems typically provide a source of random bits.

Conclusion

Advantages of pGCL Approach

- Supports the demonic choice programming construct.
  - Can be used to verify distributed algorithms.
- Verification easier to carry out than monad approach.
  - Modelling programs with expectation transformers is a useful abstraction.
- Deep embedding: can quantify over all programs.
  - May be useful for modelling a ‘spy’ in a security protocol verification.

Future Work: combine these approaches to get the best of both worlds.

Related Work

- Formal methods for probabilistic programs:
  - Hurd's thesis, 2002.
  - Probabilistic invariants for probabilistic machines, Hoang et. al., 2003.
  - Christine Paulin's work in Coq, 2002.
  - Prism model checker, Kwiatkowska et. al., 2000–
- Mechanized program semantics:
  - Formalizing Dijkstra, Harrison, 1998.
  - Hoare Logics in Isabelle/HOL, Nipkow, 2001.
  - Mechanizing program logics in higher order logic, Gordon, 1989.
  - A mechanically verified verification condition generator, Homeier and Martin, 1995.

Related Work

- Semantics of Probabilistic Programs:
  - *Semantics of Probabilistic Programs*, Kozen, 1979.
  - *Termination of Probabilistic Concurrent Processes*, Hart, Sharir and Pnueli, 1983.
  - *Probabilistic Non-Determinism*, Jones, 1990.
  - 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