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
  - 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

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):

  
  ```ml
  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.
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
  - \( \text{wlp} \) 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;

\[ \text{prob}_{\text{program}} : \mathbb{N} \to \mathbb{B}^\infty \to \{\text{success, failure}\} \times \mathbb{B}^\infty \]

- then finally prove that the program satisfies its specification.

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

- Introduction
- Approach 1: Monads
  - **Formalizing Probability**
    - Modelling Probabilistic Programs
    - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - $\omega lp$ Verification Condition Generator
  - Example Verifications
- Conclusion
Need to construct a probability space of Bernoulli\( \left( \frac{1}{2} \right) \) 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):

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

- Finally extended this measure to a \( \sigma \)-algebra:

  \[\begin{align*}
  \mathcal{E} &= \sigma(\text{prefix sets}) \\
  \mathbb{P} &= \text{Carathéodory extension of } \mu \text{ to } \mathcal{E}
  \end{align*}\]

- 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
  - $\omegalp$ Verification Condition Generator
    - 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:

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

- For example, if \texttt{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:

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

- Correctness theorem:

  \[ \Downarrow \quad \forall n, r. \quad \mathbb{P}\{s \mid \text{fst (binomial } n s) = r\} = \binom{n}{r} \left(\frac{1}{2}\right)^n \]
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):

\begin{itemize}
  \item The backward loops introduce the possibility of looping forever.
  \item But the probability of this happening is 0.
  \item \textbf{Probabilistic termination:} the program terminates with probability 1.
\end{itemize}
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\( (\frac{1}{3}) \): 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:

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

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
- Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - $\omega\text{LP}$ Verification Condition Generator
  - Example Verifications
- Conclusion
Example: The Uniform(3) Distribution

- First make a raw definition of \textit{unif3}:
  
  \[ \vdash \text{unif3} = \]
  
  \[ \text{while } (\lambda n. n = 3) \]
  
  \[ (\text{coin\_flip } (\text{coin\_flip } (\text{unit } 0) \ (\text{unit } 1)) \ (\text{coin\_flip } (\text{unit } 2) \ (\text{unit } 3))) \ 3 \]

- Next prove \textit{unif3} satisfies probabilistic termination.

- This allows us to derive a recursive definition of \textit{unif3}:
  
  \[ \vdash \text{unif3} = \text{coin\_flip } (\text{coin\_flip } (\text{unit } 0) \ (\text{unit } 1)) \ (\text{coin\_flip } (\text{unit } 2) \ \text{unif3}) \]

- The correctness theorem also follows:
  
  \[ \vdash \forall n. \ P\{s \mid \text{fst } (\text{unif3 } s) = n\} = \text{if } n < 3 \text{ then } \frac{1}{3} \text{ else } 0 \]
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))))
Example: Optimal Dice

- Correctness theorem:

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

- 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 \( \frac{47}{18} \) coin flips to produce a result, and this is also optimal.

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

\[
\begin{align*}
\vdash \ \forall n, k. \\
& \text{walk } n \ k = \\
& \quad \text{if } n = 0 \text{ then unit } k \text{ else} \ \\
& \quad \text{coin\_flip (walk } (n+1) \ (k+1)) \ (\text{walk } (n-1) \ (k+1))
\end{align*}
\]

- And prove some neat properties:

\[
\begin{align*}
\vdash \ \forall n, k. \ \forall^* s. \ 
& \text{even } (\text{fst (walk } n \ k \ s)) = \text{even } (n + k)
\end{align*}
\]
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:

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

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 `miller` in the logic to prove numbers composite:

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

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

<table>
<thead>
<tr>
<th>bits</th>
<th>El,n</th>
<th>MR</th>
<th>Gen time</th>
<th>MR1 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
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - \textit{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

\[ c_1 \ \boxslash \ c_2 \]

- 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
    - \texttt{wp} 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 $\alpha \rightarrow \text{posreal}$.
  - Expectation $P$ is weaker than $P'$ if $P' \sqsubseteq P$.
  - Think of programs as expectation transformers.
Model pGCL commands with a HOL datatype:

\[
\text{command} \equiv \text{Assert of } (\text{state} \rightarrow \text{posreal}) \times \text{command} \\
\mid \text{Abort} \\
\mid \text{Skip} \\
\mid \text{Assign of string } \times (\text{state} \rightarrow \mathbb{Z}) \\
\mid \text{Seq of command } \times \text{command} \\
\mid \text{Demon of command } \times \text{command} \\
\mid \text{Prob of } (\text{state} \rightarrow \text{posreal}) \times \text{command} \times \text{command} \\
\mid \text{While of } (\text{state} \rightarrow \mathbb{B}) \times \text{command}
\]

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

Define the following *derived commands* as syntactic sugar:

\[
\begin{align*}
v & := e & \equiv & \text{Assign } v e \\
c_1 \; ; \; c_2 & \equiv & \text{Seq } c_1 \; c_2 \\
c_1 \; \triangleright \; c_2 & \equiv & \text{Demon } c_1 \; c_2 \\
c_1 \; p \oplus \; c_2 & \equiv & \text{Prob } (\lambda s. \; p) \; c_1 \; c_2 \\
\text{Cond } b \; c_1 \; c_2 & \equiv & \text{Prob } (\lambda s. \; \text{if } b \; s \; \text{then } 1 \; \text{else } 0) \; c_1 \; c_2 \\
v & := \{e_1, \ldots, e_n\} & \equiv & v := e_1 \; \triangleright \; \cdots \; \triangleright \; v := e_n \\
v & := \langle e_1, \ldots, e_n \rangle & \equiv & v := e_1 \; 1/n \oplus \; v := \langle e_2, \ldots, e_n \rangle \\
p_1 \rightarrow c_1 \mid \cdots \mid p_n \rightarrow c_n & \equiv & \\
& \begin{cases} 
\text{Abort} & \text{if none of the } p_i \text{ hold on the current state} \\
\prod_{i \in I} c_i & \text{where } I = \{ i \mid 1 \leq i \leq n \land p_i \text{ holds} \}
\end{cases}
\end{align*}
\]

In addition, we write \( v := n + 1 \) instead of \( "v" := \lambda s. \; s \; "n" \; + \; 1 \).
Weakest Preconditions

Define weakest preconditions ($wp$) directly on commands:

\[
\vdash (wp \ (\text{Assert} \ p \ c) = wp \ c) \\
\land (wp \ \text{Abort} = \lambda r. \ \text{Zero}) \\
\land (wp \ \text{Skip} = \lambda r. \ r) \\
\land (wp \ (\text{Assign} \ v \ e) = \lambda r, s. \ r \ (\lambda w. \ \text{if} \ w = v \ \text{then} \ e \ s \ \text{else} \ s \ w)) \\
\land (wp \ (\text{Seq} \ c_1 \ c_2) = \lambda r. \ wp \ c_1 \ (wp \ c_2 \ r)) \\
\land (wp \ (\text{Demon} \ c_1 \ c_2) = \lambda r. \ \text{Min} \ (wp \ c_1 \ r) \ (wp \ c_2 \ r)) \\
\land (wp \ (\text{Prob} \ p \ c_1 \ c_2) = \\
\quad \lambda r, s. \ \text{let} \ x \leftarrow \lfloor p \ s \rfloor \leq 1 \ \text{in} \ x(wp \ c_1 \ r \ s) + (1 - x)(wp \ c_2 \ r \ s)) \\
\land (wp \ (\text{While} \ b \ c) = \\
\quad \lambda r. \ \text{expect}_\text{lfp} \ (\lambda e, s. \ \text{if} \ b \ s \ \text{then} \ wp \ c \ e \ s \ \text{else} \ r \ s))
\]
Weakest Preconditions: Example

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

\[ \text{post} \equiv \text{if } i = j \text{ then } 1 \text{ else } 0. \]

- First program:

\[ \text{pd} \equiv i := \langle 0, 1 \rangle \land j := \{0, 1\} \]
\[ \vdash \wp \text{ pd } \text{post} = \text{Zero} \]

- Second program:

\[ \text{dp} \equiv j := \{0, 1\} \land i := \langle 0, 1 \rangle \]
\[ \vdash \wp \text{ dp } \text{post} = \lambda s. 1/2. \]
Contents

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - \( \text{wlp Verification Condition Generator} \)
  - Example Verifications
- Conclusion
Weakest Liberal Preconditions

Weakest liberal conditions ($wlp$) model partial correctness.

\[\vdash (wlp \ (Assert \ p \ c) = wlp \ c) \]
\[\land (wlp \ \text{Abort} = \lambda r. \ Magic)\]
\[\land (wlp \ \text{Skip} = \lambda r. \ r)\]
\[\land (wlp \ (Assign \ v \ e) = \lambda r, s. \ r \ (\lambda w. \ \text{if} \ w = v \ \text{then} \ e \ s \ \text{else} \ s \ w))\]
\[\land (wlp \ (Seq \ c_1 \ c_2) = \lambda r. \ wlp \ c_1 \ (wlp \ c_2 \ r))\]
\[\land (wlp \ (Demon \ c_1 \ c_2) = \lambda r. \ \text{Min} \ (wlp \ c_1 \ r) \ (wlp \ c_2 \ r))\]
\[\land (wlp \ (Prob \ p \ c_1 \ c_2) = \]
\[\quad \lambda r, s. \ \text{let} \ x \leftarrow [p \ s]_{\leq 1} \ \text{in} \ x(wlp \ c_1 \ r \ s) + (1 - x)(wlp \ c_2 \ r \ s))\]
\[\land (wlp \ (While \ b \ c) = \]
\[\quad \lambda r. \ \text{expect}_\text{gfp} \ (\lambda e, s. \ \text{if} \ b \ s \ \text{then} \ wlp \ c \ e \ s \ \text{else} \ r \ s))\]
Weakest Liberal Preconditions: Example

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

\[
\text{loop} \equiv \text{While } (\lambda s. \top) \text{ Skip}
\]

- For any postcondition \( post \), we have

\[
\vdash wp \text{ loop } post = \text{Zero} \land wlp \text{ loop } post = \text{Magic}
\]

- These correspond to the Hoare triples

\[
[\bot] \text{ loop } [post] \quad \{\top\} \text{ loop } \{post\}
\]

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 $P \sqsubset wlp\ c\ q$.
- **Idea:** use a Prolog interpreter to solve for the variable $P$. 
Calculating $\text{wlp}$: Rules

Example Rules:

- Magic $\sqsubseteq \text{wlp} \text{ Abort } Q$
- $Q \sqsubseteq \text{wlp} \text{ Skip } Q$
- $R \sqsubseteq \text{wlp} \ C_2 \ Q \ \land \ \ P \sqsubseteq \text{wlp} \ C_1 \ R \ \Rightarrow$
  \[ P \sqsubseteq \text{wlp} \ (\text{Seq} \ C_1 \ C_2) \ Q \]
- $P_1 \sqsubseteq \text{wlp} \ C_1 \ Q \ \land \ \ P_2 \sqsubseteq \text{wlp} \ C_2 \ Q \ \Rightarrow$
  \[ \text{Min} \ P_1 \ P_2 \sqsubseteq \text{wlp} \ (\text{Demon} \ C_1 \ C_2) \ Q \]

Note: the Prolog interpreter automatically calculates the ‘middle condition’ in a $\text{Seq}$ command.
Calculating $wlp$: While Loops

- We use the following theorem about While loops:

$$\vdash \forall P, Q, b, c. \quad P \sqsubseteq \text{If } b \ (wlp \ c \ P) \ Q \Rightarrow P \sqsubseteq wlp \ (\text{While } b \ c) \ Q$$

- Cannot use in this form, because of the repeated occurrence of $P$ in the premise.

- Instead, provide a rule that requires an assertion:

$$R \sqsubseteq wlp \ C \ P \ \land \ P \sqsubseteq \text{If } b \ R \ Q \ \Rightarrow$$

$$P \sqsubseteq wlp \ (\text{Assert } P \ (\text{While } b \ c)) \ Q$$

- 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

contestant \textit{switch} \equiv \begin{align*}
 pc &:= \{1, 2, 3\} ; \\
c c &:= \langle 1, 2, 3 \rangle ; \\
 & \quad \text{if } pc \neq 1 \land cc \neq 1 \quad \rightarrow \quad ac := 1 \\
 | & \quad \text{if } pc \neq 2 \land cc \neq 2 \quad \rightarrow \quad ac := 2 \\
 | & \quad \text{if } pc \neq 3 \land cc \neq 3 \quad \rightarrow \quad ac := 3 ; \\
 & \text{if } \neg \text{switch} \text{ then Skip else} \\
 c c &:= (\text{if } cc \neq 1 \land ac \neq 1 \text{ then } 1 \\
 & \quad \text{else if } cc \neq 2 \land ac \neq 2 \text{ then } 2 \text{ else } 3) \\
\end{align*}

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

\[ \text{win} \equiv \text{if } cc = pc \text{ then } 1 \text{ else } 0. \]
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:

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

- 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:

\[
\text{rabin} \equiv \text{While } (1 < i) ( \\
\quad n := i ; \\
\quad \text{While } (0 < n) \\
\quad \quad (d := \langle 0, 1 \rangle ; i := i \mod d ; n := n - 1) \\
\quad )
\]

The desired postcondition representing a unique winner of the election is

\[
\text{post} \equiv \text{if } i = 1 \text{ then 1 else 0}
\]
Example: Rabin Mutual Exclusion

- The precondition that we aim to show is

\[ pre \equiv \text{if } i = 1 \text{ then } 1 \text{ else if } 1 < i \text{ then } 2/3 \text{ else } 0 \]

“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:

\[ pre \sqsubseteq \text{wp rabin } post \]
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

  \[
  \begin{align*}
  \text{if } 0 \leq n \leq i \text{ then } & (2/3) \times \text{invar1 } i \ n + \text{invar2 } i \ n \text{ else } 0 \\
  \text{where } \\
  \text{invar1 } i \ n & \equiv 1 - (\text{if } i = n \text{ then } (n + 1)/2^n \text{ else if } i = n + 1 \text{ then } 1/2^n \text{ else } 0) \\
  \text{invar2 } i \ n & \equiv \text{if } i = n \text{ then } n/2^n \text{ else if } i = n + 1 \text{ then } 1/2^n \text{ else } 0
  \end{align*}
  \]

- 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 $\text{wlp rabin} = \text{wlp annotated_rabin}$
3. Use this to reduce the goal to
   
   $$\text{pre} \sqsubseteq \text{wlp annotated_rabin post}$$

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.

|-$\text{Leq}\ (\	ext{s. if s"i" = 1 then 1 else if 1 < s"i" then 2/3 else 0})$

\(\text{(wlp rabin (\text{s. if s"i" = 1 then 1 else 0}))}\)
Contents

- Introduction
- Approach 1: Monads
  - Formalizing Probability
  - Modelling Probabilistic Programs
  - Example Verifications
- Approach 2: pGCL
  - Formalizing Probabilistic Guarded Commands
  - \( \mu p \) 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:
  - 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:
  - Mechanizing program logics in higher order logic, Gordon, 1989.
Related Work

- Semantics of Probabilistic Programs:
  - Probabilistic predicate transformers, Morgan, Mclver, Seidel and Sanders, 1994–
    - *Proof Rules for Probabilistic Loops*, Morgan, 1996