Probabilistic Guarded Commands Mechanized in HOL

Joe Hurd
joe.hurd@comlab.ox.ac.uk

Oxford University

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

Contents

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–

The HOL Theorem Prover

- Developed by Mike Gordon's Hardware Verification Group in Cambridge, first release was HOL88.
- Latest release called HOL4, developed jointly by Cambridge, Utah and ANU.
- Implements classical Higher-Order Logic: essentially first-order logic with quantification over functions.
- Sprung from the Edinburgh LCF project, so has a small logical kernel to ensure soundness.

Contents

pGCL Semantics

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

$$[P]C[Q]$$

- Precondition P is weaker than P' if $P' \Rightarrow P$.
- Think of C as a function that transforms postconditions into weakest preconditions.
- pGCL generalizes this to probabilistic programs:
  - Conditions $\alpha \to \mathbb{B}$ become *expectations* $\alpha \to [0, +\infty]$.
  - Expectation P is weaker than P' if $P' \sqsubseteq P$.
  - Think of programs as *expectation transformers*.

Expectations

- Expectations are reward functions, from states to expected rewards.
- Modelled in HOL as functions $\alpha \to [0, +\infty]$.
- Define the following operations on expectations:
  - $\mathsf{Min}\ e_1\ e_2 \ \equiv\ \lambda s.\ \mathsf{min}\ (e_1\ s)\ (e_2\ s)$
  - $e_1 \sqsubseteq e_2 \ \equiv\ \forall s.\ e_1\ s \leq e_2\ s$
  - $\mathsf{Cond}\ b\ e_1\ e_2 \ \equiv\ \lambda s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ e_1\ s\ \mathsf{else}\ e_2\ s$
  - $\mathsf{Lin}\ p\ e_1\ e_2 \ \equiv\ \lambda s.\ [p\ s]^{\leq 1} \times e_1\ s + (1 - [p\ s]^{\leq 1}) \times e_2\ s$

States

- Fix states to be mappings from variable names to integers:

$$\mathsf{state} \;\equiv\; \mathsf{string} \to \mathbb{Z}$$

- For convenience, define a state update function:

$$\mathsf{assign}\ v\ f\ s \;\equiv\; \lambda w.\ \mathsf{if}\ v = w\ \mathsf{then}\ f\ s\ \mathsf{else}\ s\ w$$

pGCL Commands

Model pGCL commands with a HOL datatype:

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

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

Derived Commands

Define the following *derived commands* as syntactic sugar:

$$
\begin{array}{rcl}
v := f & \equiv & \mathsf{Assign}\ v\ f \\
c_1\ ;\ c_2 & \equiv & \mathsf{Seq}\ c_1\ c_2 \\
c_1\ \sqcap\ c_2 & \equiv & \mathsf{Demon}\ c_1\ c_2 \\
c_1\ {}_p\!\oplus\ c_2 & \equiv & \mathsf{Prob}\ (\lambda s.\ p)\ c_1\ c_2 \\
\mathsf{Cond}\ b\ c_1\ c_2 & \equiv & \mathsf{Prob}\ (\lambda s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ c_1\ c_2 \\
v := \{e_1, \ldots, e_n\} & \equiv & v := e_1\ \sqcap\ \cdots\ \sqcap\ v := e_n \\
v := \langle e_1, \cdots, e_n\rangle & \equiv & v := e_1\ {}_{1/n}\!\oplus\ v := \langle e_2, \ldots, e_n\rangle \\
b_1 \to c_1 \mid \cdots \mid b_n \to c_n & \equiv &
\end{array}
$$

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

Weakest Preconditions

Define weakest preconditions (wp) directly on commands:

$$
\begin{array}{ll}
\vdash & (\mathsf{wp}\ \mathsf{Abort} = \lambda e.\ \mathsf{Zero}) \\
\wedge & (\mathsf{wp}\ \mathsf{Skip} = \lambda e.\ e) \\
\wedge & (\mathsf{wp}\ (\mathsf{Assign}\ v\ f) = \lambda e, s.\ e\ (\mathsf{assign}\ v\ f\ s)) \\
\wedge & (\mathsf{wp}\ (\mathsf{Seq}\ c_1\ c_2) = \lambda e.\ \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ e)) \\
\wedge & (\mathsf{wp}\ (\mathsf{Demon}\ c_1\ c_2) = \lambda e.\ \mathsf{Min}\ (\mathsf{wp}\ c_1\ e)\ (\mathsf{wp}\ c_2\ e)) \\
\wedge & (\mathsf{wp}\ (\mathsf{Prob}\ p\ c_1\ c_2) = \lambda e.\ \mathsf{Lin}\ p\ (\mathsf{wp}\ c_1\ e)\ (\mathsf{wp}\ c_2\ e)) \\
\wedge & (\mathsf{wp}\ (\mathsf{While}\ b\ c) = \lambda e.\ \mathsf{expect\_lfp}\ (\lambda e'.\ \mathsf{Cond}\ b\ (\mathsf{wp}\ c\ e')\ e))
\end{array}
$$

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:

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

- Second program:

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

Example: Monty Hall

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

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.

Contents

- Introduction
- Formalizing Probabilistic Guarded Commands
- wlp **Verification Condition Generator**
- Example: Rabin's Mutual Exclusion Algorithm
- Conclusion

Weakest Liberal Preconditions

Weakest liberal conditions (wlp) model partial correctness.

$$
\begin{array}{ll}
\vdash & (\mathsf{wlp}\ \mathsf{Abort} = \lambda e.\ \mathsf{Infty}) \\
\wedge & (\mathsf{wlp}\ \mathsf{Skip} = \lambda e.\ e) \\
\wedge & (\mathsf{wlp}\ (\mathsf{Assign}\ v\ f) = \lambda e, s.\ e\ (\mathsf{assign}\ v\ f\ s)) \\
\wedge & (\mathsf{wlp}\ (\mathsf{Seq}\ c_1\ c_2) = \lambda e.\ \mathsf{wlp}\ c_1\ (\mathsf{wlp}\ c_2\ e)) \\
\wedge & (\mathsf{wlp}\ (\mathsf{Demon}\ c_1\ c_2) = \lambda e.\ \mathsf{Min}\ (\mathsf{wlp}\ c_1\ e)\ (\mathsf{wlp}\ c_2\ e)) \\
\wedge & (\mathsf{wlp}\ (\mathsf{Prob}\ p\ c_1\ c_2) = \lambda e.\ \mathsf{Lin}\ p\ (\mathsf{wlp}\ c_1\ e)\ (\mathsf{wlp}\ c_2\ e)) \\
\wedge & (\mathsf{wlp}\ (\mathsf{While}\ b\ c) = \lambda e.\ \mathsf{expect\_gfp}\ (\lambda e'.\ \mathsf{Cond}\ b\ (\mathsf{wlp}\ c\ e')\ e))
\end{array}
$$

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

$$\vdash\ \mathsf{wp}\ \mathsf{loop}\ \mathit{post} = \mathsf{Zero}\ \wedge\ \mathsf{wlp}\ \mathsf{loop}\ \mathit{post} = \mathsf{Infty}$$

- These correspond to the Hoare triples

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

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.
  - In general, programs are shown to have desirable properties by proving *lower bounds*.
  - Example: $\vdash \ (\lambda s.\ 0.95) \ \sqsubseteq \ \mathsf{wp\ Prog\ (if\ ok\ then\ } 1 \mathsf{\ else\ } 0)$
- Can think of this as the query $P \sqsubseteq \mathsf{wlp}\ c\ q$.
- Idea: use a Prolog interpreter to solve for the variable P.

Calculating wlp: Rules

Simple rules:

- $\mathsf{Infty} \sqsubseteq \mathsf{wlp}\ \mathsf{Abort}\ Q$
- $Q \sqsubseteq \mathsf{wlp}\ \mathsf{Skip}\ Q$
- $R \sqsubseteq \mathsf{wlp}\ C_2\ Q\ \wedge\ P \sqsubseteq \mathsf{wlp}\ C_1\ R$
  $\Rightarrow$
  $P \sqsubseteq \mathsf{wlp}\ (\mathsf{Seq}\ C_1\ C_2)\ Q$

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

Calculating wlp: While Loops

- Define an assertion command: $\mathsf{Assert}\ p\ c\ \equiv\ c$.
- Provide a while rule that requires an assertion:
  - $R \sqsubseteq \mathsf{wlp}\ C\ P\ \wedge\ P \sqsubseteq \mathsf{wlp\_cond}\ b\ R\ Q$
    $\Rightarrow$
    $P \sqsubseteq \mathsf{wlp}\ (\mathsf{Assert}\ P\ (\mathsf{While}\ b\ c))\ Q$
- 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
- Formalizing Probabilistic Guarded Commands
- wlp Verification Condition Generator
- **Example: Rabin's Mutual Exclusion Algorithm**
- Conclusion

Rabin's Mutual Exclusion Algorithm

- 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

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

Rabin's Mutual Exclusion Algorithm

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.

Rabin's Mutual Exclusion Algorithm

The following pGCL program implements this data refinement:

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

The desired postcondition representing a unique winner of the election is

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

Rabin's Mutual Exclusion Algorithm

- 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:

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

Rabin's Mutual Exclusion Algorithm

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

$$\text{if } 0 \leq n \leq i \text{ then } 2/3 \times \text{invar1 } i\ n + \text{invar2 } i\ n \text{ else } 0$$

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.

Rabin's Mutual Exclusion Algorithm

The verification proceeded as follows:

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

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

4. This is now 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
- Formalizing Probabilistic Guarded Commands
- wlp Verification Condition Generator
- Example: Rabin’s Mutual Exclusion Algorithm
- **Conclusion**

Conclusion

- Formalized the theory of pGCL in higher-order logic.
  - Definitional theory, so high assurance of consistency.
- Created an automatic tool for deriving sufficient conditions for partial correctness.
  - Useful product of mechanizing a program semantics.
- HOL-4 well suited to this task.
  - Hard VCs can be passed to the user as subgoals.

Related Work

- Formal methods for probabilistic programs:
  - Christine Paulin's work in Coq, 2002.
  - Prism model checker, Kwiatkowska et. al., 2000–
- Mechanized program semantics:
  - Formalizing Dijkstra, Harrison, 1998.
  - Mechanizing program logics in higher order logic, Gordon, 1989.