Mechanizing the Probabilistic Guarded Command Language

Joe Hurd

Computing Laboratory
University of Oxford

IFIP Working Group 2.3
Tuesday 9 January 2007

Joint work with Carroll Morgan (UNSW), Annabelle McIver (Macquarie), Orieta Celiku (CMU) and Aaron Coble (Cambridge)
Talk Plan

1. Introduction
2. Formalizing pGCL
3. Verification Conditions
4. Current Work
5. Summary
Probabilistic Programs

Giving programs access to a random number generator is useful for many applications:

- Symmetry breaking
  - Rabin’s mutual exclusion algorithm
- Eliminating pathological cases
  - Randomized quicksort
- Gain in (best known?) theoretical complexity
  - Sorting nuts and bolts
- Solving a problem in an extremely simple way
  - Finding minimal cuts

**Research goal:** Apply formal methods to programs with probabilistic nondeterminism.
Probabilistic Guarded Command Language

- pGCL stands for Probabilistic Guarded Command Language.
- It’s Dijkstra’s GCL extended with probabilistic choice
  \[ c_1 \oplus p \oplus c_2 \]
- Like GCL, the semantics is based on weakest preconditions.
  - **Important**: retains nondeterministic choice
    \[ c_1 \sqcap c_2 \]
- Developed by Morgan, McIver et al. in Oxford and then Sydney, 1994–
The HOL4 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 (a.k.a. simple type theory).
- Sprung from the Edinburgh LCF project, so has a small logical kernel to ensure soundness.
Motivation

Why formalize?

- The theoretical results and program algebra are checked by logically deriving them from a simple set of definitions.
  - Example: Deriving the rules of Floyd-Hoare logic from a denotational semantics.
- When the program algebra is mechanized its feasibility can be checked by directly applying it to example programs.
  - Analysis tools that deduce from the semantics can be used to check other tools or generate test vectors.
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 $P' \implies P$.
- Think of the program $C$ 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 $\alpha \rightarrow [0, +\infty]$.
- Define the following operations on expectations:
  - $\text{Min } e_1 e_2 \equiv \lambda s. \min (e_1 s) (e_2 s)$
  - $e_1 \sqsubseteq e_2 \equiv \forall s. e_1 s \leq e_2 s$
  - $\text{Cond } b e_1 e_2 \equiv \lambda s. \text{if } b s \text{ then } e_1 s \text{ else } e_2 s$
  - $\text{Lin } p e_1 e_2 \equiv \lambda s. p s \times e_1 s + (1 - p s) \times e_2 s$
Expectation Transformers

- Expectation transformers are functions from expectations to expectations.
- Expectation transformers that correspond to probabilistic programs satisfy healthiness conditions:

  feasible \( t \equiv t \text{ Zero } = \text{ Zero} \)
  monotonic \( t \equiv \forall e_1, e_2. e_1 \sqsubseteq e_2 \implies t e_1 \sqsubseteq t e_2 \)
  scaling \( t \equiv \forall e, c. t (\lambda s. c \times e s) = \lambda s. c \times t e s \)
  subadditive \( t \equiv \forall e_1, e_2. t (\lambda s. e_1 s + e_2 s) \sqsubseteq \lambda s. t e_1 s + t e_2 s \)
  subtractive \( t \equiv \forall e, c. c \neq \infty \implies t (\lambda s. e s - c) \sqsubseteq \lambda s. t e s - c \)

- Expectations form a lattice, so expectation transformers can be up_continuous, have least and greatest fixed points, etc.
The definition of healthiness for expectation transformers is analogous to healthiness of predicate transformers in standard GCL:

\[
\text{healthy } t \equiv \text{feasible } t \land \text{sublinear } t \land \text{up-continuous } t
\]

where

\[
\text{sublinear } t \equiv \text{scaling } t \land \text{subadditive } t \land \text{subtractive } t
\]

Sublinearity in pGCL is the generalization of the conjunctivity condition in GCL.
States

- Fix states to be mappings from variable names to integers:
  \[
  \text{state} \equiv \text{string} \rightarrow \mathbb{Z}
  \]
- For convenience, define a state update function:
  \[
  \text{assign } v \ f \ s \equiv \lambda w. \text{ if } v = w \text{ then } f \ s \text{ else } s \ w
  \]
Model pGCL commands with a HOL datatype:

\[
\text{command} \equiv \text{Abort} \mid \text{Skip} \mid \text{Assign of string} \times (\text{state} \rightarrow \mathbb{Z}) \mid \text{Seq of command} \times \text{command} \mid \text{Nondet of command} \times \text{command} \mid \text{Prob of (state} \rightarrow [0, 1]) \times \text{command} \times \text{command} \mid \text{While of (state} \rightarrow \mathbb{B}) \times \text{command}
\]

**Note:** The probability in \text{Prob} can depend on the state.
Derived Commands

Define all other commands as syntactic sugar:

\[
\begin{align*}
\nu :&= f \quad \equiv \quad \text{Assign } \nu f \\
\text{c}_1 \mathbin; \text{c}_2 &\equiv \text{Seq } \text{c}_1 \text{c}_2 \\
\text{c}_1 \mathbin\sqcap \text{c}_2 &\equiv \text{Nondet } \text{c}_1 \text{c}_2 \\
\text{c}_1 \mathbin p\oplus \text{c}_2 &\equiv \text{Prob } (\lambda s. \ p) \text{c}_1 \text{c}_2 \\
\text{if } b \text{ then } \text{c}_1 \text{ else } \text{c}_2 &\equiv \text{Prob } (\lambda s. \text{if } b \ s \text{ then } 1 \text{ else } 0) \text{c}_1 \text{c}_2 \\
\nu :&= \{e_1, \ldots, e_n\} \equiv \nu :\equiv e_1 \mathbin\sqcap \cdots \mathbin\sqcap \nu :\equiv e_n \\
\nu :&= \langle e_1, \ldots, e_n \rangle \equiv \nu := e_1 1/n\oplus \nu := \langle e_2, \ldots, e_n \rangle \\
b_1 \rightarrow \text{c}_1 | \cdots | b_n \rightarrow \text{c}_n &\equiv \\
\left\{ \begin{array}{l}
\text{Abort} \quad \text{if none of the } b_i \text{ hold on the current state} \\
\prod_{i \in l} \text{c}_i \quad \text{where } l = \{i \mid 1 \leq i \leq n \land b_i \text{ holds}\}
\end{array} \right.
\end{align*}
\]
Weakest Preconditions

Define weakest preconditions (wp) directly on commands:

\[ \vdash (wp \text{ Abort} = \lambda e. \text{ Zero}) \]
\[ \land (wp \text{ Skip} = \lambda e. e) \]
\[ \land (wp (\text{Assign } v f) = \lambda e, s. e (\text{assign } v f s)) \]
\[ \land (wp (\text{Seq } c_1 c_2) = \lambda e. wp c_1 (wp c_2 e)) \]
\[ \land (wp (\text{Nondet } c_1 c_2) = \lambda e. \text{Min} (wp c_1 e) (wp c_2 e)) \]
\[ \land (wp (\text{Prob } p c_1 c_2) = \lambda e. \text{Lin} p (wp c_1 e) (wp c_2 e)) \]
\[ \land (wp (\text{While } b c) = \lambda e. \text{lfp} (\lambda e'. \text{Cond } b (wp c e') e)) \]
The major theorem of our formalization:

\[ \vdash \forall c. \text{healthy}(\text{wp } c) \]

Proof by structural induction (800 lines of HOL4 script).

The hardest part was sublinearity of while loops.

Needed several lemmas, for example:

\[ \vdash \forall t, e_1, e_2. \]
\[ \text{healthy } t \land \text{bounded } t \land e_2 \subseteq e_1 \implies t \left(\lambda s. e_1 s - e_2 s\right) \subseteq \lambda s. t e_1 s - t e_2 s \]
Example: Monty Hall

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

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:

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

- 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.
Weakest liberal preconditions (wlp) model partial correctness.

\[ \begin{align*}
\vdash & \ (\text{wlp Abort} = \lambda e. \ \text{Infty}) \\
\land & \ (\text{wlp Skip} = \lambda e. \ e) \\
\land & \ (\text{wlp (Assign } v \ f) = \lambda e, s. \ e \ (\text{assign } v \ f \ s)) \\
\land & \ (\text{wlp (Seq } c_1 \ c_2) = \lambda e. \ \text{wlp } c_1 \ (\text{wlp } c_2 \ e)) \\
\land & \ (\text{wlp (Nondet } c_1 \ c_2) = \lambda e. \ \text{Min} \ (\text{wlp } c_1 \ e) \ (\text{wlp } c_2 \ e)) \\
\land & \ (\text{wlp (Prob } p \ c_1 \ c_2) = \lambda e. \ \text{Lin} \ p \ (\text{wlp } c_1 \ e) \ (\text{wlp } c_2 \ e)) \\
\land & \ (\text{wlp (While } b \ c) = \lambda e. \ \text{gfp} \ (\lambda e'. \ \text{Cond} \ b \ (\text{wlp } c \ e') \ e))
\end{align*} \]
Weakest Liberal Preconditions: Example

- Consider the simplest infinite loop:
  
  \[
  \text{loop} \equiv \text{While } (\lambda s. \top) \text{ Skip}
  \]

- For any postcondition \( post \), we have
  
  \[\vdash \text{wp loop } post = \text{Zero } \land \text{ wlp loop } post = \text{Infty}\]

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

Simple rules:

- \( \text{Infty} \sqsubseteq \text{wlp} \) \( \text{Abort Q} \)
- \( Q \sqsubseteq \text{wlp} \) \( \text{Skip Q} \)
- \( R \sqsubseteq \text{wlp} \) \( C_2 Q \land P \sqsubseteq \text{wlp} \) \( C_1 R \)

\[ \implies \]

\( P \sqsubseteq \text{wlp} \) \((\text{Seq} \ C_1 \ C_2) \) \( Q \)

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

- Define an assertion command: Assert $p\ c \equiv \ c$.
- Provide a while rule that requires an assertion:
  - $R \sqsubseteq \text{wlp}\ C\ P \land P \sqsubseteq \text{Cond}\ B\ R\ Q$
  - $\Rightarrow P \sqsubseteq \text{wlp}\ (\text{Assert}\ P\ (\text{While}\ B\ C))\ Q$
- 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.
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 \( N \) 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:

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

The desired postcondition representing a unique winner of the election is

\[
\text{post } \equiv \text{ if } i = 1 \text{ then 1 else 0}
\]
Rabin’s Mutual Exclusion Algorithm

- The precondition that we aim to show is

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

“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:

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

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

where

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

$$\text{invar2 } i \ n \equiv \text{if } i = n \text{ then } n/2^n \text{ else if } i = n + 1 \text{ then } 1/2^n \text{ else } 0$$

- Coming up with these was the hardest part of the verification.
Rabin’s Mutual Exclusion Algorithm

The verification proceeded as follows:

1. Annotate the program to create the goal:

   \[ \text{pre} \subseteq \text{wlp annotated}_\text{rabin} \text{ post} \]

2. This is now in the correct form to apply the VC generator.

3. Finish off the VCs with 58 lines of HOL-4 proof script.

   \[ |- \text{Leq} (\forall s. \text{if } s"i" = 1 \text{ then } 1 \text{ else if } 1 < s"i" \text{ then } 2/3 \text{ else } 0) \]
   \[ (\text{wlp rabin} (\forall s. \text{if } s"i" = 1 \text{ then } 1 \text{ else } 0)) \]
This formalization started from the weakest precondition semantics of pGCL programs.

Instead can derive this from a relational semantics between initial states and probability distributions over final states:

$$\alpha \times (\alpha \rightarrow [0, 1]) \rightarrow \mathbb{B}$$

Formalizing this would verify the connection between pGCL expectations and probability theory expectations.
Loop Rules

- Practical program analysis tools need robust ways of reasoning about programs with loops.
- The usual slogan

\[
\text{total correctness} = \text{partial correctness} + \text{termination}
\]

doesn’t hold for (this formalization of) pGCL!
- Counterexample verified in HOL4:

\[
\vdash \text{wlp} (\text{While} (n = 0) (n := \langle 0, 1 \rangle)) \ One \neq \ One
\]

- What is the best way of working around this?
Summary

- Formalized the theory of pGCL in higher-order logic.
- Created an automatic tool for deriving sufficient conditions for partial correctness.
  - Useful product of mechanizing a program semantics.
- There’s still much to be done formalizing the theory and implementing practical program analysis tools.
Related Work

- Formal methods for probabilistic programs:
  - 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.