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

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

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

- 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.

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 $P' \implies P$.
- 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 $\alpha \to [0, +\infty]$.
- 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:

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

- Expectations form a lattice, so expectation transformers can be up_continuous, have least and greatest fixed points, etc.

Healthiness Condition

- The definition of healthiness for expectation transformers is analogous to healthiness of predicate transfomers in standard GCL:

$$\text{healthy } t \;\equiv\; \text{feasible } t \wedge \text{sublinear } t \wedge \text{up\_continuous } t$$

  where

$$\text{sublinear } t \;\equiv\; \text{scaling } t \wedge \text{subadditive } t \wedge \text{subtractive } t$$

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

States

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

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

- For convenience, define a state update function:

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

pGCL Commands

Model pGCL commands with a HOL datatype:

$$
\begin{array}{lcl}
\text{command} & \equiv & \text{Abort} \\
 & \mid & \text{Skip} \\
 & \mid & \text{Assign of string} \times (\text{state} \to \mathbb{Z}) \\
 & \mid & \text{Seq of command} \times \text{command} \\
 & \mid & \text{Nondet of command} \times \text{command} \\
 & \mid & \text{Prob of } (\text{state} \to [0,1]) \times \text{command} \times \text{command} \\
 & \mid & \text{While of } (\text{state} \to \mathbb{B}) \times \text{command}
\end{array}
$$

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

Derived Commands

Define all other commands as syntactic sugar:

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

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

Weakest Preconditions

Define weakest preconditions (wp) directly on commands:

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

Commands are Healthy

- 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:

$$\begin{array}{ll} \vdash & \forall t, e_1, e_2. \\ & \quad \text{healthy}\ t\ \wedge\ \text{bounded}\ t\ \wedge\ e_2 \sqsubseteq e_1 \ \Longrightarrow \\ & \quad t\ (\lambda s.\ e_1\ s - e_2\ s) \sqsubseteq \lambda s.\ t\ e_1\ s - t\ e_2\ s \end{array}$$

Example: Monty Hall

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

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:

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

- 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

Weakest liberal preconditions (wlp) model partial correctness.

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

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

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

- These correspond to the total and partial Hoare triples

$$[\bot]\ \text{loop}\ [post] \qquad \{\top\}\ \text{loop}\ \{post\}$$

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: $(\lambda s.\ 0.95) \sqsubseteq \mathrm{wlp\ prog\ (if\ ok\ then\ 1\ else\ 0)}$
- Can think of this as the query $P \sqsubseteq \mathrm{wlp}\ c\ q$.
- Idea: use a Prolog interpreter to solve for the variable P.

Calculating wlp: Rules

Simple rules:

- $\text{Infty} \sqsubseteq \text{wlp Abort } Q$
- $Q \sqsubseteq \text{wlp Skip } Q$
- $R \sqsubseteq \text{wlp } C_2 \; Q \;\; \wedge \;\; P \sqsubseteq \text{wlp } C_1 \; R$
  $\Longrightarrow$
  $P \sqsubseteq \text{wlp } (\text{Seq } C_1 \; C_2) \; Q$

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

Calculating wlp: While Loops

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

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

The desired postcondition representing a unique winner of the election is

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

Rabin's Mutual Exclusion Algorithm

- 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:

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

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

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

where

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

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

Rabin's Mutual Exclusion Algorithm

The verification proceeded as follows:

1. Annotate the program to create the goal:

$$pre \sqsubseteq \text{wlp annotated\_rabin } post$$

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.

```
|- 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))
```

Relational Semantics

- 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:

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

- 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:

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

- 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:
  - Formalizing Dijkstra, Harrison, 1998.
  - Mechanizing program logics in higher order logic, Gordon, 1989.