Prof. Joost-Pieter Katoen
Philipp Schroer

Deductive Verification of Probabilistic Programs with Caesar

— Lab Sessions 1 & 2 —

We encourage you to look at Caesar's documentation[1]. In particular, the HeyVL language documentation[2] for syntax. If you have any questions, problems, or ideas on Caesar, simply send an email to Philipp (phisch@cs.rwth-aachen.de).

We would also appreciate it if you **send your solutions to us via email**. This allows us to evaluate how far you came. Feel free to add feedback!

Send the email to Philipp (phisch@cs.rwth-aachen.de). Thanks!

Contents

[1] https://www.caesarverifier.org/docs/
[2] https://www.caesarverifier.org/docs/heyvl/

1 Setting up Caesar

Install Caesar on your computer. Installation instructions are here:

`https://www.caesarverifier.org/docs/getting-started/installation`

We highly recommend simply installing the Visual Studio Code extension. Do the walkthrough in the beginning (see installation instructions) to make sure you've got Caesar installed properly. Caesar can also be used on the command-line, but you may be missing some features.

Particularly relevant documentation pages are:

- HeyVL Procedures
- HeyVL Statements
- HeyVL Expressions

2 Loop-Free Boolean Verification

As a first step, we will verify simple loop-free *Boolean* programs with Caesar.

2.1 Verifying Minimum

Write a program in HeyVL that computes a minimum of two numbers using an if-then-else statement. Have Caesar verify that it actually computes the minimum.

Here is a template for you to fill in:

```
@wp proc minimum(x: UInt, y: UInt) -> (res: UInt)
    pre [true]
    post [res == x \cap y]
{
    ...
}
```

Syntax: The `cap` operator is the binary minimum. $[b]$ is the *Iverson bracket* that evaluates to 1 if b is `true` and to 0 otherwise.

2.2 Verifying Swap

Write a HeyVL program that swaps two numbers using an if-then-else statement and verify that it actually does swap the numbers.

2.3 Procedures and Coprocedures

Verification Conditions for Procedures and Coprocedures Consider some HeyVL code S. What happens if you wrap it in a `proc` or `coproc`? Whereas `proc`s verify *lower bounds*, `coproc`s verify *upper bounds* on the pre f with respect to the post g:

- `proc` verifies if $\forall \sigma \in \text{States}.\quad f(\sigma) \leq \text{wp}[\![S]\!](g)(\sigma)$.
- `coproc` verifies if $\forall \sigma \in \text{States}.\quad f(\sigma) \geq \text{wp}[\![S]\!](g)(\sigma)$.

Let's see the difference in action!

Consider the simple program

```
res = x / y
```

with inputs x, y of type `UInt` and output *res* of type `UInt`.

2.3.1 Verifying with a Procedure

Create a `proc` that verifies

$$\texttt{[y != 0]} \quad \sqsubseteq \quad \text{wp}[\![\texttt{res = x / y}]\!](\texttt{[res * y == x]})\,.$$

2.3.2 Verifying with a Coprocedure

Instead of a `proc`, try to use a `coproc` to show

$$\texttt{[y != 0]} \quad \sqsupseteq \quad \texttt{wp}[\![\texttt{res = x / y}]\!](\texttt{[res * y == x]})\,.$$

What do you observe? Why? Use Caesar's counterexamples. Can you fix the problem?

2.4 Verifying Exact Semantics

Determine the exact wp semantics of the division program. *Prove* using Caesar that you found the exact semantics.

3 Probabilistic Verification: Bounds on Probabilities and Distributions

Distribution Expressions Caesar has *distribution expressions*[3] to sample from distributions in assignments. For example,

```
var prob_choice:  Bool = flip(0.75)
```

chooses `true` with probability 0.75 and `false` with probability 0.25.

Write a program that returns input x with probability 0.3 and input y with probability 0.7. Store the return value in an output variable called *res*. Use type `UInt` for all variables.

3.1 Lower Bounds

In different `proc`s,

1. Verify that x is returned with probability at least 0.3.
2. Verify that y is returned with probability at least 0.7.
3. Verify using a single `proc` that (1) *and* (2) hold.
   Hint: Use an additional input variable in the specification.

3.2 Upper Bounds

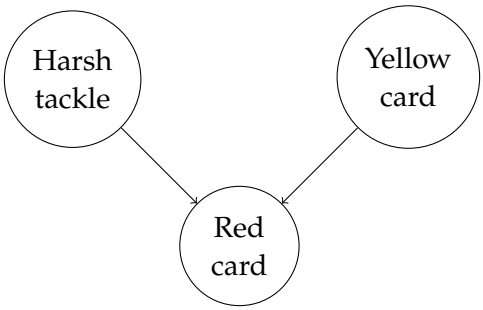
Try the above three tasks with `coproc`s. Before you try: what do you expect as results?

[3] https://www.caesarverifier.org/docs/stdlib/distributions

4 Probabilistic Verification: Bayesian Networks

4.1 Soccer Prediction

Consider the following Bayesian network modeling the probability that a football referee will show a red card (R), taking into account whether or not the player in question has tackled harshly (H) and if they have already seen a yellow card (Y):[4]



The (conditional) probability tables are given as follows:

H = 0	H = 1
0.8	0.2

Y = 0	Y = 1
0.9	0.1

	R = 0	R = 1
H = 0, Y = 0	0.99	0.01
H = 0, Y = 1	0.8	0.2
H = 1, Y = 0	0.6	0.4
H = 1, Y = 1	0.03	0.97

4.1.1 Writing the Program

Write a HeyVL `proc` that returns values from the joint distribution of the given Bayesian network.

Hint: Use HeyVL's `assert` $?(b)$ statement to encode the `observe` statement. It holds that

$$\mathsf{wp}[\![\texttt{observe } b]\!](f) \quad = \quad \mathsf{wp}[\![\texttt{assert } ?(b)]\!](f) = \lambda\sigma. \begin{cases} f(\sigma), & \text{if } b(\sigma) = \texttt{true} \\ 0, & \text{otherwise} \end{cases}.$$

4.1.2 Conditional Probabilities

Use Caesar's `proc`s and `coproc`s to calculate the probability of a harsh tackle given $Y = 1$ and $R = 1$.

[4] These values do not reflect actual UEFA Euro 2024 probabilities.

5 Probabilistic Verification: Coin Flips in a Loop

5.1 Implementation

Implement the *geometric loop* program: in a loop, flip a fair coin. If we get heads, then stop the loop. Otherwise, increment a counter c (`UInt`) which you return as an output. The counter c starts at 0.

(Caesar will not verify the program without a proof rule on the loop. That's what we'll do in the next task.)

5.2 Loop Unrolling for Lower Bounds

For $n = \{0, 1, 2, 3, 4, 5\}$ *loop unrollings* (see below), compute the expected value of c. Insert an appropriate `pre` to obtain the expected values of c after each loop iteration from Caesar. Explain the results.

Loop Unrolling Use the `@unroll` annotation on a loop to approximate wp-semantics:

```
@unroll(k, 0) while b { ... }
```

For different concrete values of k, this evaluates to the n-th fixpoint iteration in the loop semantics:

$$\texttt{wp}[\![\ \texttt{@unroll(}k\texttt{, 0) while b \{ ... \}}\ ]\!](f) \quad =: \quad \Phi_f^n(0) \quad \sqsubseteq \quad \underbrace{\texttt{wp}[\![\ \texttt{while b \{ ... \}}\ ]\!](f)}_{\operatorname{lfp} X.\ \Phi_f(X)}.$$

5.3 Guess the Expected Value

Based on your results from the unfoldings, guess the expected value after an unbounded number of iterations, i.e. guess

$$\lim_{n\to\infty} \Phi_f^n(0) \quad = \quad \operatorname{lfp} X.\ \Phi_f(X)\ .$$

5.4 Park Induction for Upper Bounds

Now *verify* that your guess is an *upper bound* to the expected value of c using Caesar. Use a `coproc` and *Park induction*; find an appropriate invariant I.

Park Induction Use the `@invariant` annotation to apply Park induction. This requires an *inductive invariant* I.

```
@invariant(I) while b { ... }
```

I is *inductive* w.r.t. post f if $\Phi_f(I) \sqsubseteq I$ holds.

If I is inductive for post f, then by Park's lemma we know I is an upper bound:[5]

$$\texttt{wp}[\![\ \texttt{while b \{ ... \}}\ ]\!](f) \quad \sqsubseteq \quad I \quad := \quad \texttt{wp}[\![\ \texttt{@invariant(}I\texttt{) while b \{ ... \}}\ ]\!](f)\,.$$

Tips:

- Use the *Caesar: Explain Verification Conditions* command in VS Code to understand the calculations inside the loop body. After running the command and if you leave empty lines inside the loop, then Caesar will show the computations for the expected value of I.
- Construct the invariant I from the cases where the loop a) runs, b) does not run.

5.5 Unbounded Coin Flips Times Two

Now adjust your program so that it runs the geometric loop *two* times. Duplicate the loop. Adjust the `pre` and the invariants accordingly.

5.6 Parametric Coin Flips in a Loop

Up until now the probability to continue always was 0.5. Adjust the solution for Section 5.4 by changing the probability to be a variable p, where it is a new parameter of type `UReal`.

Hints:

- Make sure that the `pre` also contains a constraint so that $p \in [0,1]$. The semantics of `flip(`p`)` is *undefined* otherwise.
- Strengthen the constraint to $p \in [0,1)$ to make verification easier (avoiding division by zero).
- You can try to use loop unrolling to get a hint of what the expected value of c should be.

5.7 Duelling Cowboys

Implement and verify the *Duelling Cowboys* example from the lecture. Let cowboy A start the duel. Use Park induction to prove an upper bound on the probability that cowboy A wins.

Hint: You have some freedom in how you encode the problem in HeyVL. But do think about possible required preconditions on the inputs.

[5] If I can *not* be verified as an inductive invariant, the proof rule will have weakest pre evaluate to 0, i.e. $\texttt{wp}[\![\ \texttt{@invariant(}I\texttt{) while b \{ ... \}}\ ]\!](f) = 0$.

6 Probabilistic Verification: Expected Runtimes

Expected Runtimes and Reward To encode the $+1$ in the ert semantics in HeyVL, we use the `reward 1` statement. The semantics is defined as $\mathsf{wp}[\![\texttt{reward}\ e]\!](f) = f + e$. In this section, we consider a simplified version of ert where *we only count loop iterations*. Thus, you only need to insert one `reward 1` statement in the loop body.

6.1 Positive Almost-Sure Termination for the Geometric Loop

We return back to the example from Section 5.4. Prove that the geometric loop is *positively almost-surely terminating* (PAST), i.e. that $\forall \sigma \in \mathsf{States}.\ \mathsf{ert}[\![\mathsf{geo}]\!](0)(\sigma) < \infty$.

Extra challenge: Verify PAST for a parametric version of the geometric loop (c.f. Section 5.6).

6.2 Runtime Counters: Easy to Misuse!

Why couldn't we simply prove that the expected value of c is finite? Modulo off-by-one it tracks the number of loop iterations.

Hint: What happens when $p = 1$ holds?

7 Probabilistic Verification: Almost-Sure Termination

7.1 The Symmetric Random Walk

Consider the following pGCL loop, which implements a *symmetric random walk*:

$$\texttt{while}\ x > 0\ \{\{x := x - 1\}\ [0.5]\ \{x := x + 1\}\}$$

Prove using Caesar that it is almost-surely terminating using the proof rule for AST. Choose a Boolean invariant I, a variant V, a probability p, and a decrease d and prove the following three conditions by writing corresponding `proc`/`coproc`s in HeyVL:

1. *I is a* wp*-subinvariant w.r.t post I.* You can prove that the statement

   $$\texttt{if}\ x > 0\ \{\{x := x - 1\}\ [0.5]\ \{x := x + 1\}\}\ \texttt{else}\ \{\texttt{skip}\}$$

   verifies with in a `proc` respect to to `pre` $[I]$ and `post` $[I]$.
2. *I is a* wp*-superinvariant w.r.t post I.* Analogous to the above.
3. *V satisfies the progress condition:* Here's a schemtic on how to write the condition in HeyVL:

```
@wp
proc progress_condition(init_vars: ...) -> (vars: ...)
    pre [I(init_vars)] * [G(init_vars)] * prob(V(init_vars))
    post [V(vars) <= V(init_vars) - decrease(V(init_vars))]
{
    vars = init_vars // set current state to input values
    Body
}
```

8 Probabilistic Verification: Optional Stopping Theorem

8.1 Lower Bounds for the Geometric Loop

We return back to the example from Section 5.4. However, we now want to establish *lower bounds* on the expected value of c.

Use the *Optional Stopping Theorem* for weakest pre-expectations to prove that 1 is also a lower bound on the expected value of c.

Verify `proc`s and `coproc` to discharge the separate proof obligations.

1. *I is a* wp*-subinvariant:* To encode a check of $I \sqsubseteq \Phi_f(I)$, (ab)use the `@unroll` annotation and write `@unroll(1, I) while b { ... }` to encode $\Phi_f(I)$.
2. (You have shown that the geometric loop program is PAST in section 6.1.)
3. Conditional difference boundedness:
   - Absolute value: To encode $|I - I(\sigma)|$, use $\text{ite}(I \geq I(\sigma),\ I - I(\sigma),\ I(\sigma) - I)$.

9 Probabilistic Verification: Reasoning with Axioms

9.1 Morris' Approximate Counting Algorithm

So far, we've only used expectations built using simple arithmetic operators and conditionals. However, many verification problems require exponentials. These are not built-in to Caesar, but can be added via *axioms*[6].

[6] Documentation: `https://www.caesarverifier.org/docs/heyvl/domains`.

Domains and Axioms For the following tasks, insert this piece of code at the top of your HeyVL file:

```
domain Exponentials {
    func pow2(exponent: UInt): UReal

    axiom pow2_base pow2(0) == 1
    axiom pow2_step forall exponent: UInt.
        pow2(exponent + 1) == 2 * pow2(exponent)

    axiom pow2_at_least_one forall exponent: UInt.
        pow2(exponent) >= 1
}
```

It declares a new *domain* named `Exponentials` with the new *uninterpreted function* `pow2` that takes an `UInt` parameter and returns an `UReal` value. Initially, Caesar knows nothing about `pow2`. The *axiom declarations* declare assumptions for Caesar about the `pow2` function. The first two define `pow2` inductively. The third axiom is a fact that could be easily proven by induction using the first two axioms, but that Caesar's underlying SMT solver needs the third axiom as assistance.

Note that Caesar will be unable to provide counterexamples with these axiom definitions. This is because the SMT solver is no longer complete on this fragment.[7]

Morris' Approximate Counting Algorithm Counting views, likes, or clicks on the modern web necessitates a smarter approach to counting than locking a centralized counter on every access. The *approximate counting algorithm* by Morris[8] avoids a lot of locking by only increasing a counter *probabilistically*.

We want to verify this algorithm and model it by a loop that runs n iterations (for n views/likes/etc.). We keep a counter d, initialized at 1. In every iteration, it is incremented with probability $1/d$. Verify that the expected value of 2^d after termination is at most $n+1$, showing that the implementation indeed correctly over-approximates the count n. Use Park induction.

9.2 Coupon Collector's Problem

The *Coupon Collector's Problem* is a classic problem in probability theory. The problem is as follows: you have n different coupons, and you want to collect all of them. You can collect a coupon by sampling uniformly at random from the n coupons. The question is: how many samples do you need to collect all n coupons?

Write a HeyVL program that implements the coupon collection and verify an upper bound on the expected number of samples needed to collect all n coupons.[9]

Use a loop that runs n iterations, iterating a value i starting at n down to 1. Each iteration i represents the state of collection with i coupons already collected. To model the expected number of samples needed to collect the remaining coupons, write `reward` n/i.[10]

[7] See Caesar's debugging documentation page for more information on this problem and how to handle such problems.
[8] Morris, R. *Counting large numbers of events in small registers*. Communications of the ACM 21, 10 (1978), 840–842
[9] We won't spoil the solution here, but if you're struggling, you can look up the solution on Wikipedia.
[10] One could also model the nested sampling as a nested loop, but we stick to the simpler approach here.

Harmonic Numbers in HeyVL Harmonic numbers will show up in the solution. The following axiomatization for harmonic numbers should suffice for the task:

```
domain Harmonics {
    func harmonic(n: UInt): UReal

    axiom harmonic_base harmonic(0) == 0
    axiom harmonic_step forall n: UInt. harmonic(n + 1) == 1/(n+1) + harmonic(n)
}
```