Prof. Joost-Pieter Katoen
Philipp Schroer

Deductive Verification of Probabilistic Programs with Caesar

— Lab Session 1 —

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

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

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

2.3.2 Verifying with a Coprocedure

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

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

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?

4 Probabilistic Verification: Coin Flips in a Loop

4.1 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.)

4.2 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.

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

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

$$\texttt{@unroll(}k\texttt{, 0) while b \{ ... \}}$$

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

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

4.3 Guess the Expected Value

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

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

4.4 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.

$$\texttt{@invariant(}I\texttt{) while b \{ ... \}}$$

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:[4]

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

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.

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

4.5 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.

4.6 Parametric Coin Flips in a Loop

Up until now the probability to continue always was 0.5. Adjust the solution for Section 4.4 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.

4.7 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.