Caesar: A Deductive Verifier for Probabilistic Programs

Dafny 2024

Philipp Schröer[1], Kevin Batz[1], Benjamin Kaminski[2], Joost-Pieter Katoen[1], Christoph Matheja[3]

RWTH Aachen University[1], Saarland University and University College London[2], Technical University of Denmark[3]

The *geometric loop* program C_{geo}:

$$
\begin{array}{l}
c := 0;\ run := \texttt{true}; \\
\texttt{while}\,(run)\,\{ \\
\quad \{\, run := \texttt{false} \,\}\,[0.5]\,\{\, c := c + 1 \,\} \\
\}
\end{array}
$$

The *geometric loop* program C_{geo}:

$$
\begin{aligned}
&c := 0;\ run := \texttt{true};\\
&\texttt{while}\,(run)\,\{\\
&\quad \underbrace{\{\, run := \texttt{false} \,\}\,[0.5]\,\{\, c := c + 1 \,\}}_{\textit{probabilistic choice}}\\
&\}
\end{aligned}
$$

The *geometric loop* program C_{geo}:

$$
\begin{array}{l}
c := 0;\ run := \texttt{true};\\
\texttt{while}\ (run)\ \{\\
\quad \{\ run := \texttt{false}\ \}\ [0.5]\ \{\ c := c + 1\ \}\\
\}
\end{array}
$$

1. Probability of final states terminating with $c = 1$?

0.25

The *geometric loop* program C_{geo}:

c := 0; *run* := true;
while (*run*) {
  { *run* := false } [0.5] { c := c + 1 }
}

1. Probability of final states terminating with $c = 1$?

   0.25

2. Expected counter c after termination?

   $0.5 \cdot 0 + 0.25 \cdot 1 + 0.125 \cdot 2 + \ldots = 1.0$

The *geometric loop* program C_{geo}:

```
c := 0; run := true;
while (run) {
  { run := false } [0.5] { c := c + 1 }
}
```

1. Probability of final states terminating with $c = 1$?

   0.25

2. Expected counter c after termination?

   $0.5 \cdot 0 + 0.25 \cdot 1 + 0.125 \cdot 2 + \ldots = 1.0$

3. Probability of termination?

   1.0 (“almost-surely terminating”)

The *geometric loop* program C_{geo}:

$$
\begin{array}{l}
c := 0;\ run := \mathtt{true};\\
\mathtt{while}\ (run)\ \{\\
\quad \{\, run := \mathtt{false} \,\}\ [0.5]\ \{\, c := c + 1 \,\}\\
\}
\end{array}
$$

1. Probability of final states terminating with $c = 1$?

   0.25

2. Expected counter c after termination?

   $0.5 \cdot 0 + 0.25 \cdot 1 + 0.125 \cdot 2 + \ldots = 1.0$

3. Probability of termination?

   1.0 ("almost-surely terminating")

4. Expected runtime?

   2.0 ("positively almost-surely terminating")

Our Verification Infrastructure for Probabilistic Programs

lower and upper bounds on expected values

(amortized) expected run-times

almost-sure termination

positive almost-sure termination

Our Verification Infrastructure for Probabilistic Programs

lower and upper bounds on expected values
(amortized) expected run-times
almost-sure termination
positive almost-sure termination
probabilistic sensitivity
conditional expected values
expected resources

Park induction
k-induction
martingales

Our Verification Infrastructure for Probabilistic Programs

lower and upper bounds on expected values
(amortized) expected run-times
almost-sure termination
positive almost-sure termination
probabilistic sensitivity
conditional expected values
expected resources

Park induction
k-induction
martingales

Our Motivation: How to automate verification of the above?

Our Verification Infrastructure for Probabilistic Programs

lower and upper bounds on expected values *... more?*
(amortized) expected run-times almost-sure termination
positive almost-sure termination probabilistic sensitivity
conditional expected values expected resources

Park induction
k-induction
martingales
... more?

Our Motivation: How to automate verification of the above?
How to make the automation extensible?

Our Verification Infrastructure for Probabilistic Programs

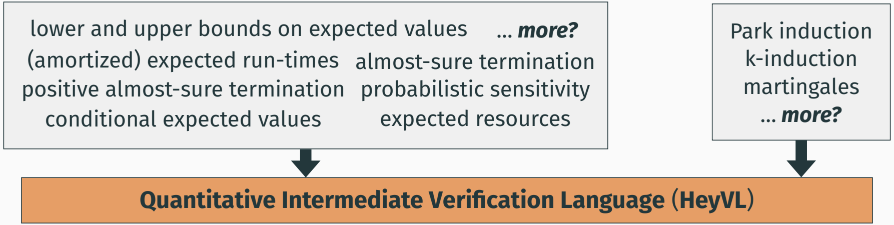
lower and upper bounds on expected values ... ***more?***
(amortized) expected run-times
almost-sure termination
positive almost-sure termination
probabilistic sensitivity
conditional expected values
expected resources

Park induction
k-induction
martingales
... ***more?***

Our Motivation: How to automate verification of the above?
How to make the automation extensible?

⇒ An intermediate language for the verification of probabilistic programs.
"Build the probabilistic version of *Boogie/Viper*"

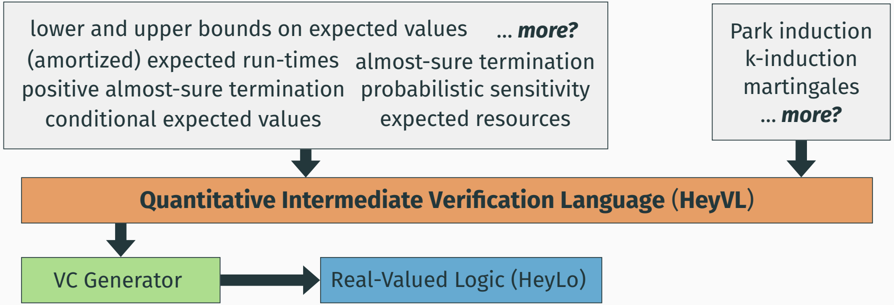
Our Verification Infrastructure for Probabilistic Programs



We present:

- A novel *intermediate language* to verify probabilistic programs (*HeyVL*),

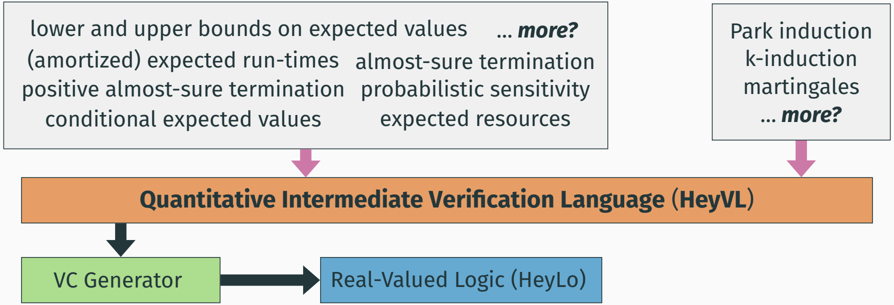
Our Verification Infrastructure for Probabilistic Programs



We present:

- A novel *intermediate language* to verify probabilistic programs (*HeyVL*),
- A new *assertion language* for quantities (*HeyLo*),

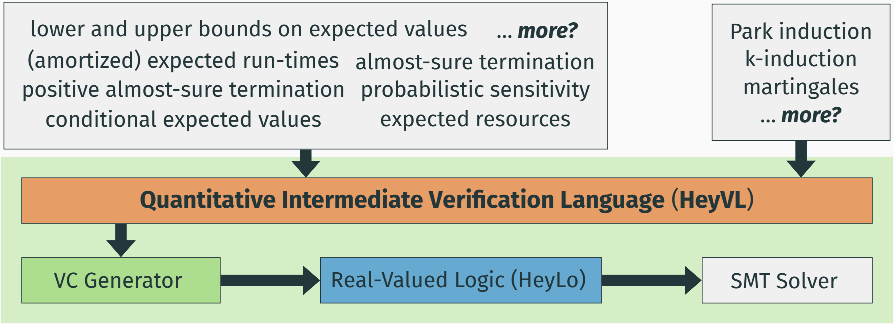
Our Verification Infrastructure for Probabilistic Programs



We present:

- A novel *intermediate language* to verify probabilistic programs (*HeyVL*),
- A new *assertion language* for quantities (*HeyLo*),
- Encodings of *properties* and *proof rules* into HeyVL,

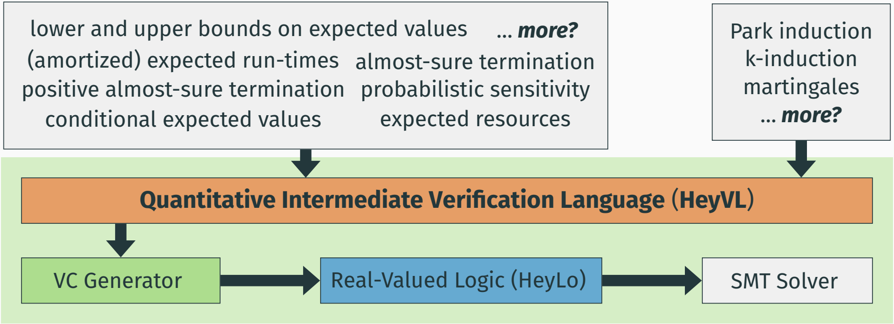
Our Verification Infrastructure for Probabilistic Programs



We present:

- A novel *intermediate language* to verify probabilistic programs (*HeyVL*),
- A new *assertion language* for quantities (*HeyLo*),
- Encodings of *properties* and *proof rules* into HeyVL,
- and an implementation called *Caesar*.

Our Verification Infrastructure for Probabilistic Programs



We present:

- A novel *intermediate language* to verify probabilistic programs (*HeyVL*),
- A new *assertion language* for quantities (*HeyLo*),
- Encodings of *properties* and *proof rules* into HeyVL,
- and an implementation called *Caesar*.

Classical Programs

Predicates

Specification: $\{P\}\ S\ \{\downarrow Q\}$

$P, Q : \Sigma \to \mathbb{B}$

Classical Programs

Predicates

Specification:

$\{P\}\ S\ \{\downarrow Q\}$

$P, Q : \Sigma \to \mathbb{B}$

Meaning:

P is a *subset* of
the states terminating in Q.

Classical Programs

Predicates

Specification: $\{P\}\ S\ \{\downarrow Q\}$

$P, Q : \Sigma \to \mathbb{B}$

Meaning: P is a *subset* of the states terminating in Q.

Deductively: $P \subseteq \mathrm{wp}_{\mathbb{P}}[\![S]\!](Q)$

"weakest preconditions"

Classical Programs

Predicates

Specification: $\{P\}\ S\ \{\downarrow Q\}$

$P, Q : \Sigma \to \mathbb{B}$

Meaning: P is a *subset* of the states terminating in Q.

Deductively: $P \subseteq \mathrm{wp}_{\mathbb{P}}[\![S]\!](Q)$

"weakest preconditions"

IVL: `assume` P`;` encode$[\![S]\!]$`;` `assert` Q

	Classical Programs	**Probabilistic Programs**
	Predicates	*Expectations*
Specification:	$\{P\}\ S\ \{\downarrow Q\}$ $P, Q : \Sigma \to \mathbb{B}$	$\{f\}_{\leq}\ S\ \{\downarrow g\}$ $f, g : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$
Meaning:	P is a *subset* of the states terminating in Q.	f is a *lower bound* to the expected value of g.
Deductively:	$P \subseteq \mathrm{wp}_{\mathbb{P}}[\![S]\!](Q)$ "weakest preconditions"	
IVL:	`assume` P`;` encode$[\![S]\!]$`;` `assert` Q	

	Classical Programs	**Probabilistic Programs**
	Predicates	*Expectations*
Specification:	$\{P\}\ S\ \{\downarrow Q\}$ $P, Q : \Sigma \to \mathbb{B}$	$\{f\}_{\leq}\ S\ \{\downarrow g\}$ $f, g : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$
Meaning:	P is a *subset* of the states terminating in Q.	f is a *lower bound* to the expected value of g.
Deductively:	$P \subseteq \mathrm{wp}_{\mathbb{P}}[\![S]\!](Q)$ "weakest preconditions"	$f \leq \mathrm{wp}_{\mathbb{E}}[\![S]\!](g)$ "weakest preexpectations"
IVL:	`assume` P`;` encode$[\![S]\!]$`;` `assert` Q	

	Classical Programs	**Probabilistic Programs**
	Predicates	*Expectations*
Specification:	$\{P\}\ S\ \{\downarrow Q\}$ $P, Q : \Sigma \to \mathbb{B}$	$\{f\}_{\leq}\ S\ \{\downarrow g\}$ $f, g : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$
Meaning:	P is a *subset* of the states terminating in Q.	f is a *lower bound* to the expected value of g.
Deductively:	$P \subseteq \mathrm{wp}_{\mathbb{P}}[\![S]\!](Q)$ "weakest preconditions"	$f \leq \mathrm{wp}_{\mathbb{E}}[\![S]\!](g)$ "weakest preexpectations"
IVL:	$\texttt{assume}\ P\texttt{;}\ \mathrm{encode}[\![S]\!]\texttt{;}\ \texttt{assert}\ Q$	$\underbrace{\texttt{assume}\ f\texttt{;}\ \mathrm{encode}[\![S]\!]\texttt{;}\ \texttt{assert}\ g}_{\text{HeyVL}}$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics?

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \rightarrow g \quad = \quad ???$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics?

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \to g \quad = \quad ???$$

Properties:

1. *Heyting algebra:*

$$f \leq g \to h$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics?

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \to g \quad = \quad ???$$

Properties:

1. *Heyting algebra:*

$$f \leq g \to h \quad \text{iff} \quad f \sqcap g \leq h$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics?

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \to g \quad = \quad ???$$

Properties:

1. *Heyting algebra:*

$$f \le g \to h \quad \text{iff} \quad f \sqcap g \le h$$

2. *Deduction theorem:*

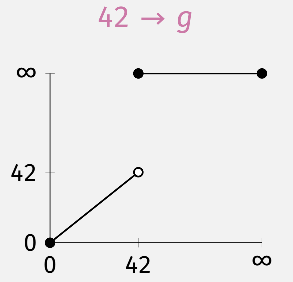
$$f \le g \quad \text{iff} \quad f \to g \equiv \infty$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics:

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \to g \quad = \quad ???$$



"Assume $g \geq 42$ to be true"

Properties:

1. *Heyting algebra:*

$$f \leq g \to h \quad \text{iff} \quad f \sqcap g \leq h$$

2. *Deduction theorem:*

$$f \leq g \quad \text{iff} \quad f \to g \equiv \infty$$

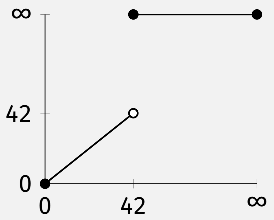
Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics:

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \rightarrow g \quad = \quad \lambda\sigma.\ \begin{cases} \infty, & \text{if } f(\sigma) \leq g(\sigma) \\ g(\sigma), & \text{else} \end{cases}$$

$42 \rightarrow g$



"Assume $g \geq 42$ to be true"

Properties:

1. *Heyting algebra:*

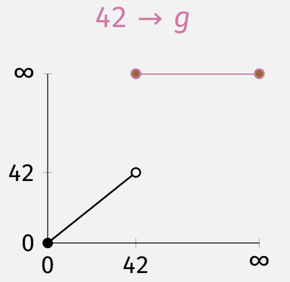
$$f \leq g \rightarrow h \quad \text{iff} \quad f \sqcap g \leq h$$

2. *Deduction theorem:*

$$f \leq g \quad \text{iff} \quad f \rightarrow g \equiv \infty$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$



"Assume $g \geq 42$ to be true"

Assume semantics:

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \to g \quad = \quad \lambda\sigma.\ \begin{cases} \infty, & \text{if } f(\sigma) \leq g(\sigma) \\ g(\sigma), & \text{else} \end{cases}$$

Properties:

1. *Heyting algebra:*

$$f \leq g \to h \quad \text{iff} \quad f \sqcap g \leq h$$

2. *Deduction theorem:*

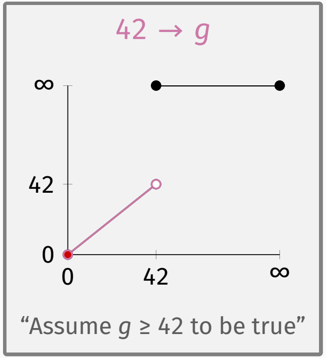
$$f \leq g \quad \text{iff} \quad f \to g \equiv \infty$$

Assert semantics with minimum:

$$[\![\texttt{assert}\ f]\!](g) \quad = \quad f \sqcap g$$

Assume semantics:

$$[\![\texttt{assume}\ f]\!](g) \quad = \quad f \rightarrow g \quad = \quad \lambda\sigma.\ \begin{cases} \infty, & \text{if } f(\sigma) \leq g(\sigma) \\ g(\sigma), & \text{else} \end{cases}$$



Properties:

1. *Heyting algebra:*

$$f \leq g \rightarrow h \quad \text{iff} \quad f \sqcap g \leq h$$

2. *Deduction theorem:*

$$f \leq g \quad \text{iff} \quad f \rightarrow g \equiv \infty$$

```
assume c + 0.5
{ run := false } [0.5] { c := c + 1 }
assert c
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
assume c + 0.5

{ run := false } [0.5] { c := c + 1 }

assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
assume c + 0.5

{ run := false } [0.5] { c := c + 1 }

// c ⊓ ∞
assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
assume c + 0.5
```

```
{ run := false } [0.5] { c := c + 1 }
// c
// c ⊓ ∞
assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
assume c + 0.5

// 0.5 · c + 0.5 · (c + 1)
{ run := false } [0.5] { c := c + 1 }
// c
// c ⊓ ∞
assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
assume c + 0.5
// c + 0.5
// 0.5 · c + 0.5 · (c + 1)
{ run := false } [0.5] { c := c + 1 }
// c
// c ⊓ ∞
assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
// (c + 0.5) → (c + 0.5)
assume c + 0.5
// c + 0.5
// 0.5 · c + 0.5 · (c + 1)
{ run := false } [0.5] { c := c + 1 }
// c
// c ⊓ ∞
assert c
// ∞
```

A HeyVL program S *verifies* iff $[\![S]\!](\infty) \equiv \infty$.

```
// ∞
// (c + 0.5) → (c + 0.5)
assume c + 0.5
// c + 0.5
// 0.5 · c + 0.5 · (c + 1)
{ run := false } [0.5] { c := c + 1 }
// c
// c ⊓ ∞
assert c
// ∞
```

The HeyVL program verifies, therefore

$$\{c + 0.5\}_{\leq}\ S'\ \{c\}$$

where $S' = \{\, run := \texttt{false} \,\}\, [0.5]\, \{\, c := c + 1 \,\}$.

HeyVL

Expectations

lower bounds

S	$[\![S]\!](g)$
`assert` f	$f \sqcap g$
`assume` f	$f \to g$
`havoc` x	$\inf x\colon g$

Omitted: `validate`, `reward`, branching

Classical IVL

Predicates

lower bounds

S	$[\![S]\!](Q)$
`assert` P	$P \wedge Q$
`assume` P	$P \Rightarrow Q$
`havoc` x	$\forall x\colon Q$

HeyVL

Expectations

lower bounds

S	$[\![S]\!](g)$
`assert` f	$f \sqcap g$
`assume` f	$f \rightarrow g$
`havoc` x	$\inf x\colon g$

Omitted: `validate`, `reward`, branching

- **HeyVL** generalizes classical IVLs.

Classical IVL

Predicates

lower bounds

S	$[\![S]\!](Q)$
`assert` P	$P \wedge Q$
`assume` P	$P \Rightarrow Q$
`havoc` x	$\forall x \colon Q$

HeyVL

Expectations

lower bounds

S	$[\![S]\!](g)$
`assert` f	$f \sqcap g$
`assume` f	$f \rightarrow g$
`havoc` x	$\inf x \colon g$

upper bounds

S	$[\![S]\!](g)$
`coassert` f	$f \sqcup g$
`coassume` f	$f \leftarrow g$
`cohavoc` x	$\sup x \colon g$

Omitted: `validate`, `reward`, branching

- **HeyVL** generalizes classical IVLs.
- **HeyVL** has dual verification statements for *upper bounds reasoning*.

More than 40 examples:

- with 12 proof rules for loops,
- lower and upper bounds,
- procedures with recursion,
- user-defined data structures.

Case Studies

More than 40 examples:

- with 12 proof rules for loops,
- lower and upper bounds,
- procedures with recursion,
- user-defined data structures.

Problem	Verification Technique	Source
LPROB	wlp + Park induction	McIver and Morgan [2005]
	wlp + latticed k-induction	**(new?)**
UPROB	wlp + ω-invariants	Kaminski [2019]
LEXP	wp + ω-invariants	Kaminski [2019]
	wp + Optional Stopping Theorem	Hark et al. [2019]
UEXP	wp + Park induction	McIver and Morgan [2005]
	wp + latticed k-induction	Batz et al. [2021]
CEXP	conditional wp	Olmedo et al. [2018]
LERT	ert calculus + ω-invariants	Kaminski et al. [2016]
UERT	ert calculus + UEXP rules	Kaminski et al. [2016]
AST	parametric super-martingale rule	McIver et al. [2018]
PAST	program analysis with martingales	Chakarov and Sankaranarayanan [2013]
???	**more proof rules**	**you?**

To reason about `while` (b) $\{S\}$ loops, we can use user-provided *invariants*.

(Let $\vec{x}$ be the modified variables in the loop.)

To reason about `while` (b) $\{S\}$ loops, we can use user-provided *invariants*.

(Let $\vec{x}$ be the modified variables in the loop.)

Classical IVL

Predicates

Let $I \in \mathbb{P}$ be an invariant candidate.

```
assert I
havoc x⃗
assume I
if (b) {
  encode⟦S⟧
  assert I; assume false
}
```

To reason about `while` (b) $\{S\}$ loops, we can use user-provided *invariants*.

(Let $\vec{x}$ be the modified variables in the loop.)

Classical IVL

Predicates

Let $I \in \mathbb{P}$ be an invariant candidate.

```
assert I
havoc x⃗
assume I
if (b) {
  encode⟦S⟧
  assert I; assume false
}
```

HeyVL

Expectations

Let $I \in \mathbb{E}$ be an invariant candidate.

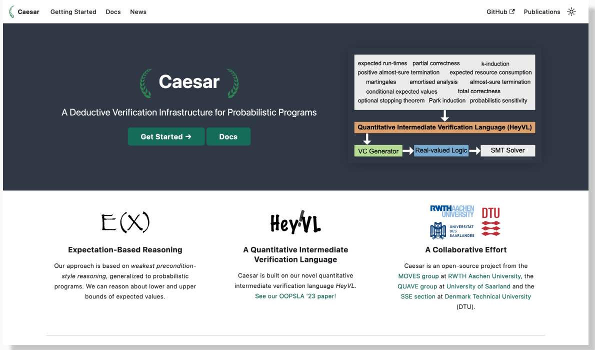
```
assert I
havoc x⃗
validate; assume I
if (b) {
  encode⟦S⟧
  assert I; assume 0
}
```

In our OOPSLA ’23 paper:

- *HeyVL:* An *intermediate language* to verify probabilistic programs,
- *HeyLo:* An *assertion language* to reason about quantities,
- Case studies of HeyVL encodings.

Online:

- The verifier Caesar
  - written in Rust, open source
- Language documentation
- Extended version of the paper



`www.caesarverifier.org`