Proof Rules for Probabilistic Loops

Seminar on Probabilistic Programs

David Korzeniewski

Lehrstuhl für Informatik II, RWTH Aachen

February 5, 2015

Outline

Outline

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: $\textbf{if}(cond)\{prog_1\}\textbf{else}\{prog_2\}$
- loops: $\textbf{while}(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: **if**$(cond)\{prog_1\}$**else**$\{prog_2\}$
- loops: **while**$(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: **if**$(cond)\{prog_1\}$**else**$\{prog_2\}$
- loops: **while**$(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: $\textbf{if}(cond)\{prog_1\}\textbf{else}\{prog_2\}$
- loops: $\textbf{while}(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: $\textbf{if}(cond)\{prog_1\}\textbf{else}\{prog_2\}$
- loops: $\textbf{while}(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: **if**$(cond)\{prog_1\}$**else**$\{prog_2\}$
- loops: **while**$(cond)\{prog\}$

Programs

Program Syntax

- atomic: **skip** | **abort** | $x := E$
- sequential execution: $prog_1 ; prog_2$
- probabilistic choice: $\{prog_1\}[p]\{prog_2\}$
- nondeterministic choice: $\{prog_1\}\square\{prog_2\}$
- branching: **if**$(cond)\{prog_1\}$**else**$\{prog_2\}$
- loops: **while**$(cond)\{prog\}$

State Space and Predicates

State Space

set of all valuations of the program variables

Standard Predicate

A *standard predicate* is a set of states

Predicates can be defined by *boolean expressions over program variables*.

State Space and Predicates

State Space

set of all valuations of the program variables

Standard Predicate

A *standard predicate* is a set of states

Predicates can be defined by *boolean expressions over program variables*.

Standard Predicate Transformers

Question: From which (initial) states are we guaranteed to establish a given *postcondition*?

Definition: The **weakest precondition** of the postcondition: $wp.prog.(post)$

A program is a *predicate transformer*, it transforms a predicate, the postcondition, into another predicate, the weakest precondition.

Standard Predicate Transformers

Question: From which (initial) states are we guaranteed to establish a given *postcondition*?
Definition: The **weakest precondition** of the postcondition: $wp.prog.(post)$

A program is a *predicate transformer*, it transforms a predicate, the postcondition, into another predicate, the weakest precondition.

Standard Predicate Transformers

Question: From which (initial) states are we guaranteed to establish a given *postcondition*?
Definition: The **weakest precondition** of the postcondition: $wp.prog.(post)$

A program is a *predicate transformer*, it transforms a predicate, the postcondition, into another predicate, the weakest precondition.

Weakest Precondition

- $wp.abort.g := false$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \wedge wp.prog'.g$

Weakest Precondition

- $wp.abort.g := false$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \wedge wp.prog'.g$

Weakest Precondition

- $wp.abort.g := false$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \wedge wp.prog'.g$

Weakest Precondition

- $wp.abort.g := false$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \wedge wp.prog'.g$

Weakest Precondition

- $wp.abort.g := false$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \wedge wp.prog'.g$

Example: Weakest Precondition

Faulty Addition

```
1 if(x < 0) {
2     result := x + y □ result := (x + y) * 2
3 }
4 else {
5     result := x + y
6 }
```

Postcondition: $result = x + y$

Weakest Precondition: $x \geq 0 \wedge x = -y$

Example: Weakest Precondition

Faulty Addition

```
1 if(x < 0) {
2     result := x + y □ result := (x + y) * 2
3 }
4 else {
5     result := x + y
6 }
```

Postcondition: $result = x + y$

Weakest Precondition: $x \geq 0 \wedge x = -y$

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

From Standard to Probabilistic Predicates

Requirements for probabilistic predicates:

- standard predicates must be usable in combination with probabilistic predicates
- we can get more precise information than just “good” or “bad” from probabilities
- idea of predicate transformer: predicates have to work with stepwise backward reasoning

We use **expected values**:

- *true* has expected worth 1, *false* 0
- expected value reflects the worth of a state
- expectations work well with the stepwise reasoning

Probabilistic Predicates: Expectations

Standard Case:
"good" and "bad" states

sets of states
or functions from states to $\{0, 1\}$

Probabilistic Case:
"probably/often good" or
"probably/often bad"
expected values
functions from states to $[0, 1]$

Probabilistic Predicate

A *probabilistic predicate* is a function from the state space to $[0, 1]$.

Probabilistic predicates can be defined by *real valued expressions over program variables*

Probabilistic Predicates: Expectations

Standard Case:	Probabilistic Case:
"good" and "bad" states	"probably/often good" or "probably/often bad"
sets of states	expected values
or functions from states to $\{0, 1\}$	functions from states to $[0, 1]$

Probabilistic Predicate

A *probabilistic predicate* is a function from the state space to $[0, 1]$.

Probabilistic predicates can be defined by *real valued expressions over program variables*

Weakest Preexpectation

Question: What is the expected value for *post* in the initial states?

Definition

The **weakest preexpectation** $wp.prog.(post)$ is the expected value of *post* in the worst case.

Weakest Preexpectation

Question: What is the expected value for *post* in the initial states?

Definition

The **weakest preexpectation** $wp.prog.(post)$ is the expected value of *post* in the worst case.

Example: Probabilistic Predicates

Probably Faulty Addition

```
1 if(x < 0) {
2     result := x + y [0.5] result := (x + y) * 2
3 }
4 else {
5     result := x + y [0.999] result := (x + y) * 2
6 }
```

post-predicate: $result = x + y$

weakest preexpectation:

- 1 if $x = -y$
- 0.999 if $x \geq 0 \wedge x \neq -y$
- 0.5 if $x < 0 \wedge x \neq -y$

Example: Probabilistic Predicates

Probably Faulty Addition

```
1 if(x < 0) {
2     result := x + y [0.5] result := (x + y) * 2
3 }
4 else {
5     result := x + y [0.999] result := (x + y) * 2
6 }
```

post-predicate: $result = x + y$

weakest preexpectation:

- 1 if $x = -y$
- 0.999 if $x \geq 0 \wedge x \neq -y$
- 0.5 if $x < 0 \wedge x \neq -y$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ "everywhere less or equal"
- $\Lleftarrow$ "everywhere greater or equal"
- $\equiv$ "everywhere equal"
- $\sqcap, \sqcup, \overline{g}$ as min, max, $1 - g$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ "everywhere less or equal"
- $\Lleftarrow$ "everywhere greater or equal"
- $\equiv$ "everywhere equal"
- $\sqcap, \sqcup, \overline{g}$ as min, max, $1 - g$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ “everywhere less or equal”
- $\Lleftarrow$ “everywhere greater or equal”
- $\equiv$ “everywhere equal”
- $\sqcap, \sqcup, \overline{g}$ as $\mathit{min}, \mathit{max}, 1 - g$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ “everywhere less or equal”
- $\Lleftarrow$ “everywhere greater or equal”
- $\equiv$ “everywhere equal”
- $\sqcap, \sqcup, \overline{g}$ as $\mathit{min}, \mathit{max}, 1 - g$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ “everywhere less or equal”
- $\Lleftarrow$ “everywhere greater or equal”
- $\equiv$ “everywhere equal”
- $\sqcap, \sqcup, \overline{g}$ as $\mathit{min}, \mathit{max}, 1 - g$

Notation

- $g, h, \ldots$ are probabilistic predicates
- $\langle G \rangle$ standard predicate as probabilistic predicate
- $\Rrightarrow$ “everywhere less or equal”
- $\Lleftarrow$ “everywhere greater or equal”
- $\equiv$ “everywhere equal”
- $\sqcap, \sqcup, \overline{g}$ as $\mathit{min}, \mathit{max}, 1 - g$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Formal Semantics

- $wp.abort.g := 0$
- $wp.skip.g := g$
- $wp.(x := expr).g := g[x \mapsto expr]$
- $wp.(prog; prog').g := wp.prog.(wp.prog'.g)$
- $wp.(prog \square prog').g := wp.prog.g \sqcap wp.prog'.g$
- $wp.(prog[p]prog').g := p \cdot wp.prog.g + \overline{p} \cdot wp.prog'.g$
- $wp.loop.g := \mu h\,.\,(\langle G \rangle \sqcap wp.body.h) \sqcup (\overline{\langle G \rangle} \sqcap g)$

Outline

Standard Loops

For standard loops we use *invariants:*

$$G \wedge I \Rightarrow wp.body.I$$

For an invariant holds:

$$I \Rightarrow wp.loop.(\neg G \wedge I)$$

Standard Loops

For standard loops we use *invariants:*

$$G \wedge I \Rightarrow wp.body.I$$

For an invariant holds:

$$I \Rightarrow wp.loop.(\neg G \wedge I)$$

Weakest Liberal Preexpectation

Wie “ignore” nontermination:

- $wlp.abort.g := 1$
- $wlp.loop.g := \eta h\,.\,(\langle G\rangle \sqcap wlp.body.h) \sqcup (\overline{\langle G\rangle} \sqcap g)$,
  η largest fixed point
- everything else: $wlp.prog.g := wp.prog.g$

Expectation Invariants

for standard Programs:
$G \wedge I \Rightarrow wp.body.I$

Instead of strict invariance, we require:

$$\langle G \rangle \sqcap I \Rrightarrow wlp.body.I$$

The value of an *expectation invariant* may not decrease if the loop guard holds

Expectation Invariants

for standard Programs:
$G \wedge I \Rightarrow wp.body.I$

Instead of strict invariance, we require:

$$\langle G \rangle \sqcap I \Rrightarrow wlp.body.I$$

The value of an *expectation invariant* may not decrease if the loop guard holds

Expectation Invariants

for standard Programs:
$G \wedge I \Rightarrow wp.body.I$

Instead of strict invariance, we require:

$$\langle G \rangle \sqcap I \Rrightarrow wlp.body.I$$

The value of an *expectation invariant* may not decrease if the loop guard holds

Rule for Partial Correctness

for standard Programs:
$I \Rrightarrow wp.loop.(\neg G \wedge I)$

Lemma 1

Let I be a wlp-invariant of *loop*, then:

$$I \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:
Use properties of largest fixed points:
$x \leq f(x)$ implies $x \leq \eta f$.

Rule for Partial Correctness

for standard Programs:
$I \Rightarrow wp.loop.(\neg G \wedge I)$

Lemma 1

Let I be a wlp-invariant of *loop*, then:

$$I \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

Use properties of largest fixed points:

$x \leq f(x)$ implies $x \leq \eta f$.

Notations

- $T := wp.loop.1$ the "termination expectation"
- $g \,\&\, h := (g + h - 1) \sqcup 0$
  stronger "and" than $\sqcap$
  $wp.prog.(g \,\&\, h) \Leftarrow wlp.prog.(g) \,\&\, wp.prog.(h)$

Notations

- $T := wp.loop.1$ the "termination expectation"
- $g \,\&\, h := (g + h - 1) \sqcup 0$
  stronger "and" than $\sqcap$
  $wp.prog.(g \,\&\, h) \Leftarrow wlp.prog.(g) \,\&\, wp.prog.(h)$

Rule 1 for Total Correctness

Lemma 1: $I \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I)$

Lemma 2

Let I be a wlp-invariant for *loop*. Then

$$I \,\&\, T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

- split *wp* in *wlp* & T
- apply Lemma 1

Useful if I or T is standard.

Rule 1 for Total Correctness

Lemma 1: $I \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I)$

Lemma 2

Let I be a wlp-invariant for *loop*. Then

$$I \,\&\, T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

- split *wp* in *wlp* & T
- apply Lemma 1

Useful if I or T is standard.

Rule 1 for Total Correctness

Lemma 1: $I \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I)$

Lemma 2

Let I be a wlp-invariant for *loop*. Then

$$I \,\&\, T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

- split *wp* in *wlp* & T
- apply Lemma 1

Useful if I or T is standard.

Rule 2 for Total Correctness

for standard Programs:
$I \Rightarrow wp.loop.(\neg G \wedge I)$
Lemma 2: *wlp*-invariant I
$I \,\&\, T \Rightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$

Theorem 3

If I is a wp-invariant of *loop*, then

$$I \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

- Construct *wlp* invariant $I' := I + 1 - T$
- apply Lemma 2

Rule 2 for Total Correctness

for standard Programs:
$I \Rightarrow wp.loop.(\neg G \wedge I)$
Lemma 2: *wlp*-invariant I
$I \,\&\, T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$

Theorem 3

If I is a wp-invariant of *loop*, then

$$I \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$$

Proof Idea:

- Construct *wlp* invariant $I' := I + 1 - T$
- apply Lemma 2

Example: Uniform Selection

Uniform Selection

```
1 init:
2     ℓ := 0;
3     h := N;
4 loop:
5     while ( ℓ + 1 ≠ h) {
6         p := ⌊(ℓ+h)/2⌋;
7         ℓ := p [(h−p)/(h−ℓ)] h:= p
8     }
```

$post := \langle \ell = C \rangle$

$wp.init(wp.loop.post) \equiv \frac{1}{N}$?

$I := \langle \ell \leq C < h \rangle / (h - \ell)$

Example: Uniform Selection

Uniform Selection

```
1 init:
2       ℓ := 0;
3       h := N;
4 loop:
5       while ( ℓ + 1 ≠ h) {
6             p := ⌊(ℓ+h)/2⌋;
7             ℓ := p [(h−p)/(h−ℓ)] h:= p
8       }
```

$post := \langle \ell = C \rangle$

$wp.init(wp.loop.post) \equiv \frac{1}{N}$?

$I := \langle \ell \leq C < h \rangle / (h - \ell)$

Example: Uniform Selection

Uniform Selection

```
1 init:
2     ℓ := 0;
3     h := N;
4 loop:
5     while ( ℓ + 1 ≠ h) {
6         p := ⌊(ℓ+h)/2⌋;
7         ℓ := p [(h−p)/(h−ℓ)] h:= p
8     }
```

$post := \langle \ell = C \rangle$

$wp.init(wp.loop.post) \equiv \frac{1}{N}$?

$I := \langle \ell \leq C < h \rangle / (h - \ell)$

Example: Uniform Selection

Uniform Selection

```
1 init:
2     ℓ := 0;
3     h := N;
4 loop:
5     while ( ℓ + 1 ≠ h) {
6         p := ⌊(ℓ+h)/2⌋;
7         ℓ := p [(h−p)/(h−ℓ)] h:= p
8     }
```

$post := \langle \ell = C \rangle$

$wp.init(wp.loop.post) \equiv \frac{1}{N}$?

$I := \langle \ell \leq C < h \rangle / (h - \ell)$

Example: Uniform Selection

```
1 init:
2  ℓ := 0;
3  h := N;
4 loop:
5  while (ℓ+1≠h){
6   p := ⌊(ℓ+h)/2⌋;
7   ℓ := p[(h−p)/(h−ℓ)]h:= p
8  }
```

- $I := \langle \ell \leq C < h \rangle / (h - \ell)$
- show $I \sqcap \langle \ell + 1 \neq h \rangle \Rrightarrow wp.body.(I)$
- show termination using standard variant.
- apply Theorem 3 for total correctness
  $I \Rrightarrow wp.loop.(\langle \ell + 1 = h \rangle \sqcap I) \equiv$
  $wp.loop.(\langle \ell = C \rangle)$
- *wp* of the whole program:
  $\langle 0 \leq C < N \rangle / N \equiv wp.init.(I) \Rrightarrow$
  $wp.prog.\langle \ell = C \rangle$

Example: Uniform Selection

```
1 init:
2  ℓ := 0;
3  h := N;
4 loop:
5  while (ℓ+1≠h){
6   p := ⌊(ℓ+h)/2⌋;
7   ℓ := p[(h−p)/(h−ℓ)]h:= p
8  }
```

- $I := \langle \ell \leq C < h \rangle / (h - \ell)$
- show $I \sqcap \langle \ell + 1 \neq h \rangle \Rrightarrow wp.body.(I)$
- show termination using standard variant.
- apply Theorem 3 for total correctness
  $I \Rrightarrow wp.loop.(\langle \ell + 1 = h \rangle \sqcap I) \equiv wp.loop.(\langle \ell = C \rangle)$
- *wp* of the whole program:
  $\langle 0 \leq C < N \rangle / N \equiv wp.init.(I) \Rrightarrow wp.prog.\langle \ell = C \rangle$

Example: Uniform Selection

```
1 init:
2  ℓ := 0;
3  h := N;
4 loop:
5  while (ℓ+1≠h){
6   p := ⌊(ℓ+h)/2⌋;
7   ℓ := p[(h−p)/(h−ℓ)]h:= p
8  }
```

- $I := \langle \ell \leq C < h \rangle / (h - \ell)$
- show $I \sqcap \langle \ell + 1 \neq h \rangle \Rrightarrow wp.body.(I)$
- show termination using standard variant.
- apply Theorem 3 for total correctness
  $I \Rrightarrow wp.loop.(\langle \ell + 1 = h \rangle \sqcap I) \equiv wp.loop.(\langle \ell = C \rangle)$
- *wp* of the whole program:
  $\langle 0 \leq C < N \rangle / N \equiv wp.init.(I) \Rrightarrow wp.prog.\langle \ell = C \rangle$

Example: Uniform Selection

```
1 init:
2  ℓ := 0;
3  h := N;
4 loop:
5  while (ℓ+1≠h){
6   p := ⌊(ℓ+h)/2⌋;
7   ℓ := p[(h−p)/(h−ℓ)]h:= p
8  }
```

- $I := \langle \ell \leq C < h \rangle / (h - \ell)$
- show $I \sqcap \langle \ell + 1 \neq h \rangle \Rrightarrow wp.body.(I)$
- show termination using standard variant.
- apply Theorem 3 for total correctness
  $I \Rrightarrow wp.loop.(\langle \ell + 1 = h \rangle \sqcap I) \equiv wp.loop.(\langle \ell = C \rangle)$
- *wp* of the whole program:
  $\langle 0 \leq C < N \rangle / N \equiv wp.init.(I) \Rrightarrow wp.prog.\langle \ell = C \rangle$

Example: Uniform Selection

```
1 init:
2  ℓ := 0;
3  h := N;
4 loop:
5  while (ℓ+1≠h){
6   p := ⌊(ℓ+h)/2⌋;
7   ℓ := p[(h−p)/(h−ℓ)]h:= p
8  }
```

- $I := \langle \ell \leq C < h \rangle / (h - \ell)$
- show $I \sqcap \langle \ell + 1 \neq h \rangle \Rrightarrow wp.body.(I)$
- show termination using standard variant.
- apply Theorem 3 for total correctness
  $I \Rrightarrow wp.loop.(\langle \ell + 1 = h \rangle \sqcap I) \equiv$
  $wp.loop.(\langle \ell = C \rangle)$
- *wp* of the whole program:
  $\langle 0 \leq C < N \rangle / N \equiv wp.init.(I) \Rrightarrow$
  $wp.prog.\langle \ell = C \rangle$

Outline

Standard Variants

Termination for standard programs using **variants**: If for integer expression V holds:

- V decreases in every iteration and
- $V > 0$ for all states in the loop

then termination is guaranteed

"Maximal number of iterations until termination" is a variant for any loop with guaranteed termination.

Standard Variants

Termination for standard programs using **variants**: If for integer expression V holds:

- V decreases in every iteration and
- $V > 0$ for all states in the loop

then termination is guaranteed

"Maximal number of iterations until termination" is a variant for any loop with guaranteed termination.

Standard Variants

Termination for standard programs using **variants**: If for integer expression V holds:

- V decreases in every iteration and
- $V > 0$ for all states in the loop

then termination is guaranteed

"Maximal number of iterations until termination" is a variant for any loop with guaranteed termination.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

0-1 Law for Termination

For every subset of the state space holds:
the infimum over the escape probabilities is 0 or 1

- trap state / bottom scc
- sequence with escape probability converging to 0
- otherwise eventual escape is almost sure

$\Rightarrow$ If for each state in the loop the termination probability is at least p, then the loop terminates almost surely.

Variant Rule

For probabilistic programs:

- V decreases with at least $p > 0$ in each iteration
- V is between L and H for all states in the loop

If such V, L, H and p exist, then the loop terminates almost surely.

Proof Idea:
Use the 0-1 law; $L \leq V < H$ defines the subset of the state space and p is less or equal to the infimum of the termination probabilities.

Complete for finite programs:
V as "distance to terminal state" and p as probability to move on the shortest path out of the loop.
$L = 0$, $H = |\text{Statespace}|$

Variant Rule

For probabilistic programs:

- V decreases with at least $p > 0$ in each iteration
- V is between L and H for all states in the loop

If such V, L, H and p exist, then the loop terminates almost surely.

Proof Idea:
Use the 0-1 law; $L \leq V < H$ defines the subset of the state space and p is less or equal to the infimum of the termination probabilities.

Complete for finite programs:
V as "distance to terminal state" and p as probability to move on the shortest path out of the loop.
$L = 0$, $H = |\text{Statespace}|$

Variant Rule

For probabilistic programs:

- V decreases with at least $p > 0$ in each iteration
- V is between L and H for all states in the loop

If such V, L, H and p exist, then the loop terminates almost surely.

Proof Idea:
Use the 0-1 law; $L \leq V < H$ defines the subset of the state space and p is less or equal to the infimum of the termination probabilities.

Complete for finite programs:
V as "distance to terminal state" and p as probability to move on the shortest path out of the loop.
$L = 0$, $H = |\text{Statespace}|$

Outline

Rules

Correctness Rules:

- Partial Correctness: $I_{wlp} \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I_{wlp})$
  uses wlp-invariant
- Total Correctness: $I_{wlp}\ \&\ T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I_{wlp})$
  useful if I_{wlp} or T is standard
  uses wlp-invariant
- Total Correctness: $I \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$
  requires wp-invariant I

Termination Rules:

- Variant: V decreases with $p > 0$ and $L \leq V < H$

Rules

Correctness Rules:

- Partial Correctness: $I_{wlp} \Rrightarrow wlp.loop.(\overline{\langle G \rangle} \sqcap I_{wlp})$
  uses wlp-invariant
- Total Correctness: $I_{wlp}\ \&\ T \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I_{wlp})$
  useful if I_{wlp} or T is standard
  uses wlp-invariant
- Total Correctness: $I \Rrightarrow wp.loop.(\overline{\langle G \rangle} \sqcap I)$
  requires wp-invariant I

Termination Rules:

- Variant: V decreases with $p > 0$ and $L \leq V < H$

Applications and Extensions

- rules are foundation of automatic correctness proofs [HMM05]
- also applicable for programs with infinite state space
- we still have to find invariants
- more about invariants in the following talk!

Applications and Extensions

- rules are foundation of automatic correctness proofs [HMM05]
- also applicable for programs with infinite state space
- we still have to find invariants
- more about invariants in the following talk!

Applications and Extensions

- rules are foundation of automatic correctness proofs [HMM05]
- also applicable for programs with infinite state space
- we still have to find invariants
- more about invariants in the following talk!

Applications and Extensions

- rules are foundation of automatic correctness proofs [HMM05]
- also applicable for programs with infinite state space
- we still have to find invariants
- more about invariants in the following talk!

Thank you for your attention!

Questions?

Sources


Christel Baier, Joost-Pieter Katoen, et al., *Principles of model checking*, MIT press Cambridge, 2008.


Edsger Wybe Dijkstra, *A discipline of programming*, vol. 4, prentice-hall Englewood Cliffs, 1976.


Friedrich Gretz, J Katoen, and Annabelle McIver, *Operational versus weakest precondition semantics for the probabilistic guarded command language*, Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on, IEEE, 2012, pp. 168–177.


Joe Hurd, Annabelle McIver, and Carroll Morgan, *Probabilistic guarded commands mechanized in hol*, Theoretical Computer Science **346** (2005), no. 1, 96–112.


Charles Antony Richard Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580.


Joost-Pieter Katoen, Annabelle K McIver, Larissa A Meinicke, and Carroll C Morgan, *Linear-invariant generation for probabilistic programs*, Static Analysis, Springer, 2011, pp. 390–406.


Annabelle McIver and Charles Carroll Morgan, *Abstraction, refinement and proof for probabilistic systems*, Springer, 2006.


CC Morgan, *Proof rules for probabilistic loops*, Proceedings of the BCS-FACS 7th Refinement Workshop, Workshops in Computing. Springer Verlag, 1996.