Logic in Computer Science: A Short Introduction to Program Correctness

Hans-Dieter Hiep

Centrum Wiskunde & Informatica (CWI)

Leiden Institute for Advanced Computer Science (LIACS)

PhD Day Vereniging voor Logica, July 1st, 2022

Background

Program verification

> *"How can one check a routine in the sense of making sure that it is right?"*
>
> — *Alan Turing (1949)*

Background

Program verification

> *"How can one check a routine in the sense of making sure that it is right?"*
>
> *— Alan Turing (1949)*

Motivation

1940s —

> *If you don't have a working machine yet, better make sure your program is right!*
> *(First actual case of bug being found)*

Background

Program verification

> *"How can one check a routine in the sense of making sure that it is right?"*
>
> *— Alan Turing (1949)*

Motivation

1940s —

> *If you don't have a working machine yet, better make sure your program is right!*
> *(First actual case of bug being found)*

2010s —

> *Programming errors lead to computer insecurities.*
> *(Apple's SSL bug)*

A Short Introduction

Logic versus Programming

A **sentence** describes situations of the world.

Logic versus Programming

A **sentence** describes situations of the world.

A **program** describes step-by-step processes.

Logic versus Programming

A **sentence** describes situations of the world.

A **program** describes step-by-step processes.

Formal syntax and its semantics

Logic versus Programming

A **sentence** describes situations of the world.

A **program** describes step-by-step processes.

Formal syntax and its semantics

Is a formula **true**?

Logic versus Programming

A **sentence** describes situations of the world.

A **program** describes step-by-step processes.

Formal syntax and its semantics

Is a formula **true**?

Is a program **correct**?

Logic versus Programming

A **sentence** describes situations of the world.

A **program** describes step-by-step processes.

Formal syntax and its semantics

Is a formula **true**?

Is a program **correct**?

Proof systems

Example

Program:

compute the minimum of two integers

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then
2. if $x < y$: assign z the value of x

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then
2. if $x < y$: assign z the value of x
2. if $x \geq y$: assign z the value of y

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then
2. if $x < y$: assign z the value of x
2. if $x \geq y$: assign z the value of y
3. terminate

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then
2. if $x < y$: assign z the value of x
2. if $x \geq y$: assign z the value of y
3. terminate

Correctness:

Example

Program:

compute the minimum of two integers

Formal syntax:

if $x < y$ **then** $z := x$ **else** $z := y$ **fi**

Semantics:

1. compare x and y, then
2. if $x < y$: assign z the value of x
2. if $x \geq y$: assign z the value of y
3. terminate

Correctness:

Is $z = min(x, y)$ true after the program terminates?

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ assignment statement

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ assignment statement
- $S_1 ; S_2$ sequential composition

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1;\ S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1 ; S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement
- **while** p **do** S **od** — looping statement

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1 ; S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement
- **while** p **do** S **od** — looping statement

where:

- x is a **variable**

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1 ; S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement
- **while** p **do** S **od** — looping statement

where:

- x is a **variable**
- t is a **term** (of $\mathscr{L}$)

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1 ; S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement
- **while** p **do** S **od** — looping statement

where:

- x is a **variable**
- t is a **term** (of $\mathscr{L}$)
- p is a (quantifier-free) **formula** (of $\mathscr{L}$)

Programming Language

Let $\mathscr{L}$ be a first-order language (e.g. arithmetic).

We then define a (simple) programming language $\mathscr{P}(\mathscr{L})$.

The **statements** are formed:

- $x := t$ — assignment statement
- $S_1;\ S_2$ — sequential composition
- **if** p **then** S_1 **else** S_2 **fi** — branching statement
- **while** p **do** S **od** — looping statement

where:

- x is a **variable**
- t is a **term** (of $\mathscr{L}$)
- p is a (quantifier-free) **formula** (of $\mathscr{L}$)

(Possibly many other statements, but we keep it simple here!)

Semantics

Logic: we give semantics *a la* **Tarski**.

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

We define a binary relation $\mathscr{M}[\![S]\!]$ on states.

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

We define a binary relation $\mathscr{M}[\![S]\!]$ on states.

- $(\sigma, \sigma[x := t]) \in \mathscr{M}[\![x := t]\!]$

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

We define a binary relation $\mathscr{M}[\![S]\!]$ on states.

- $(\sigma, \sigma[x := t]) \in \mathscr{M}[\![x := t]\!]$
- $(\sigma, \rho) \in \mathscr{M}[\![S_1; S_2]\!]$ if
  $(\sigma, \tau) \in \mathscr{M}[\![S_1]\!]$ and $(\tau, \rho) \in \mathscr{M}[\![S_2]\!]$

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

We define a binary relation $\mathscr{M}[\![S]\!]$ on states.

- $(\sigma, \sigma[x := t]) \in \mathscr{M}[\![x := t]\!]$
- $(\sigma, \rho) \in \mathscr{M}[\![S_1; S_2]\!]$ if
  $(\sigma, \tau) \in \mathscr{M}[\![S_1]\!]$ and $(\tau, \rho) \in \mathscr{M}[\![S_2]\!]$
- $(\sigma, \tau) \in \mathscr{M}[\![\textbf{if } p \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!]$ if
  $\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S_1]\!]$ or
  $\mathscr{M}, \sigma \not\models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S_2]\!]$

Semantics

Logic: we give semantics *a la* **Tarski**.

Given a structure $\mathscr{M}$ and a formula p of our language $\mathscr{L}$.

The satisfaction relation $\mathscr{M}, \sigma \models p$ where σ is a **state**.

Programming: we give a **state transformer** semantics.

We define a binary relation $\mathscr{M}[\![S]\!]$ on states.

- $(\sigma, \sigma[x := t]) \in \mathscr{M}[\![x := t]\!]$
- $(\sigma, \rho) \in \mathscr{M}[\![S_1; S_2]\!]$ if
  $(\sigma, \tau) \in \mathscr{M}[\![S_1]\!]$ and $(\tau, \rho) \in \mathscr{M}[\![S_2]\!]$
- $(\sigma, \tau) \in \mathscr{M}[\![\textbf{if } p \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!]$ if
  $\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S_1]\!]$ or
  $\mathscr{M}, \sigma \not\models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S_2]\!]$
- etc.

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Given a model $\mathscr{M}$, we define $\mathscr{M} \models \{p\}\ S\ \{q\}$ as
$\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S]\!]$ implies $\mathscr{M}, \tau \models q$ for all σ, τ.

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Given a model $\mathscr{M}$, we define $\mathscr{M} \models \{p\}\ S\ \{q\}$ as
$\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S]\!]$ implies $\mathscr{M}, \tau \models q$ for all σ, τ.

Example

$\models \{\textbf{true}\}$ **if** $x < y$ **then** $z := x$ **else** $z := y$ **fi** $\{z = min(x, y)\}$

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Given a model $\mathscr{M}$, we define $\mathscr{M} \models \{p\}\ S\ \{q\}$ as
$\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S]\!]$ implies $\mathscr{M}, \tau \models q$ for all σ, τ.

Example

$\models \{\textbf{true}\}\ \textbf{if}\ x < y\ \textbf{then}\ z := x\ \textbf{else}\ z := y\ \textbf{fi}\ \{z = min(x, y)\}$

A proof system to derive correctness formulas?

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Given a model $\mathscr{M}$, we define $\mathscr{M} \models \{p\}\ S\ \{q\}$ as
$\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S]\!]$ implies $\mathscr{M}, \tau \models q$ for all σ, τ.

Example

$\models \{\textbf{true}\}$ **if** $x < y$ **then** $z := x$ **else** $z := y$ **fi** $\{z = min(x, y)\}$

A proof system to derive correctness formulas?

Soundness: all derivated correctness formulas are valid.

Program Logic

$$\{p\}\ S\ \{q\}$$

is a **correctness formula**, where p and q are formula of $\mathscr{L}$.

Given a model $\mathscr{M}$, we define $\mathscr{M} \models \{p\}\ S\ \{q\}$ as
$\mathscr{M}, \sigma \models p$ and $(\sigma, \tau) \in \mathscr{M}[\![S]\!]$ implies $\mathscr{M}, \tau \models q$ for all σ, τ.

Example

$\models \{\textbf{true}\}$ **if** $x < y$ **then** $z := x$ **else** $z := y$ **fi** $\{z = min(x, y)\}$

A proof system to derive correctness formulas?

Soundness: all derivated correctness formulas are valid.

Completeness: all valid correctness formulas are derivable.

Proof system

Our proof system for valid correctness formulas

$$\vdash \{p\}\ S\ \{q\}$$

depends on a proof system for valid formulas

$$\vdash p$$

Proof system

Our proof system for valid correctness formulas

$$\vdash \{p\}\ S\ \{q\}$$

depends on a proof system for valid formulas

$$\vdash p$$

A Hilbert-style proof system: **axioms** and **rules** of inference.

Proof system

Our proof system for valid correctness formulas

$$\vdash \{p\}\ S\ \{q\}$$

depends on a proof system for valid formulas

$$\vdash p$$

A Hilbert-style proof system: **axioms** and **rules** of inference.

We abstract from the underlying system, by a so-called **oracle**.

Proof system

Our proof system for valid correctness formulas

$$\vdash \{p\}\ S\ \{q\}$$

depends on a proof system for valid formulas

$$\vdash p$$

A Hilbert-style proof system: **axioms** and **rules** of inference.

We abstract from the underlying system, by a so-called **oracle**.

Our proof system consists of:

- **Analytic** rules (follows purely from program semantics)
- **Synthetic** rules (relies on underlying logical structure)

Program Logic

Definition (1/2: analytical rules and axiom)

ASSIGNMENT

$$\frac{}{\{p[x := t]\}\ x := t\ \{p\}}$$

COMPOSITION

$$\frac{\{p\}\ S_1\ \{r\} \quad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

CONDITIONAL

$$\frac{\{p \wedge B\}\ S_1\ \{q\} \quad \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$$

WHILE

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg B\}}$$

Definition (2/2: synthetic rule)

CONSEQUENCE (adaptation)

$$\frac{p \to p' \quad \{p'\}\, S\, \{q'\} \quad q' \to q}{\{p\}\, S\, \{q\}}$$

Definition (2/2: synthetic rule)

CONSEQUENCE (adaptation)

$$\frac{p \to p' \quad \{p'\}\ S\ \{q'\} \quad q' \to q}{\{p\}\ S\ \{q\}}$$

Now, $\vdash \{p\}\ S\ \{q\}$ means

there exists a proof with $\{p\}\ S\ \{q\}$ as conclusion.

Proof Complexity

Definition (Size of program)

- $\ell(\bar{x} := \bar{t}) = 1$
- $\ell(S_1;\ S_2) = \ell(S_1) + \ell(S_2) + 1$
- $\ell(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \ell(S_1) + \ell(S_2) + 1$
- $\ell(\textbf{while } B \textbf{ do } S \textbf{ od}) = \ell(S) + 1$

Proof Complexity

Definition (Size of program)

- $\ell(\bar{x} := \bar{t}) = 1$
- $\ell(S_1;\ S_2) = \ell(S_1) + \ell(S_2) + 1$
- $\ell(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \ell(S_1) + \ell(S_2) + 1$
- $\ell(\textbf{while } B \textbf{ do } S \textbf{ od}) = \ell(S) + 1$

Lemma

If $\vdash \{p\}\ S\ \{q\}$ *then there exists a proof with at most* $2 \times \ell(S)$ *rule applications and* $\{p\}\ S\ \{q\}$ *as conclusion.*

Linear Proofs

Lemma

If $\vdash \{p\}\ S\ \{q\}$ *then there exists a proof with at most* $2 \times \ell(S)$ *rule applications and* $\{p\}\ S\ \{q\}$ *as conclusion.*

Technique: **proof normalization**

Linear Proofs

Lemma

If $\vdash \{p\}\ S\ \{q\}$ *then there exists a proof with at most* $2 \times \ell(S)$ *rule applications and* $\{p\}\ S\ \{q\}$ *as conclusion.*

Technique: **proof normalization**

$$\dfrac{p \to p' \qquad \dfrac{p' \to p'' \quad \{p''\}\ S\ \{q''\} \quad q'' \to q'}{\{p'\}\ S\ \{q'\}} \qquad q' \to q}{\{p\}\ S\ \{q\}}$$

can be collapsed into

$$\dfrac{p \to p'' \quad \{p''\}\ S\ \{q''\} \quad q'' \to q}{\{p\}\ S\ \{q\}}$$

My Contributions

Formal meta-theory

Formalize program correctness in a **higher-order logic**

Formal meta-theory

Formalize program correctness in a **higher-order logic**

Dependent type theory, calculus of constructions

Formal meta-theory

Formalize program correctness in a **higher-order logic**

Dependent type theory, calculus of constructions

My work:

- using the proof assistant Coq

Formal meta-theory

Formalize program correctness in a **higher-order logic**

Dependent type theory, calculus of constructions

My work:

- using the proof assistant Coq

My goal: full formal meta-theoretical verification

Program Correctness in Practice

We have seen just the basics.

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software?

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software? **Yes.**

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software? **Yes.**

My focus: the Java programming language

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software? **Yes.**

My focus: the Java programming language

- Many and complex programming features
- Difficult program semantics
- Expression problem
- Frame problem

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software? **Yes.**

My focus: the Java programming language

- Many and complex programming features
- Difficult program semantics
- Expression problem
- Frame problem

My work:

- Verification of standard library code
  (Java Collection Framework's LinkedList class)

Program Correctness in Practice

We have seen just the basics.

Can this be applied to real-world software? **Yes.**

My focus: the Java programming language

- Many and complex programming features
- Difficult program semantics
- Expression problem
- Frame problem

My work:

- Verification of standard library code
  (Java Collection Framework's LinkedList class)

My goal: verification of Dutch election software