On Logic and Computability

S Akshay

Dept of CSE, IIT Bombay

Special lecture @ DIC course
30 Jan 2023

Bugs in programs can be costly!

Bugs in programs can be costly!

June 4, 1996 - Ariane 5 Flight 501.

"Working code for the Ariane 4 rocket is reused in the Ariane 5, but the Ariane 5's faster engines trigger a bug in an arithmetic routine inside the rocket's flight computer. The error is in the code that converts a 64-bit floating-point number to a 16-bit signed integer. The faster engines cause the 64-bit numbers to be larger in the Ariane 5 than in the Ariane 4, triggering an overflow condition that results in the flight computer crashing."

Source: Simson Garfinkel, Wired Magazine, 2005: `https://www.wired.com/2005/11/historys-worst-software-bugs/`

Bugs in programs can be costly!

June 4, 1996 - Ariane 5 Flight 501.

1993-Intel Pentium floating point divide.

"A silicon error causes Intel's highly promoted Pentium chip to make mistakes when dividing floating-point numbers that occur within a specific range. For example, dividing 4195835.0/3145727.0 yields 1.33374 instead of 1.33382, an error of 0.006 percent... The bug ultimately costs Intel $475 million."

Source: Simson Garfinkel, Wired Magazine, 2005: `https://www.wired.com/2005/11/historys-worst-software-bugs/`

Bugs in programs can be costly!

June 4, 1996 - Ariane 5 Flight 501.

1993-Intel Pentium floating point divide.

2005 - Toyota Prius Bug.
"Toyota announced a recall of 160,000 of its Prius hybrid vehicles following reports of vehicle warning lights illuminating for no reason, and cars' gasoline engines stalling unexpectedly. But unlike the large-scale auto recalls of years past, the root of the Prius issue wasn't a hardware problem - it was a programming error in the smart car's embedded code. The Prius had a software bug. "

Source: Simson Garfinkel, Wired Magazine, 2005: `https://www.wired.com/2005/11/historys-worst-software-bugs/`

Bugs in programs can be costly!

June 4, 1996 - Ariane 5 Flight 501.

1993-Intel Pentium floating point divide.

2005 - Toyota Prius Bug.

- Software testing not enough always...
- Need mathematically formal and rigorous guarantees.
- How do we formalize software/programs?

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us start with an example

- What does this program do?

	$y := 1;$
	$z := 0;$
P1	*while* $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$

Let us start with an example

- What does this program do?

	$y := 1;$
	$z := 0;$
P1	*while* $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$

- **P1** computes $x!$ and stores it in y.

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$

- **P1** computes $x!$ and stores it in y.
- But does it do exactly that?

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$
- What if input $x = -2$?

- **P1** computes $x!$ and stores it in y.
- But does it do exactly that?

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$
- What if input $x = -2$?

- **P1** computes $x!$ and stores it in y.
- But does it do exactly that?
- In fact, if $x \geq 0$, then **P1** computes $x!$ and stores it in y.

Let us start with an example

- What does this program do?

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Let us try out some runs

- If input $x = 1$, then output $y = 1$
- If input $x = 2$, then output $y = 2$
- If input $x = 3$, then output $y = 6$
- If input $x = 4$, then output $y = 24$
- What if input $x = -2$?

- **P1** computes $x!$ and stores it in y.
- But does it do exactly that?
- In fact, if $x \geq 0$, then **P1** computes $x!$ and stores it in y.
- How do we prove that?!

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

The first important idea about correctness:

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

The first important idea about correctness:

A formal specification is important

- Correctness is a property relating the input and output values and the program.

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

The first important idea about correctness:

A formal specification is important

- Correctness is a property relating the input and output values and the program.
- Can be written as mathematical/logical expressions

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

The first important idea about correctness:

A formal specification is important

- Correctness is a property relating the input and output values and the program.
- Can be written as mathematical/logical expressions
  - E.g, If $(x \geq 0)$, then **P1** will stop and give $(y = x!)$.

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$.

The first important idea about correctness:

A formal specification is important

- Correctness is a property relating the input and output values and the program.
- Can be written as mathematical/logical expressions
  - E.g, If $(x \geq 0)$, then **P1** will stop and give $(y = x!)$.
  - For short, we sometimes write it as a triple: $\{x \geq 0\}\mathbf{P1}\{y = x!\}$

Proving programs correct

When **P1** *is run with any input* x*, if* **P1** *stops, we have* $y = x!$*.*

The first important idea about correctness:

A formal specification is important

- Correctness is a property relating the input and output values and the program.
- Can be written as mathematical/logical expressions
  - E.g, If $(x \geq 0)$, then **P1** will stop and give $(y = x!)$.
  - For short, we sometimes write it as a triple: $\{x \geq 0\}\mathbf{P1}\{y = x!\}$
  - $(x \geq 0)$ is called the Pre-condition and $(y = x!)$ is called the Post-condition.

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

- Mathematical expressions/(in)equalities over say integers: $x + 5 \leq 8 * y$, etc

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

- Mathematical expressions/(in)equalities over say integers: $x + 5 \leq 8 * y$, etc
- Boolean combinations
  - True, False
  - negation of an expression: $\neg(x = 0)$
  - conjunction of two expressions: $(x \geq 0) \ \wedge \ (y \neq 0)$

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

- Mathematical expressions/(in)equalities over say integers: $x + 5 \leq 8 * y$, etc
- Boolean combinations
  - True, False
  - negation of an expression: $\neg(x = 0)$
  - conjunction of two expressions: $(x \geq 0) \ \wedge \ (y \neq 0)$
- Just like COVID-variants, we use Greek symbols to denote them $\{\varphi\}$**Prog**$\{\psi\}$.

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

- Mathematical expressions/(in)equalities over say integers: $x + 5 \leq 8 * y$, etc
- Boolean combinations
  - True, False
  - negation of an expression: $\neg(x = 0)$
  - conjunction of two expressions: $(x \geq 0) \ \wedge \ (y \neq 0)$
- Just like COVID-variants, we use Greek symbols to denote them $\{\varphi\}\mathbf{Prog}\{\psi\}$.

This defines what is called a "logic".

Formal specification and logic

More generally, we can write things like $(x \geq 0) \ \wedge \ (y \neq 0)$ **Prog** $(y = x!)$.
– $(x \geq 0) \ \wedge \ (y \neq 0)$ is the Pre-condition and $(y = x!)$ is the Post-condition.

- Mathematical expressions/(in)equalities over say integers: $x + 5 \leq 8 * y$, etc
- Boolean combinations
  - True, False
  - negation of an expression: $\neg(x = 0)$
  - conjunction of two expressions: $(x \geq 0) \ \wedge \ (y \neq 0)$
- Just like COVID-variants, we use Greek symbols to denote them $\{\varphi\}$**Prog**$\{\psi\}$.

This defines what is called a "logic".

Hoare Logic or Floyd-Hoare Logic

- Developed by Tony Hoare (based on Robert Floyd)
- Triples are called Hoare triples.
- Hoare won Turing Award in 1980, Floyd in 1978!

Building a proof of correctness bottom up

Consider a simple program

P2	$\{$
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	$\}$

Building a proof of correctness bottom up

Consider a simple program

P2	$\{$
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	$\}$

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

$y := 4;$

$x := y + 1$

1. Thus, we can down write the program line by line

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

True

$y := 4;$

$x := y + 1$

$\{x < 6\}$

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

$True$

$y := 4;$

$x := y + 1$

$\{x < 6\}$ Assignment

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.
3. Use "rules" to go backward! Weakest Pre-condition

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

$True$

$y := 4;$

$\{y + 1 < 6\}$

$x := y + 1$

$\{x < 6\}$ Assignment

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.
3. Use "rules" to go backward! Weakest Pre-condition

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

True

$y := 4;$

$\{y < 5\}$

$\{y + 1 < 6\}$ Logically implies

$x := y + 1$

$\{x < 6\}$ Assignment

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.
3. Use "rules" to go backward! Weakest Pre-condition
4. Can also use simple reasoning over basic arithmetic.

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

True	
$\{4 < 5\}$	
$y := 4;$	
$\{y < 5\}$	Assignment
$\{y + 1 < 6\}$	Logically implies
$x := y + 1$	
$\{x < 6\}$	Assignment

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.
3. Use "rules" to go backward! Weakest Pre-condition
4. Can also use simple reasoning over basic arithmetic.

Building a proof of correctness bottom up

Consider a simple program

P2	{
inp $\emptyset$	$y := 4;$
out y	$x := y + 1;$
	}

True	
$\{4 < 5\}$	Logically implies
$y := 4;$	
$\{y < 5\}$	Assignment
$\{y + 1 < 6\}$	Logically implies
$x := y + 1$	
$\{x < 6\}$	Assignment

1. Thus, we can down write the program line by line
2. Write down what must hold before and after as formula.
3. Use "rules" to go backward! Weakest Pre-condition
4. Can also use simple reasoning over basic arithmetic.

Using Logic for Program correctness

Main idea: Use logic to capture what must be true before and after a command.

- Break up the program into individual command lines.
- Reason about each command using a "rule".
- Stitch them together to get a proof of desired specification.

Using Logic for Program correctness

Main idea: Use logic to capture what must be true before and after a command.

- Break up the program into individual command lines.
- Reason about each command using a "rule".
- Stitch them together to get a proof of desired specification.
  - If $\{\varphi\}$**Prog1**$\{\psi\}$ and $\{\psi\}$**Prog2**$\{\xi\}$, then we can conclude $\{\varphi\}$**Prog1; Prog2**$\{\xi\}$.

The difficult part: the while loop!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What must be true about the while loop?
- For it to be correct, before and after must be the same!
- This is called a **Loop invariant**
  - Starting with $x = 6$, write values of x, y, z before and after loop.

The difficult part: the while loop!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What must be true about the while loop?
- For it to be correct, before and after must be the same!
- This is called a **Loop invariant**
  - Starting with $x = 6$, write values of x, y, z before and after loop.
  - Notice that $y = z!$ always holds!

The difficult part: the while loop!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What must be true about the while loop?
- For it to be correct, before and after must be the same!
- This is called a **Loop invariant**
  - Starting with $x = 6$, write values of x, y, z before and after loop.
  - Notice that $y = z!$ always holds!

Further we have:

- Pre-condn of while loop implies it: $(y = 1 \ \wedge \ z = 0) \implies (y = z!)$.
- Together with negation of guard, implies post-condn: $(y = z! \ \wedge \ x = z) \implies (y = x!)$ is valid.

The difficult part: the while loop!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What must be true about the while loop?
- For it to be correct, before and after must be the same!
- This is called a **Loop invariant**
  - Starting with $x = 6$, write values of x, y, z before and after loop.
  - Notice that $y = z!$ always holds!

Further we have:

- Pre-condn of while loop implies it: $(y = 1 \ \wedge \ z = 0) \implies (y = z!)$.
- Together with negation of guard, implies post-condn: $(y = z! \ \wedge \ x = z) \implies (y = x!)$ is valid.

So, here is the full proof for the factorial program now!

The factorial example solved!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\quad z := z + 1$

$\quad y := y * z$

$\}$

$\{y = x!\}$

The factorial example solved!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\quad z := z + 1$

$\quad y := y * z$

$\{y = z!\}$

$\}$

$\{y = x!\}$

The factorial example solved!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\quad z := z + 1$

$\{y \cdot z = z!\}$

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = x!\}$

The factorial example solved!

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z+1) = (z+1)!\}$

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = x!\}$

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z+1) = (z+1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = x!\}$

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

$\{y = z!\}$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

$\{y = z!\}$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$z := 0;$

$\{y = z!\}$

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$ Implied

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$y := 1;$

$\{y = 0!\}$

$z := 0;$

$\{y = z!\}$ Assignment

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$ Implied

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$\{1 = 0!\}$

$y := 1;$

$\{y = 0!\}$ Assignment

$z := 0;$

$\{y = z!\}$ Assignment

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$ Implied

The factorial example solved!

P1 *inp x* *out y*	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$\{True\}$

$\{1 = 0!\}$ Implied

$y := 1;$

$\{y = 0!\}$ Assignment

$z := 0;$

$\{y = z!\}$ Assignment

while $(z \neq x)\{$

$\{y = z! \ \wedge \ z \neq x\}$ Loop Inv. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \ \wedge \ \neg(z \neq x)\}$ Loop Inv. and Not of Guard

$\{y = x!\}$ Implied

What is a talk without some HomeWork?

Ex1 *inp* x, y *out* z	$a := 0;$ $z := 0;$ *while* $(a \neq y)\{$ $\quad z := z + x;$ $\quad a := a + 1;$ $\}$

Ex2 *inp* x, y *out* d	$r := x;$ $d := 0;$ *while* $(r \geq y)\{$ $\quad r := r - y;$ $\quad d := d + 1;$ $\}$

What is a talk without some HomeWork?

Ex1 *inp* x, y *out* z	$a := 0;$ $z := 0;$ *while* $(a \neq y)\{$ $\quad z := z + x;$ $\quad a := a + 1;$ $\}$

Ex2 *inp* x, y *out* d	$r := x;$ $d := 0;$ *while* $(r \geq y)\{$ $\quad r := r - y;$ $\quad d := d + 1;$ $\}$

Exercise Show correctness

1. $\{True\}\text{Ex1}\{(z = x \cdot y)\}$
2. $\{y \geq 0\}\text{Ex1}\{(z = x \cdot y)\}$

What is a talk without some HomeWork?

Ex1 *inp* x, y *out* z	$a := 0;$ $z := 0;$ *while* $(a \neq y)\{$ $\quad z := z + x;$ $\quad a := a + 1;$ $\}$

Ex2 *inp* x, y *out* d	$r := x;$ $d := 0;$ *while* $(r \geq y)\{$ $\quad r := r - y;$ $\quad d := d + 1;$ $\}$

Exercise Show correctness

1. $\{True\}\text{Ex1}\{(z = x \cdot y)\}$
2. $\{y \geq 0\}\text{Ex1}\{(z = x \cdot y)\}$

Exercise

1. What does the program *Ex*2 do? Figure out Hoare triples and show correctness.

A few points to note

We “guessed” the loop invariant!

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.

A few points to note

We “guessed” the loop invariant!

- if invariant is too weak proof can’t be completed.
- Computing strongest invariant of a loop: Impossible in general!

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.
- Computing strongest invariant of a loop: Impossible in general!

Takeaways for YOUR program

1. Write programs with an idea of the formal specification!
2. When writing loops think of invariant should hold (so it satisfies the spec).

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.
- Computing strongest invariant of a loop: Impossible in general!

Takeaways for YOUR program

1. Write programs with an idea of the formal specification!
2. When writing loops think of invariant should hold (so it satisfies the spec).

So, why don't programmers always do this

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.
- Computing strongest invariant of a loop: Impossible in general!

Takeaways for YOUR program

1. Write programs with an idea of the formal specification!
2. When writing loops think of invariant should hold (so it satisfies the spec).

So, why don't programmers always do this

- Good ones do, at least at a high level!

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.
- Computing strongest invariant of a loop: Impossible in general!

Takeaways for YOUR program

1. Write programs with an idea of the formal specification!
2. When writing loops think of invariant should hold (so it satisfies the spec).

So, why don't programmers always do this

- Good ones do, at least at a high level!
- Difficult with more complicated programs $\sim$ 1 Million lines of code!

A few points to note

We "guessed" the loop invariant!

- if invariant is too weak proof can't be completed.
- Computing strongest invariant of a loop: Impossible in general!

Takeaways for YOUR program

1. Write programs with an idea of the formal specification!
2. When writing loops think of invariant should hold (so it satisfies the spec).

So, why don't programmers always do this

- Good ones do, at least at a high level!
- Difficult with more complicated programs $\sim$ 1 Million lines of code!
- Can you automate it?

Can we automate program verification?

Can we write programs to get such proofs of correctness of other programs?

- Programmer still has to provide the "intended" loop invariant.
- But verifying if they are correct is easy and can be automated,
- All other steps can also be automated!

Can we automate program verification?

Can we write programs to get such proofs of correctness of other programs?

- Programmer still has to provide the "intended" loop invariant.
- But verifying if they are correct is easy and can be automated,
- All other steps can also be automated!
- Such tools exist... e.g., Coq, Isabelle, F*.
- Many more things to handle: Heaps, dynamic data, pointers etc.

Can we automate program verification?

Can we write programs to get such proofs of correctness of other programs?

- Programmer still has to provide the "intended" loop invariant.
- But verifying if they are correct is easy and can be automated,
- All other steps can also be automated!
- Such tools exist... e.g., Coq, Isabelle, F*.
- Many more things to handle: Heaps, dynamic data, pointers etc.

Many advanced techniques and tools for automatic synthesis of invariants.

Can we automate program verification?

Can we write programs to get such proofs of correctness of other programs?

- Programmer still has to provide the "intended" loop invariant.
- But verifying if they are correct is easy and can be automated,
- All other steps can also be automated!
- Such tools exist... e.g., Coq, Isabelle, F*.
- Many more things to handle: Heaps, dynamic data, pointers etc.

Many advanced techniques and tools for automatic synthesis of invariants.

- Abstract interpretation
- Constraint solving

Can we automate program verification?

Can we write programs to get such proofs of correctness of other programs?

- Programmer still has to provide the "intended" loop invariant.
- But verifying if they are correct is easy and can be automated,
- All other steps can also
- Such tools exist... e.g.,
- Many more things to ha .

Many advanced techniques a iants.

- Abstract interpretation
- Constraint solving

But...

On the role of logic in CS

A whole zoo of logics .... depending on the application/system of interest.

- Propositional and predicate logic
- Temporal logic
- Much more.. see in CS 228.

So, what is the role of Logic?

- A Language for formal specification
- A Calculus for Reasoning!
- Link between proof and automation, man and machine :P

1. Men are mortal
2. Socrates is a man

Socrates is mortal

On the role of logic in CS

A whole zoo of logics .... depending on the application/system of interest.

- Propositional and predicate logic
- Temporal logic
- Much more.. see in CS 228.

So, what is the role of Logic?

- A Language for formal specification
- A Calculus for Reasoning!
- Link between proof and automation, man and machine :P

1. Men are mortal
2. Socrates is a man

Socrates is mortal

Intuitive Rule/Pattern:

1. α are β
2. γ is an α

γ is β

On the role of logic in CS

A whole zoo of logics .... depending on the application/system of interest.

- Propositional and predicate logic
- Temporal logic
- Much more.. see in CS 228.

So, what is the role of Logic?

- A Language for formal specification
- A Calculus for Reasoning!
- Link between proof and automation, man and machine :P

1. Men are mortal
2. Socrates is a man

Socrates is mortal

Intuitive Rule/Pattern:

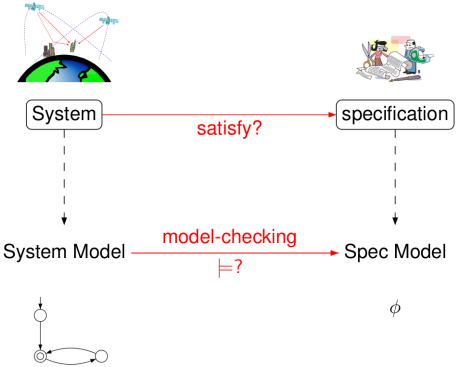
1. α are β
2. γ is an α

γ is β

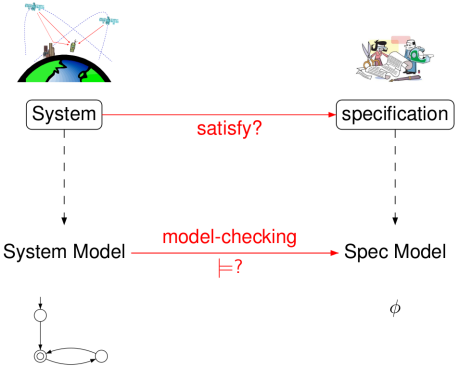
1. A barber shaves all those who don't shave themselves
2. The barber needs a shave

Who shaves the barber?

On the role of logic in CS: Model checking

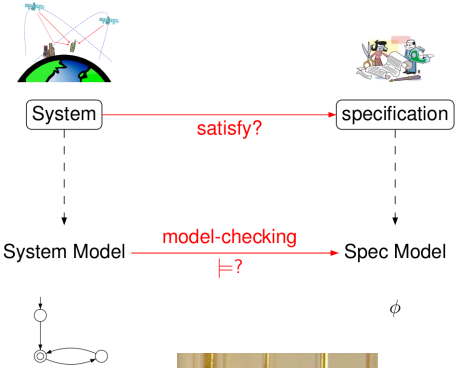


On the role of logic in CS: Model checking

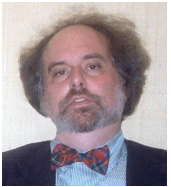


- Widely used in hardware and software verification

On the role of logic in CS: Model checking



- Widely used in hardware and software verification
- More Turing awards:
  - Amir Pneuli in 1997
  - Ed Clarke, E. Allen Emerson, Joseph Sifakis in 2007



Part 2: Computability

Let's go back to our simple program

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Stopping matters

- Correctness: when it stops it satisfies the specification.
- Termination: it always stops and it satisfies the specification.

Let's go back to our simple program

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Stopping matters

- Correctness: when it stops it satisfies the specification.
- Termination: it always stops and it satisfies the specification.

E.g., $\{True\}\mathbf{P1}\{y = x!\}$ is correct; $\{x \geq 0\}\mathbf{P1}\{y = x!\}$ is correct and terminates.

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while, so the only thing to change is the while rule!

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while, so the only thing to change is the while rule!

Core idea!

- for correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while, so the only thing to change is the while rule!

Core idea!

- for correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while, so the only thing to change is the while rule!

Core idea!

- for correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.
  - hence, it can decrease only finitely many times and hence the loop terminates.

Total correctness

- Correctness rules till now do not say anything if a program loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while, so the only thing to change is the while rule!

Core idea!

- for correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.
  - hence, it can decrease only finitely many times and hence the loop terminates.
  - Such an expression is called a variant.

Revisiting the Factorial example

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?

Revisiting the Factorial example

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?
- easy! the expression $x - z$.

Revisiting the Factorial example

P1 *inp* x *out* y	$y := 1;$ $z := 0;$ *while* $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?
- easy! the expression $x - z$.
- Note that it is always nonnegative.
- When it is 0, the loop terminates!

Can we always get variants to show loop termination?

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- If input $x = 4$, then
- If input $x = 5$, then
- If input $x = 11$, then

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- If input $x = 4$, then $c = 4, 2, 1$
- If input $x = 5$, then
- If input $x = 11$, then

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x$;
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\}$;
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- If input $x = 4$, then $c = 4, 2, 1$
- If input $x = 5$, then $c = 5, 16, 8, 4, 2, 1$
- If input $x = 11$, then

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- If input $x = 4$, then $c = 4, 2, 1$
- If input $x = 5$, then $c = 5, 16, 8, 4, 2, 1$
- If input $x = 11$, then $c = 11, 34, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- If input $x = 4$, then $c = 4, 2, 1$
- If input $x = 5$, then $c = 5, 16, 8, 4, 2, 1$
- If input $x = 11$, then $c = 11, 34, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$
- Is there any input for which it does not terminate!?

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)$ $\{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- Is there any input for which it does not terminate!?
- In other words, is $\{x > 0\}\mathbf{P3}\{True\}$ totally correct?

Can we always get variants to show loop termination?

Consider the program

P3
inp x
out c
	$c := x;$
	while $(c \neq 1)\{$
	  if $(c \mod 2 = 0)\ \{c := c/2\};$
	  else $\{c := 3 * c + 1;\}$
	$\}$

Let us try our some runs:

- Is there any input for which it does not terminate!?
- In other words, is $\{x > 0\}$**P3**$\{True\}$ totally correct?

This is called the Collatz function.

- There is no known initial value $x > 0$ for which Collatz does not terminate.
- But there is no known proof that it always terminates!

So, how hard is Program Termination?

It is in fact impossible! But what does that even mean?

So, how hard is Program Termination?

An impossible program

- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.

So, how hard is Program Termination?

An impossible program

- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.
- Does there exist an algorithm/program M, such that it takes as input any such program P, and answers Yes if P halts and NO otherwise?

So, how hard is Program Termination?

An impossible program

- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.
- Does there exist an algorithm/program M, such that it takes as input any such program P, and answers Yes if P halts and NO otherwise?

Goes back to the dawn of Computer Science

So, how hard is Program Termination?

An impossible program

- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.
- Does there exist an algorithm/program M, such that it takes as input any such program P, and answers Yes if P halts and NO otherwise?

Goes back to the dawn of Computer Science

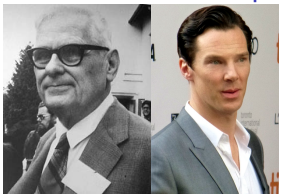


Figure: Alonso Church & Alan Turing

So, how hard is Program Termination?

An impossible program

- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.
- Does there exist an algorithm/program M, such that it takes as input any such program P, and answers Yes if P halts and NO otherwise?

Goes back to the dawn of Computer Science

Figure: Alonso Church & Alan Turing

So, how hard is Program Termination?

An impossible program

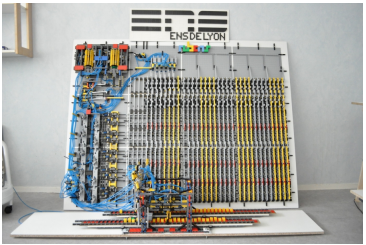
- For any program P, can "we say" if program P halts?
  - Let us only consider those programs that have no inputs, e.g., **P2** above.
- Does there exist an algorithm/program M, such that it takes as input any such program P, and answers Yes if P halts and NO otherwise?

Goes back to the dawn of Computer Science

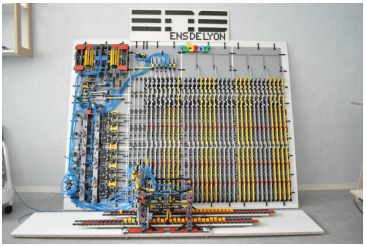
Figure: Alonso Church & Alan Turing

- The Church-Turing Hypothesis
- Capture our intuitive notion of an "algorithm"
- Foundation of modern computers and programs

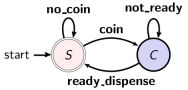
Formalizing computability: Turing machines



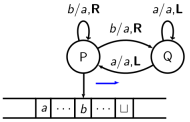
Formalizing computability: Turing machines



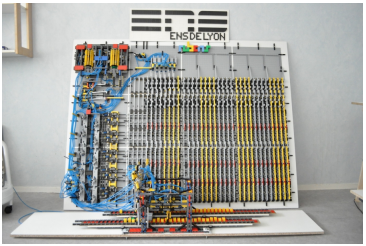
Finite instruction machine with finite memory (Finite State Automata)



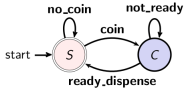
Finite instruction machine with unbounded memory (Turing machine)



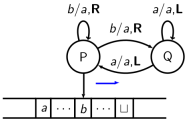
Formalizing computability: Turing machines



Finite instruction machine with finite memory (Finite State Automata)

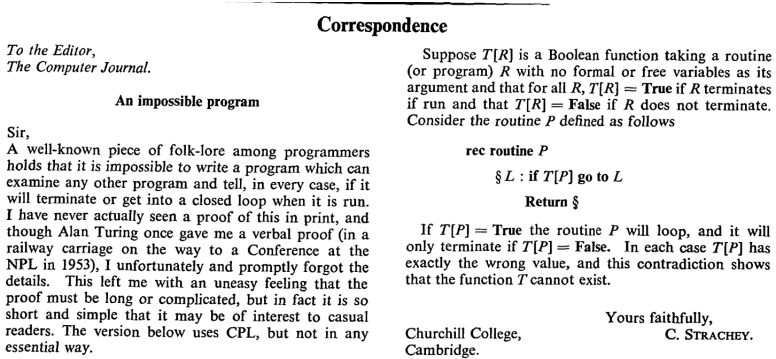


Finite instruction machine with unbounded memory (Turing machine)



- A theoretical model for computability
- Basis of modern computers and programs
- Much more about this in CS310: Automata theory course

An impossible program: a historical proof!



Correspondence

To the Editor,
The Computer Journal.

An impossible program

Sir,
A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose $T[R]$ is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, $T[R] =$ **True** if R terminates if run and that $T[R] =$ **False** if R does not terminate. Consider the routine P defined as follows

rec routine P

§ L : **if** $T[P]$ **go to** L

Return §

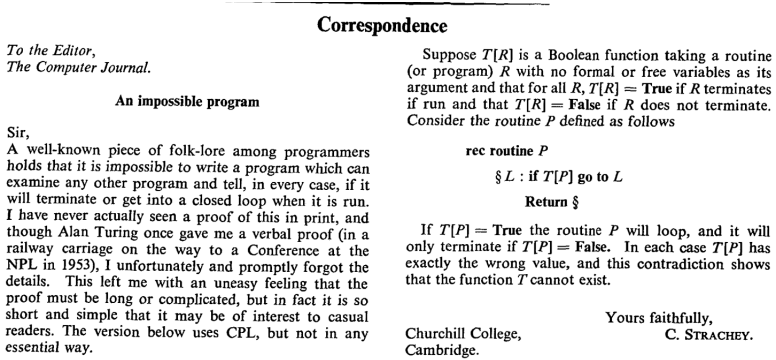
If $T[P] =$ **True** the routine P will loop, and it will only terminate if $T[P] =$ **False**. In each case $T[P]$ has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully,
C. STRACHEY.

Churchill College,
Cambridge.

Figure: From C. Strachey's, letter to the editor, The Computer Journal, Volume 7, Issue 4, January 1965

An impossible program: a historical proof!



Correspondence

To the Editor,
The Computer Journal.

An impossible program

Sir,
A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose $T[R]$ is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, $T[R] =$ **True** if R terminates if run and that $T[R] =$ **False** if R does not terminate. Consider the routine P defined as follows

rec routine P

§ L : **if** $T[P]$ **go to** L

Return §

If $T[P] =$ **True** the routine P will loop, and it will only terminate if $T[P] =$ **False**. In each case $T[P]$ has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully,
C. STRACHEY.

Churchill College,
Cambridge.

Figure: From C. Strachey's, letter to the editor, The Computer Journal, Volume 7, Issue 4, January 1965

- A proof goes via "diagonalization"! (hint: can you show that real numbers are uncountable?)

Conclusion

To know what you can do, you must know what you can't!

A few take home points

- To solve a problem, you need to formalize it: to model and reason.
- Logic is a calculus for (automated) reasoning.
- Automata and Turing machines are models of computation.
- There are problems that are hard for a computer to solve .. and then there are problems that are impossible!
- From impossibility rises possibilities: what to do when faced with impossible problems?