Does my program ever finish?

Alexey Bakhirkin
Supervisor: Nir Piterman

University of Leicester, Department of Computer Science

What *program analysis* people do

They reason about programs (without running them),

- especially, low-level imperative programs, e.g., device drivers;
- because the price of failure is high, and there's demand from software industry;
- because low-level programs usually do simple things, and can be tackled by formal methods.

What *program analysis* people do

They reason about programs (without running them),

- especially, low-level imperative programs, e.g., device drivers;
- because the price of failure is high, and there's demand from software industry;
- because low-level programs usually do simple things, and can be tackled by formal methods.

They like hard problems, e.g.,

- boolean satisfiability – a famous NP-complete problem important for analyzing digital hardware;
- or *halting problem* – a famous undecidable problem.

Many programs are supposed to finish

And many of them are important.

- Device drivers (a set of procedures that react to events).
- GPU programs (load program – provide input – wait – collect result).
- And many more (a recent Microsoft Azure outage is attributed to a non-termination of bug).

Does my program print “Hello world”?

If you can prove that a program finishes, you can prove that it does something good eventually.

```
print "Hello world"
```

Does my program print “Hello world”?

If you can prove that a program finishes, you can prove that it does something good eventually.

```
print "Hello world"
```

```
while True:
  do something
print "Hello world"
```

Does my program print “Hello world”?

```
days = // days since 1 Jan 1980
year = 1980
while days > 365:
  if year is leap:
    if days > 366:
      days = days - 366
      year = year + 1
  else:
    days = days - 365
    year = year + 1
print "Hello world"
```

Does my program print “Hello world”?

```
days = // days since 1 Jan 1980
year = 1980
while days > 365:
  if year is leap:
    if days > 366:
      days = days - 366
      year = year + 1
  else:
    days = days - 365
    year = year + 1
print "Hello world"
```

- Based on a bug that froze many Zune devices on 31 Dec 2008. The official response was, “Wait until battery dies”.
- People are bad at finding such bugs; machines can be good.
- But the problem is hard.

Programs are (not) Turing machines

We do not see programs as Turing machines, we think of *transition systems* instead, i.e.,

- a program has states (configurations): which line we're executing and what are the values of variables;
- and transitions (*transition relation*): how can we move between states.

Programs are (not) Turing machines

We do not see programs as Turing machines, we think of *transition systems* instead, i.e.,

- a program has states (configurations): which line we're executing and what are the values of variables;
- and transitions (*transition relation*): how can we move between states.

Two general approaches

- Variables are finite strings of bits, and operations are logical. Then, finite number of configurations, and problems reduce to decidable boolean satisfiability. But for programs, its model (description) might be huge.
- Variables are mathematical numbers. Then infinite numebr of configurations, undecidability translates, and you need more complicated math. But the models are smaller.

Does my program print “Hello world”?

```
x = // ask user for a positive number
while x ≠ 1:
  if x is even:
    x = x / 2
  else:
    x = 3*x + 1
print "Hello world"
```

Does my program print "Hello world"?

```
x = // ask user for a positive number
while x ≠ 1:
  if x is even:
    x = x / 2
  else:
    x = 3*x + 1
print "Hello world"
```

- If x is a natural number – then no one knows, but mathematicians suspect that it does (Collatz conjecture).
- Checked up to $5 \cdot 2^{60}$ (as Wikipedia claims).
- So, undecidability translates, but maths offers many tools for numbers (e.g., linear programming).

Soundness vs. completeness

Computer scientists are used to undecidable problems. They try to devise techniques that are

- *sound*: if we give a definite answer, it should be correct;
- may be *incomplete*: we are allowed to say that *we do not know*.

Soundness vs. completeness

Computer scientists are used to undecidable problems. They try to devise techniques that are

- *sound*: if we give a definite answer, it should be correct;
- may be *incomplete*: we are allowed to say that *we do not know*.

An undecidable problem will become a pair of complementary:

- *try* to prove that a program finishes (i.e., try to prove that there's no bug);
- *try* to prove that a program may not finish (i.e., try to find a bug).

Not all programs that finish can be proven to do so and vice versa. The idea it to handle as many interesting programs as possible. E.g., many interesting programs mostly use *linear arithmetic*.

What is a proof of that a program finishes?

A better studied problem

- Remember, computer scientists like simple things.
- For a state, find a bound on the number of steps (lines of code, loop iterations) until the program finishes.

What is a proof of that a program finishes?

A better studied problem

- Remember, computer scientists like simple things.
- For a state, find a bound on the number of steps (lines of code, loop iterations) until the program finishes.

Ranking function

Try to map every state to a value that

- cannot decrease forever (e.g., a natural number);
- decreases as the program runs;
- keyword: *well-founded relation*.

So, a ranking function

```
x = // ask user for a
    // positive number
k = // ask user
while x > 0:
  x = x + k
```

So, a ranking function

```
x = // ask user for a
    // positive number
k = // ask user
while x > 0:
  x = x + k
```

$$\begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \text{ and } k < 0 \end{cases}$$

So, a ranking function

```
x = // ask user for a
    // positive number
k = // ask user
while x > 0:
  x = x + k
```

$$\begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \text{ and } k < 0 \end{cases}$$

The ranking function is usually partial, so 2 aspects:

- what decreases;
- when it decreases (from what states our program finishes).

If there is a piecewise-linear ranking function, it will usually be found by modern techniques. They are good at finding linear relations between things, e.g., the number of steps and variables.

What is a proof that a program might never finish?

A less studied problem

Remember, two aspects.

- The *where*-aspect is more or less agreed upon (*recurrence set*).

What is a proof that a program might never finish?

A less studied problem

Remember, two aspects.

- The *where*-aspect is more or less agreed upon (*recurrence set*).

What is being researched.

- The *what*-aspect (some finite proof that there is an infinite execution).
- How to find recurrence sets (right heuristics are needed).
- In case of uncertainty, which choices lead to which behaviours (a hidden *when*-aspect).

What is a proof that a program might never finish?

A less studied problem

Remember, two aspects.

- The *where*-aspect is more or less agreed upon (*recurrence set*).

What is being researched.

- The *what*-aspect (some finite proof that there is an infinite execution).
- How to find recurrence sets (right heuristics are needed).
- In case of uncertainty, which choices lead to which behaviours (a hidden *when*-aspect).

And a bit on what do I do

Just submitted a conference paper on (just another approach to) *what can be a proof*.

A few words on the research area

- The questions and the mathematical foundations of techniques are not new.
- But the important advances are quite recent.
- The problem is chalenging: need to produce many things at the same time: what, where, when.
- Useful on its own to debug critical programs, but also other problems reduce to (non-)termination.
- Much is to be done, and every small advancement is appreciated.

A few words on the research area

- The questions and the mathematical foundations of techniques are not new.
- But the important advances are quite recent.
- The problem is chalenging: need to produce many things at the same time: what, where, when.
- Useful on its own to debug critical programs, but also other problems reduce to (non-)termination.
- Much is to be done, and every small advancement is appreciated.

Thanks