In search of software perfection

Xavier Leroy

2019-08-21

Collège de France and Inria

A formative experience (Jan 1988)

— Your 100 000 lines of code embedded in Ariane 4…
Are you sure there are no bugs?

— Sir! We tested them very carefully!

Second formative experience (Spring 1988)

— I'm looking for a summer internship in systems programming or maybe in compilation.

— Well, I know a language that could use more compilation work. It's called CAML.

Program proof

Verification of high-assurance software

Mostly code reviews and lots of tests.

Limitations:

- Incomplete: cannot explore all possible behaviors of the program.
  Testing shows the presence, not the absence of bugs.
  E. W. Dijkstra, 1969
- Expensive: writing and validating the test suite against the specifications is hugely expensive at the highest assurance levels.

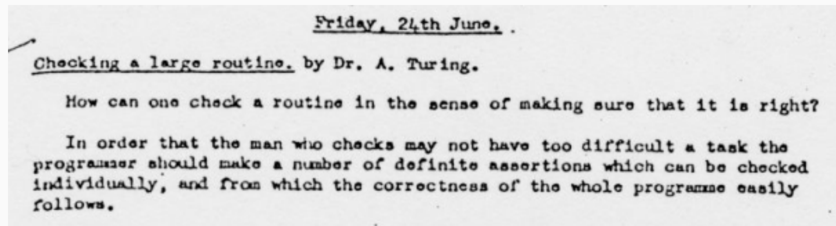
Formal verification

Using computation and deduction, establish properties that hold of all possible executions of the program.

Properties range from robustness (no crashes) to full correctness (w.r.t. specifications).

An old idea

Alan Turing, *Checking a large routine*, **1949**.


Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Talk given at the inaugural conference of the EDSAC computer, Cambridge University, June 1949. The manuscript was corrected, commented, and republished by F.L. Morris and C.B. Jones in *Annals of the History of Computing*, 6, 1984.

Turing's "large routine"

Compute n! using additions only.

Two nested loops.

```
int fac (int n)
{
  int s, r, u, v;
  u = 1;
  for (r = 1; r < n; r++) {
    v = u; s = 1;
    do {
      u = u + v;
    } while (s++ < r);
  }
  return u;
}
```

Turing's "large routine"

No structured programming in 1949; just flowcharts.

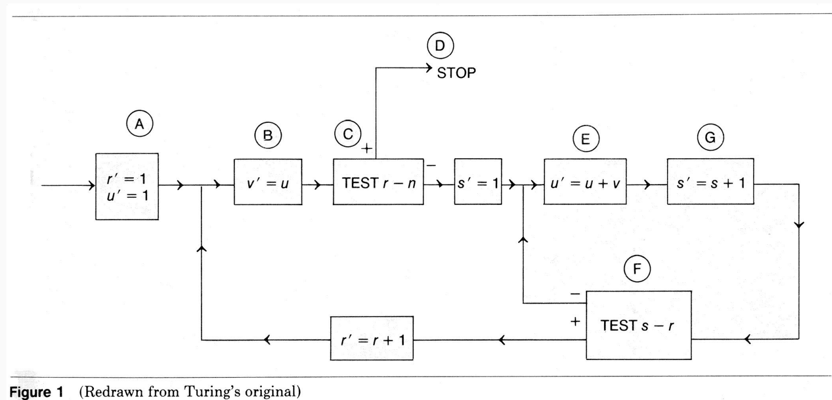


Figure 1 (Redrawn from Turing's original)

Turing's genius idea

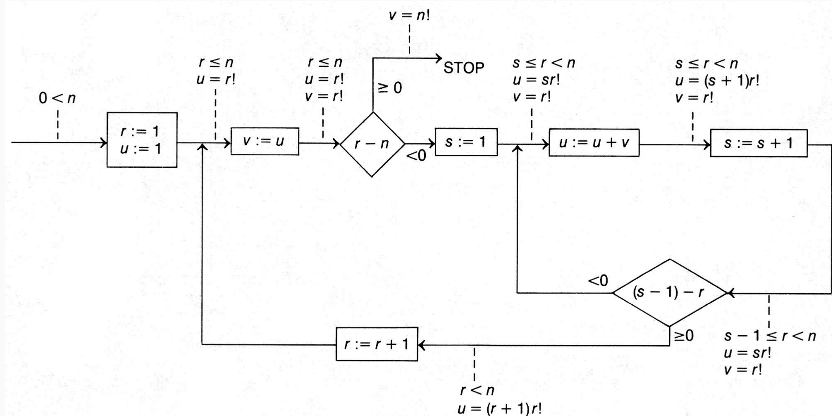
Every program point is associated with a logical invariant: a relation between values of variables that hold in every execution.

STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
27					s	$s + 1$	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		$\lfloor r$	$\lfloor r$		$s \lfloor r$	$(s + 1) \lfloor r$	$(s + 1) \lfloor r$
31			$\lfloor r$	$\lfloor n$	$\lfloor r$	$\lfloor r$	$\lfloor r$
	TO Ⓑ WITH $r' = 1$ $u' = 1$	TO Ⓒ	TO Ⓓ IF $r = n$ TO Ⓔ IF $r < n$		TO Ⓖ	TO Ⓑ WITH $r' = r + 1$ IF $s \geq r$ TO Ⓔ WITH $s' = s + 1$ IF $s < r$	TO Ⓕ

Figure 2 (Redrawn from Turing's original)

Turing's genius idea

In more modern notation:



To verify the program, it's enough to check that each assertion logically implies the assertions at successor points.

The next 60 years

- **1967** R. Floyd, *Assigning meanings to programs*. Reinvents and generalizes Turing's idea.
- **1969** C. A. R. Hoare, *An axiomatic basis for computer programming*. A logic $\{P\}\ c\ \{Q\}$ to reason about structured programs.
- **1970–2000** General conviction: not usable in practice.
- **1976–1980** Restricted, more automatic approaches: abstract interpretation, model checking.
- **circa 2000** Much progress in automated theorem proving (SMT).
- **mid 2000** Practically-usable tools for program proof.

Frama-C WP demo

Programming with a proof assistant

Propositions as Types, Proofs as Programs

Curry (1958) observes and Howard (1969) studies in more details a beautiful correspondence between a calculus and a logic:

simply-typed λ-calculus	intuitionistic logic
type	proposition
term (program)	proof (“construction”)
reduction (execution)	cut elimination (normalization)

Unified frameworks for computation and proof

Generalizing the Curry-Howard correspondence:

- Martin-Löf type theory (1972–1980) ($\rightsquigarrow$ Agda)
- Coquand and Huet's Calculus of Constructions (1985) ($\rightsquigarrow$ Coq, Lean)

Based on lambda-calculus + dependent types (Π, Σ) + stratification in universes.

Provide highly expressive frameworks for computation and proofs.

Another approach to program proof

If we write programs in such a dependently-typed lambda-calculus, we will be able to reason about programs directly inside the logic.

No program logic is needed to mediate between programs and logical propositions if the functions and the data structures of the program are functions and objects of the mathematical logic already!

Contrasting the two approaches

Frama-C style: distinguish between computational functions (`strlen`) and logical functions (`length`), often axiomatized.

```
/*@ logic integer length(const char * s);
  @ axiom length_0:
      ∀ s; valid_string(s) ==> s[length[s]] == 0;
  @ axiom length_1:
      ∀ s, i; valid_string(s) /\ 0 <= i < length[s] ==> s[i] != 0;
  @*/
```

Computational functions are specified using logical functions.

```
/*@ requires valid_string(s);
  @ ensures \result == length[s];
  @*/
size_t strlen(const char * s) { ... }
```

Contrasting the two approaches

Coq-style: the same functions can be used in computations and in theorems.

```
Fixpoint length(l: list A) : nat :=
  match l with nil => O | h :: t => S (length l) end.

Definition combine(l1 l2: list A) : option (list A) :=
  if length l1 =? length l2 then Some (zip l1 l2) else None.

Theorem length_map:
  forall f l, length (map f l) = length l.
```

A requirement: hyperpure functional programming

When programming in a proof assistant, we must program in "hyperpure" functional style:

- No imperative features
  ($\Rightarrow$ persistent data structures, monads, etc)
- All functions must provably terminate.

(Haskell is not hyperpure; F* is because nontermination is a monadic effect.)

Coq demo

Is software perfection within reach?

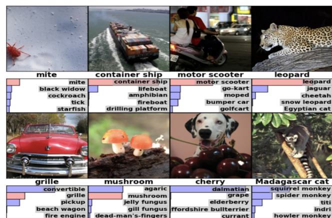
Is software perfection within reach?

Program proof and mechanized logics are a huge step forward.

They reduce the problem of trusting the program to that of trusting its formal specifications.

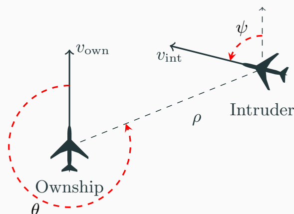
- Formal specifications must be available.
  (Control-command applications: OK; Web applications: ???)
- Formal specifications should be as clear and simple as possible.
- Formal specifications must be reviewed and tested.
  (Executable specs a plus.)

Image classification



No specification

ACAS-Xu collision avoidance



Geometric specification

Formal verification: G. Katz et al, 2017

Some other limitations

Hardware is not as perfect as we software people like to assume.
(Skylake HT bug, Rowhammer, Meltdown, Spectre, ...)

Specification languages are in their infancy.
(Domain-specific specification languages?)

We teach logic badly in maths and CS courses.

Try and prove your programs.

They will thank you for that.