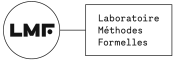
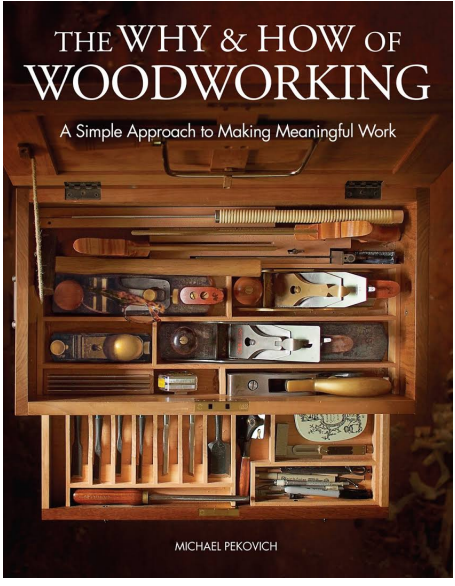
The Why and How of Deductive Program Verification

Jean-Christophe Filliâtre
CNRS

Marktoberdorf Summer School 2026

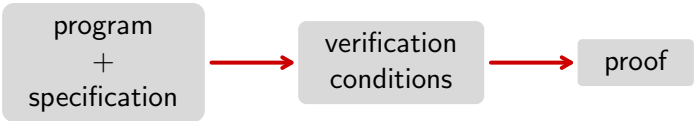
lecture 1



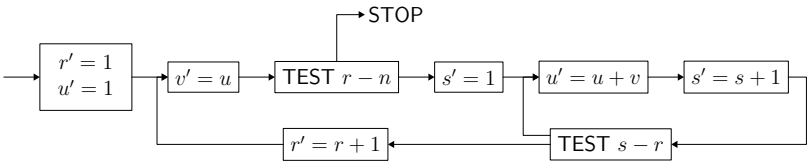

THE WHY & HOW OF
WOODWORKING
A Simple Approach to Making Meaningful Work
MICHAEL PEKOVICH

https://usr.lmf.cnrs.fr/~jcf/marktoberdorf-2026/

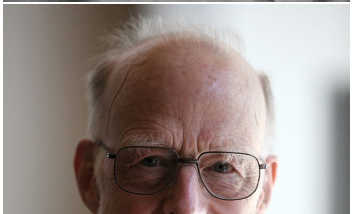
(also on Discord)


program
+
specification
verification
conditions
proof

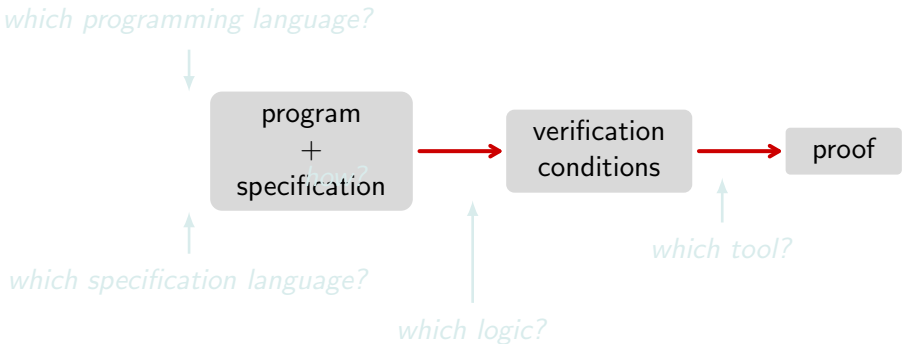
A. M. Turing. Checking a large routine. 1949.

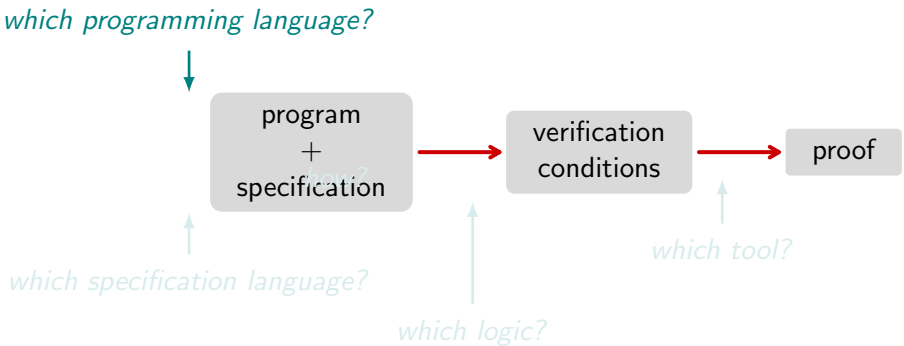


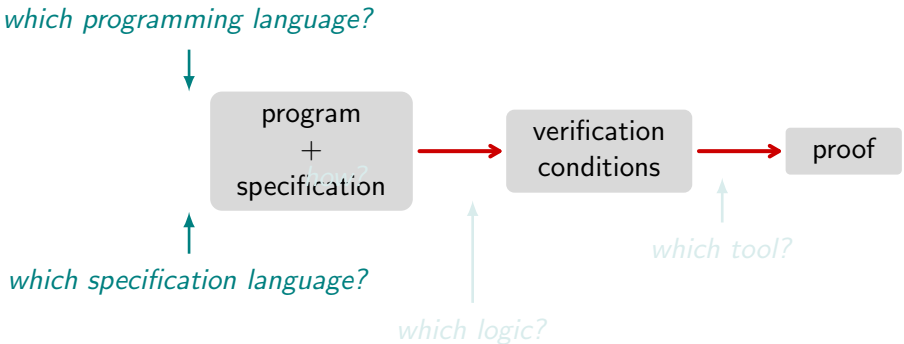
Robert Floyd.
Assigning Meanings to Programs.
1967

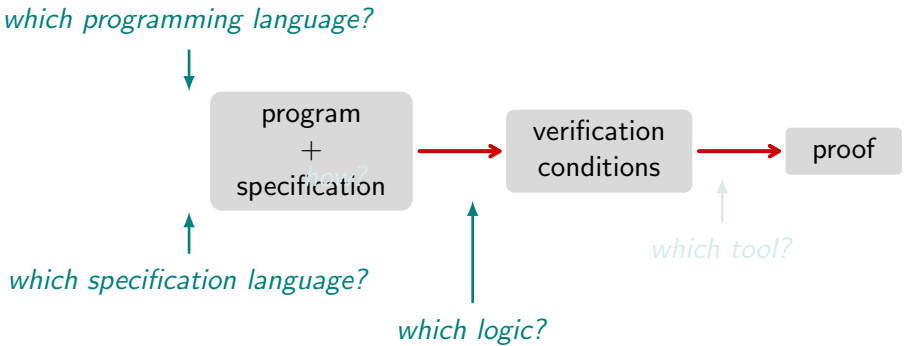


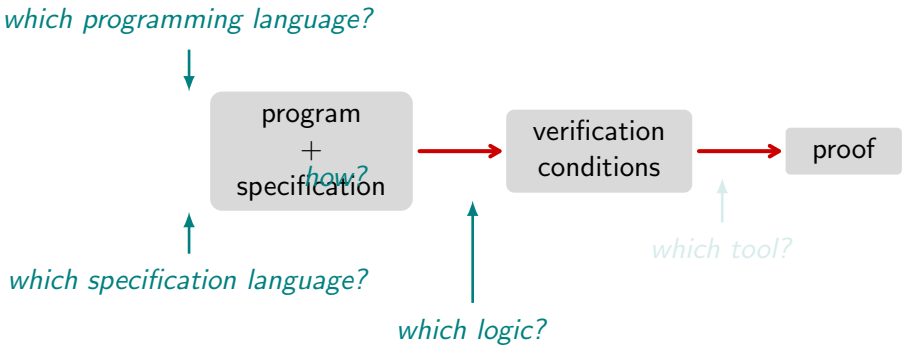
Tony Hoare.
An Axiomatic Basis for Computer Programming.
1969

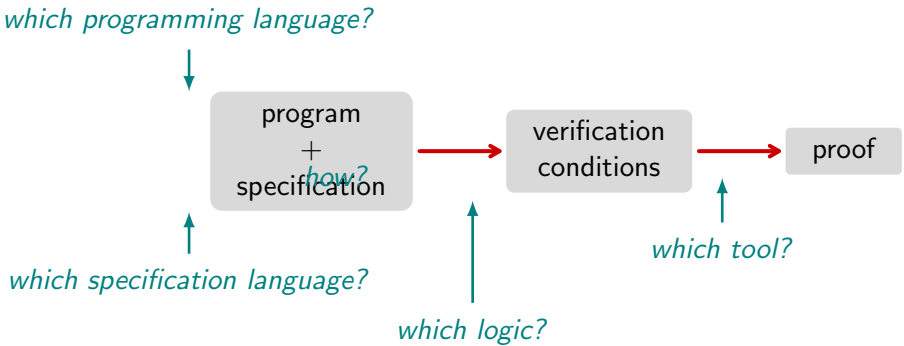

which programming language?
program
+
specification
verification
conditions
proof
which specification language?
which logic?
which tool?

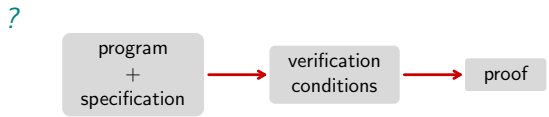

which programming language?
program
+
specification
verification
conditions
proof
which specification language?
which logic?
which tool?


which programming language?
program
+
specification
verification
conditions
proof
which specification language?
which logic?
which tool?


which programming language?
program
+
specification
verification
conditions
proof
which tool?
which specification language?
which logic?


which programming language?
program
+
specification
how?
verification
conditions
proof
which tool?
which specification language?
which logic?


which programming language?
program
+
specification
how?
verification
conditions
proof
which tool?
which specification language?
which logic?



tackling the deductive verification of an existing programming language, such C or Rust, is a daunting task

instead we use intermediate verification languages, e.g.

- Boogie
- Why3 ← *this lecture*
- Viper
- Coma

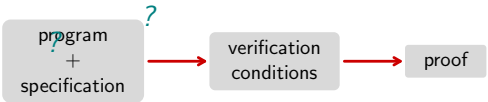
which specification language?

plenty of options

- first-order logic
- higher-order logic
- type theory
- separation logic
- dynamic logic
- etc.

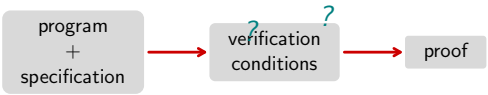
Why3 uses an extension of first-order logic

how to compute verification conditions?

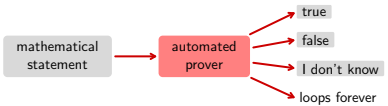


we can use weakest preconditions

we will explain that in detail tomorrow



we use automated theorem provers



Why3 supports 25+ of them

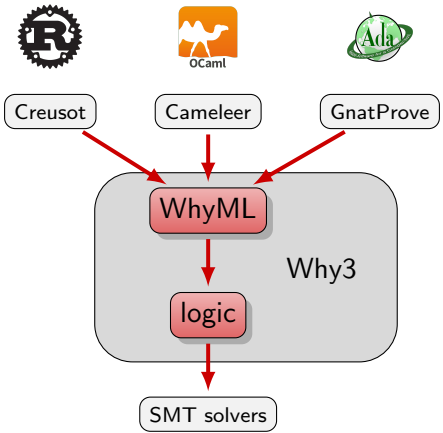
among them are SMT solvers, such as Z3, CVC5, Alt-Ergo, Colibri 2, etc.

SMT solvers combine

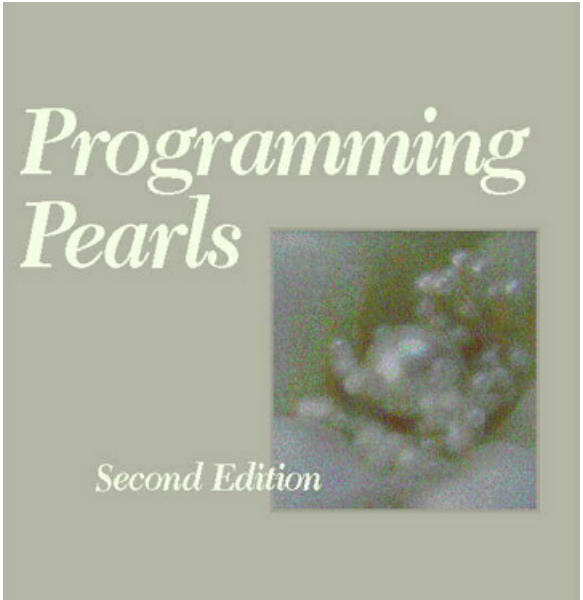
- propositional logic
- quantifiers
- builtin theories
  - uninterpreted functions
  - equality
  - integer arithmetic
  - theory of arrays
  - etc.

$$\frac{\begin{array}{l} f : \mathbb{Z} \to \mathbb{Z} \\ \forall x.\, f(x) > f(x-1) + 1 \\ f(0) = 40 \vee f(1) = 40 \end{array}}{\exists y.\, f(y) > 42}$$

such a combination is undecidable, which means heuristics and incompleteness


OCaml
Ada
Creusot
Cameleer
GnatProve
WhyML
Why3
logic
SMT solvers

today	a short demo of Why3
lecture 2	from programs to provers
lecture 3	programs to the rescue of logic
lecture 4	a case study

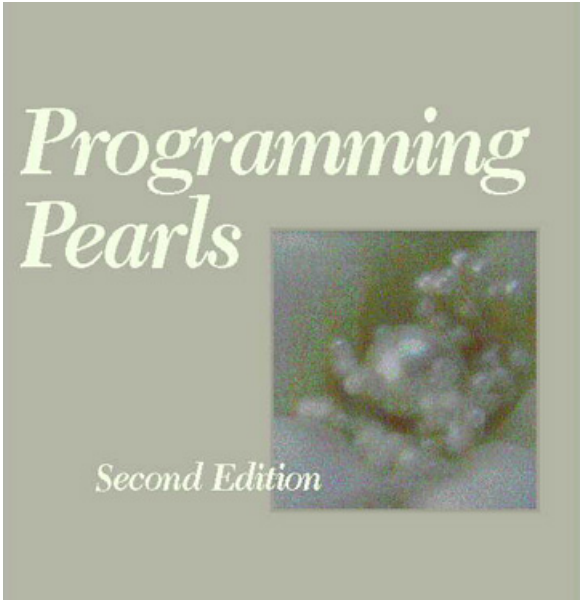


8. Algorithms Design Techniques

maximum subarray problem

2	2	-5	4	-1	2	1	-5	4

skip



8. Algorithms Design Techniques

maximum subarray problem

2	2	-5	4	-1	2	1	-5	4

6

skip

```
use int.Int          import integers and arrays
use array.Array      from the standard library
```

a program function — *returns an integer* `s`

```
let maximum_subarray (a: array int) : (s: int)
= ...
```

takes an array `a` *as parameter*

```
let maximum_subarray (a: array int) : (s: int)
= (* the maximum in a[0..i[ *)
  let ref maxsum = 0 in


  (* the maximum ending on i *)
  let ref curmax = 0 in

  for i = 0 to length a - 1 do




    curmax <- curmax + a[i];
    if curmax < 0 then ( curmax <- 0; );
    if curmax > maxsum then ( maxsum <- curmax; )
  done;
  return maxsum
```

```
use array.ArraySum          import logical function sum
                            from the standard library
```

$$\texttt{sum a lo hi} \stackrel{\text{def}}{=} \sum_{\texttt{lo} \leqslant i < \texttt{hi}} \texttt{a}[i]$$

included *excluded*

(more on Friday)

a few things to discuss and to agree on:

- is the array a allowed to be empty?
- is an empty subarray (with sum 0) a valid answer?

here is a proposed specification:

```
let maximum_subarray (a: array int) : (s: int)
  ensures { forall l h. 0 <= l <= h <= length a -> sum a l h <= s }
  ensures { exists l h. 0 <= l <= h <= length a /\ sum a l h  = s }
= ...
```

we have to agree on this specification

otherwise, the whole proof is a waste of time

```
let maximum_subarray (a: array int) : (s: int)
= (* the maximum in a[0..i[ is a[lo..hi[ *)
  let ref maxsum = 0 in
  let ghost ref lo = 0 in
  let ghost ref hi = 0 in
  (* the maximum ending on i is a[cl..i[ *)
  let ref curmax = 0 in
  let ghost ref cl = 0 in
  for i = 0 to length a - 1 do




    curmax <- curmax + a[i];
    if curmax < 0 then ( curmax <- 0; cl <- i+1 );
    if curmax > maxsum then ( maxsum <- curmax; lo <- cl; hi <- i+1 )
  done;
  return maxsum
```

```
let maximum_subarray (a: array int) : (s: int)
= (* the maximum in a[0..i[ is a[lo..hi[ *)
  let ref maxsum = 0 in
  let ghost ref lo = 0 in
  let ghost ref hi = 0 in
  (* the maximum ending on i is a[cl..i[ *)
  let ref curmax = 0 in
  let ghost ref cl = 0 in
  for i = 0 to length a - 1 do
    invariant { forall l. 0 <= l <= i -> sum a l i <= curmax }
    invariant { 0 <= cl <= i /\ sum a cl i = curmax }
    invariant { forall l h. 0 <= l <= h <= i -> sum a l h <= maxsum }
    invariant { 0 <= lo <= hi <= i /\ sum a lo hi = maxsum }
    curmax <- curmax + a[i];
    if curmax < 0 then ( curmax <- 0; cl <- i+1 );
    if curmax > maxsum then ( maxsum <- curmax; lo <- cl; hi <- i+1 )
  done;
  return maxsum
```

modify function `maximum_subarray` so that empty segments are not allowed anymore

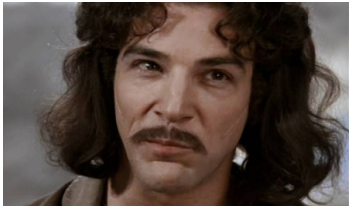
you have to adapt

- the function contract
- the code
- the loop invariants

takeaways

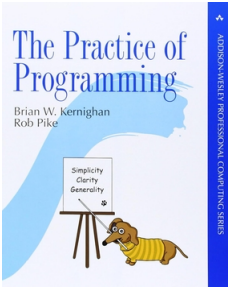
a specification must be validated by a human

I do not think it means what you think it means



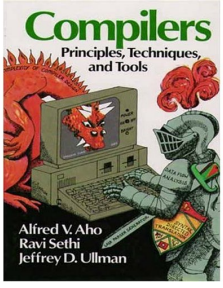
in verification, good practices are important

this is not different from programming



deductive verification benefits from intermediate languages

this is not different from compilers



thank you