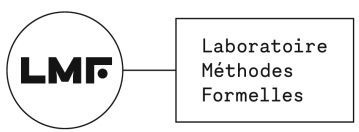
Why3, une plateforme pour la vérification déductive

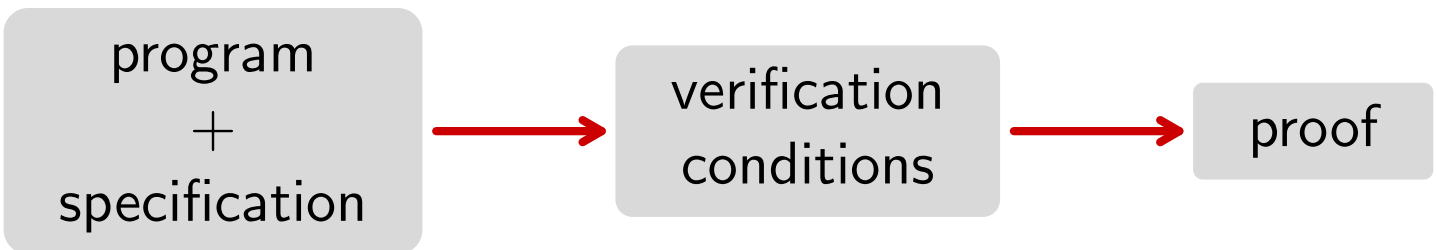
Jean-Christophe Filliâtre

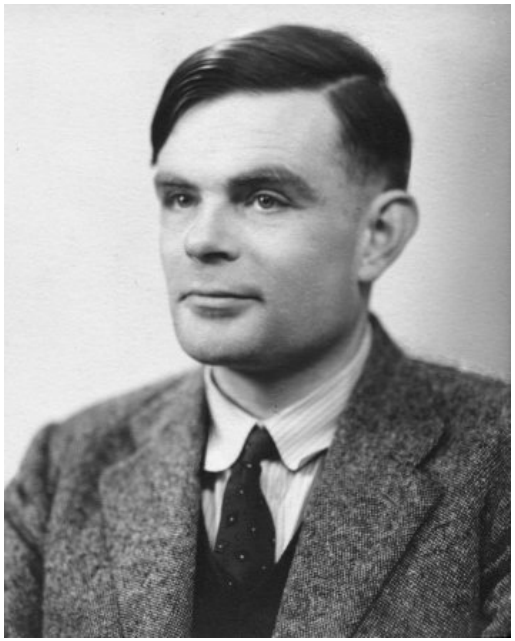
Why3, une plateforme pour la vérification déductive

Jean-Christophe Filliâtre

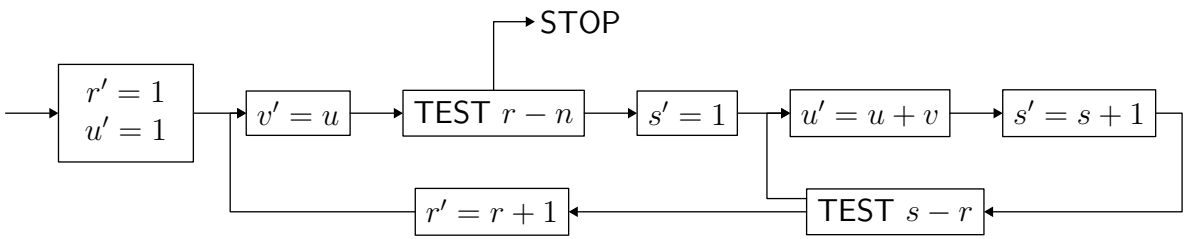
Journées FAC
Toulouse, 6 avril 2023



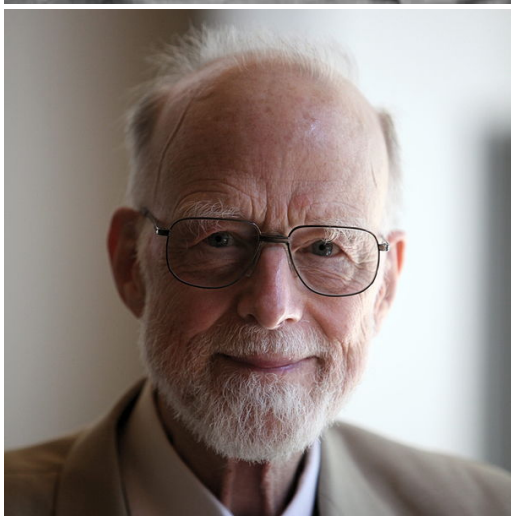

program
+
specification
verification
conditions
proof



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

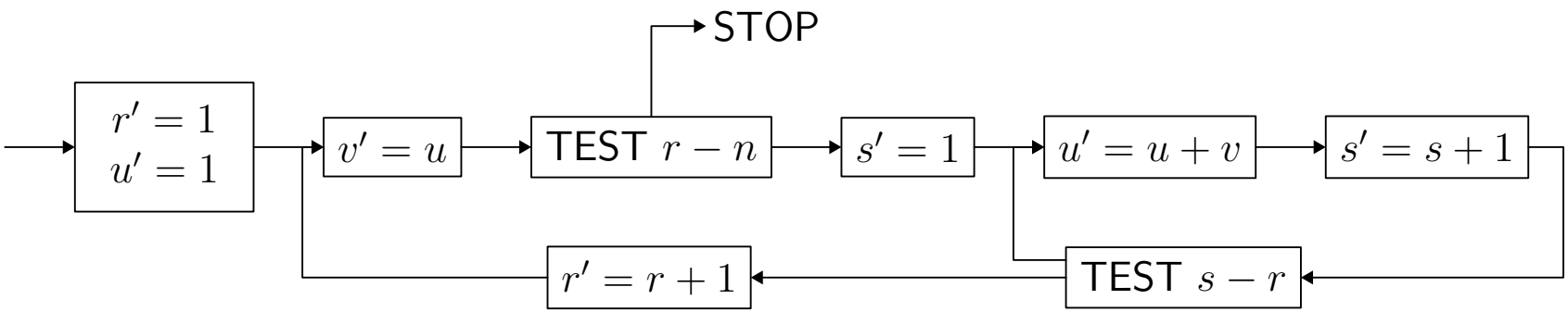


Robert Floyd.
Assigning Meanings to Programs.
1967

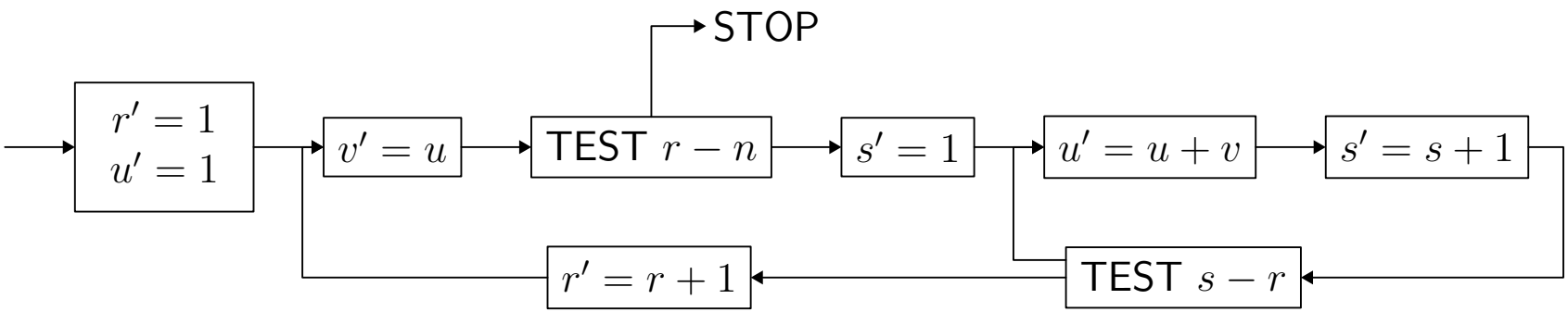


Tony Hoare.
An Axiomatic Basis for Computer Programming.
1969

checking a large routine (Turing, 1949)

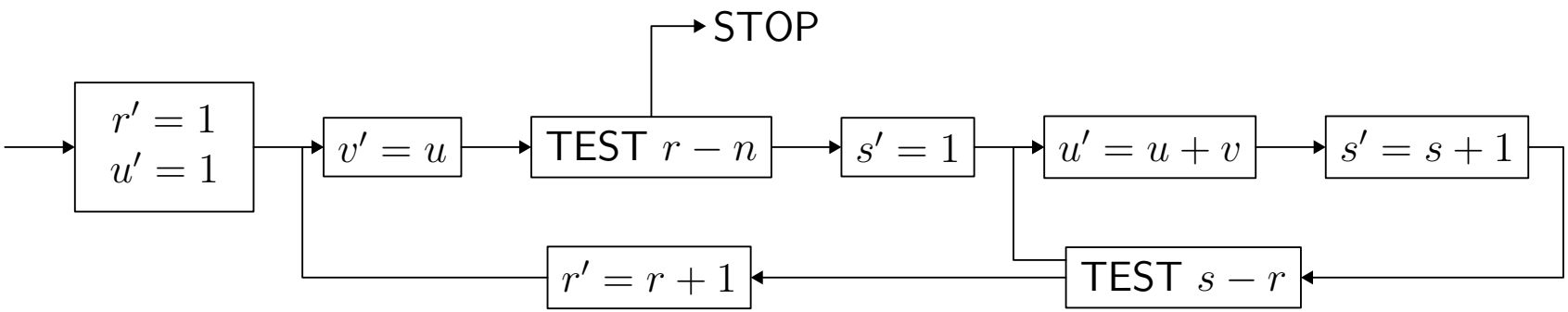


checking a large routine (Turing, 1949)



$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$

checking a large routine (Turing, 1949)



precondition $\{n \geq 0\}$

$u \leftarrow 1$

`for` $r = 0$ `to` $n - 1$ `do`

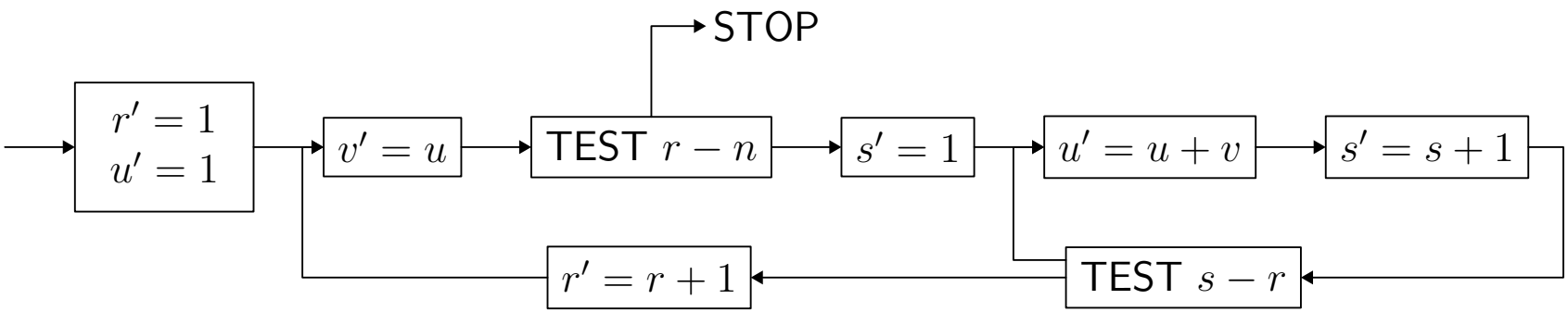
  $v \leftarrow u$

  `for` $s = 1$ `to` r `do`

    $u \leftarrow u + v$

postcondition $\{u = n!\}$

checking a large routine (Turing, 1949)



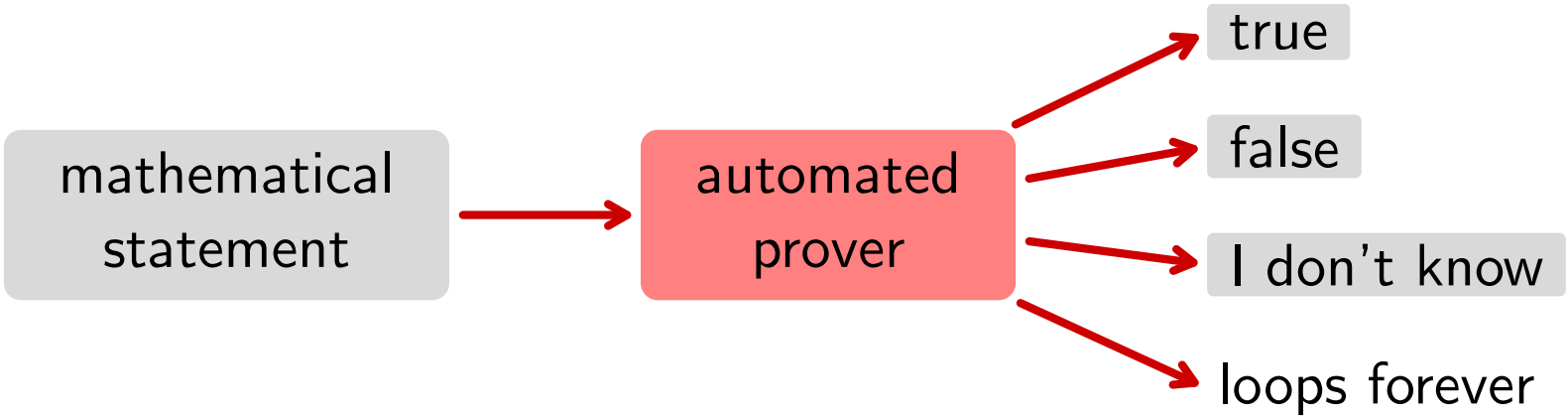
precondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do` invariant $\{u = r!\}$
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do` invariant $\{u = s \times r!\}$
    $u \leftarrow u + v$
postcondition $\{u = n!\}$

```
forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = n!) /\
  (0 <= n - 1 ->
    1 = 0! /\
    (forall u:int.
      (forall r:int. 0 <= r /\ r <= n - 1 -> u = r! ->
        (1 > r -> u = (r + 1)!) /\
        (1 <= r ->
          u = 1 * r! /\
          (forall u1:int.
            (forall s:int. 1 <= s /\ s <= r -> u1 = s * r! ->
              (forall u2:int.
                 u2 = u1 + u -> u2 = (s + 1) * r!)) /\
            (u1 = (r + 1) * r! -> u1 = (r + 1)!)))) /\
      (u = ((n - 1) + 1)! -> u = n!)))
```

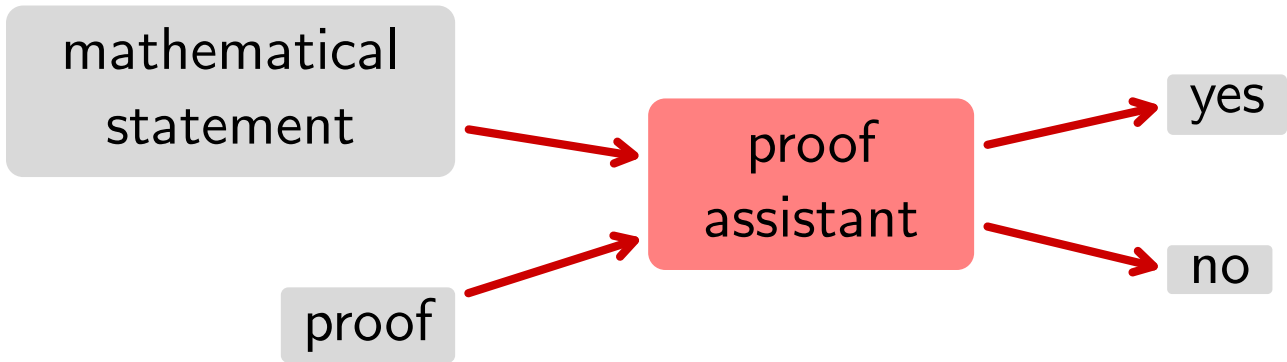
what do we do with this mathematical statement?

we could perform a manual proof (as Turing and Hoare did)
but it is long, tedious, and error-prone

so we turn to tools that mechanize mathematical reasoning



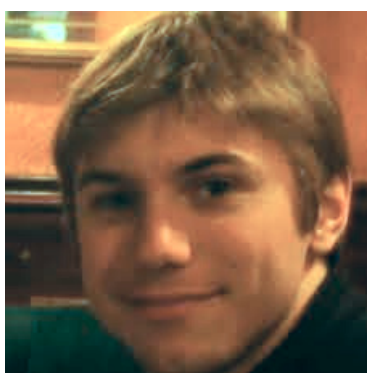
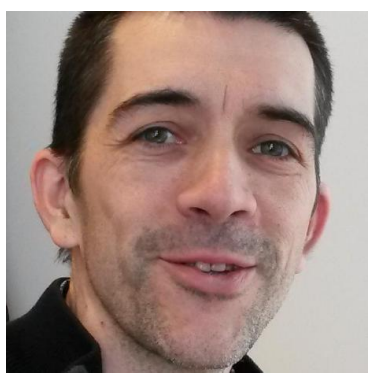
examples: Z3, CVC5, Alt-Ergo, Vampire, SPASS, etc.

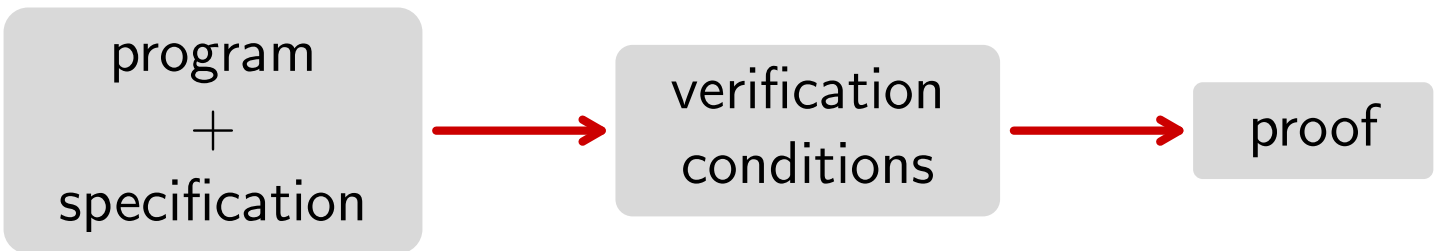


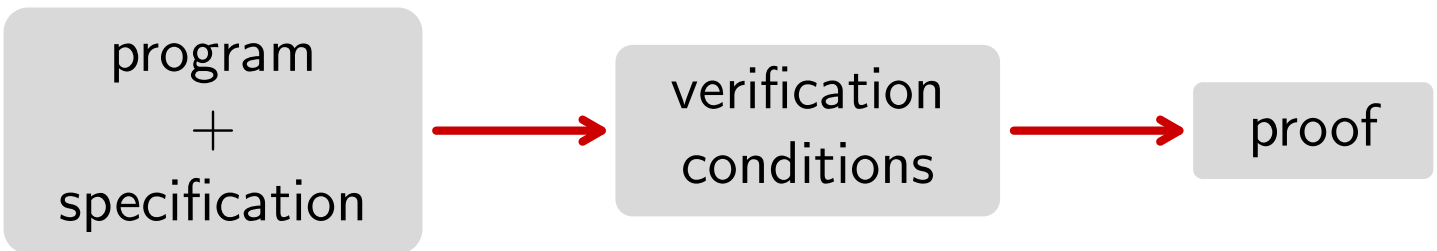
examples: Coq, Isabelle, PVS, HOL Light, etc.

a tool for deductive program verification
that leverages automated and interactive provers

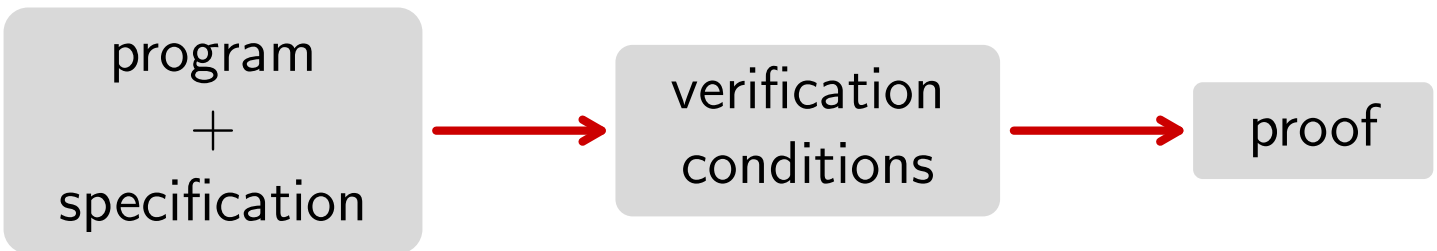
- 1999: prototype inside Coq
- 2001: standalone tool, with Coq, PVS, and Simplify provers
- 2009: new implementation, Why3

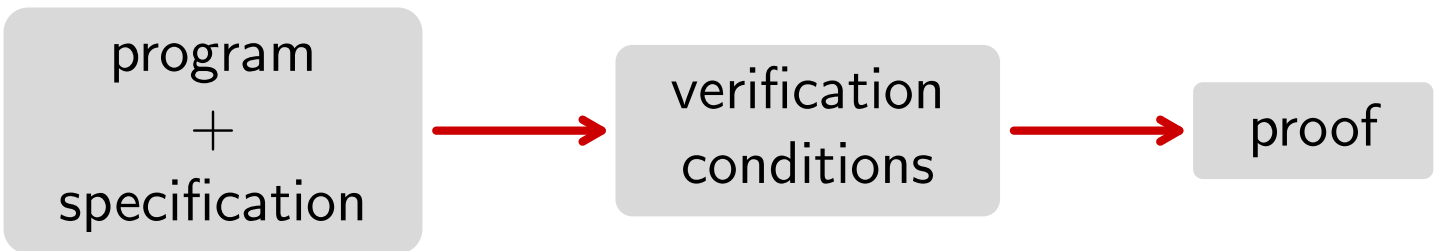


program
+
specification
verification
conditions
proof



- language design

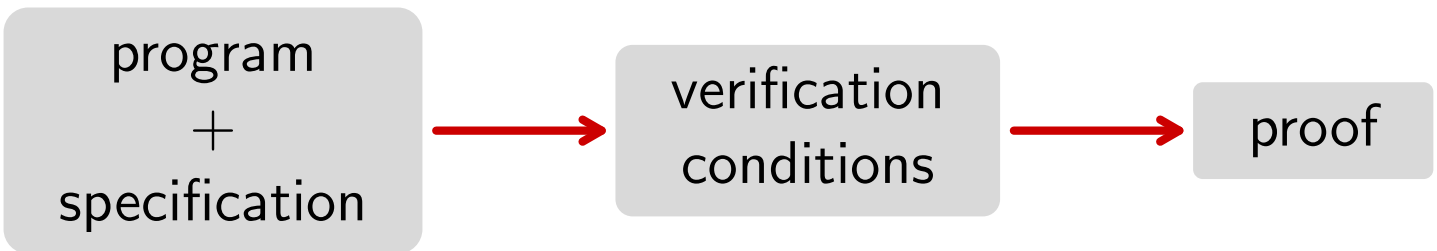


- language design
- logic design



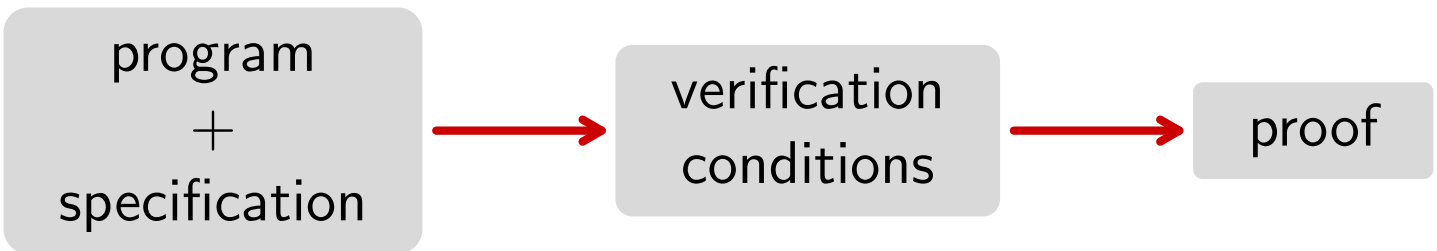
- language design
- logic design

- efficiency



- language design
- logic design

- efficiency
- intermediate language

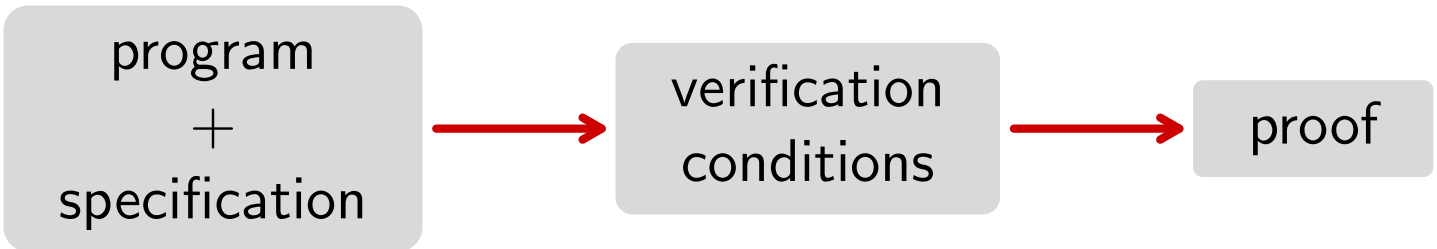


- language design
- logic design

- efficiency
- intermediate language

- support many provers

what do we do

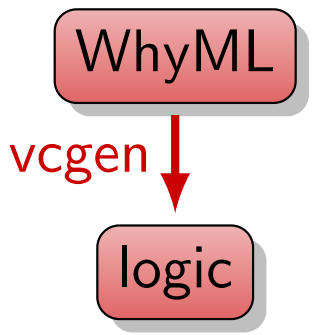


- language design
- logic design

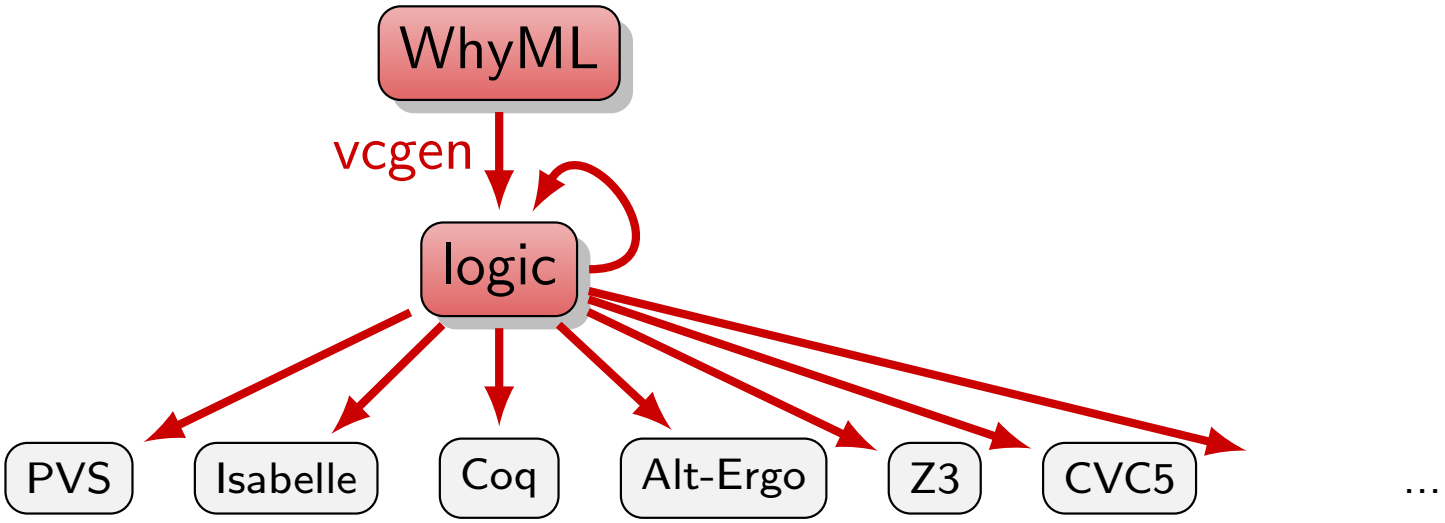
- efficiency
- intermediate language

- support many provers
- use them well

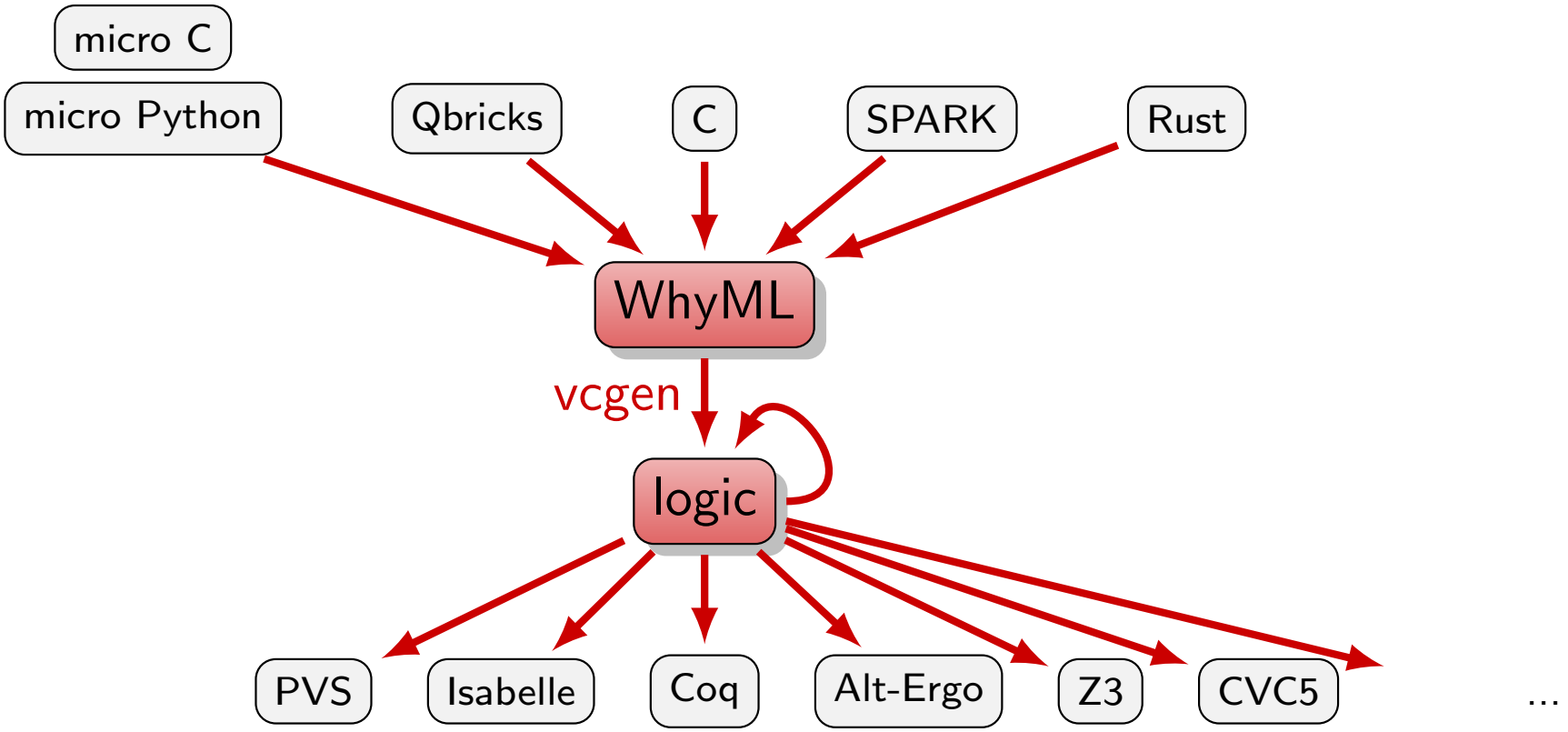
1. overview of Why3
2. three focuses
   - first things first
   - termination
   - ghost code
3. live demo
   - a small puzzle


WhyML
vcgen
logic

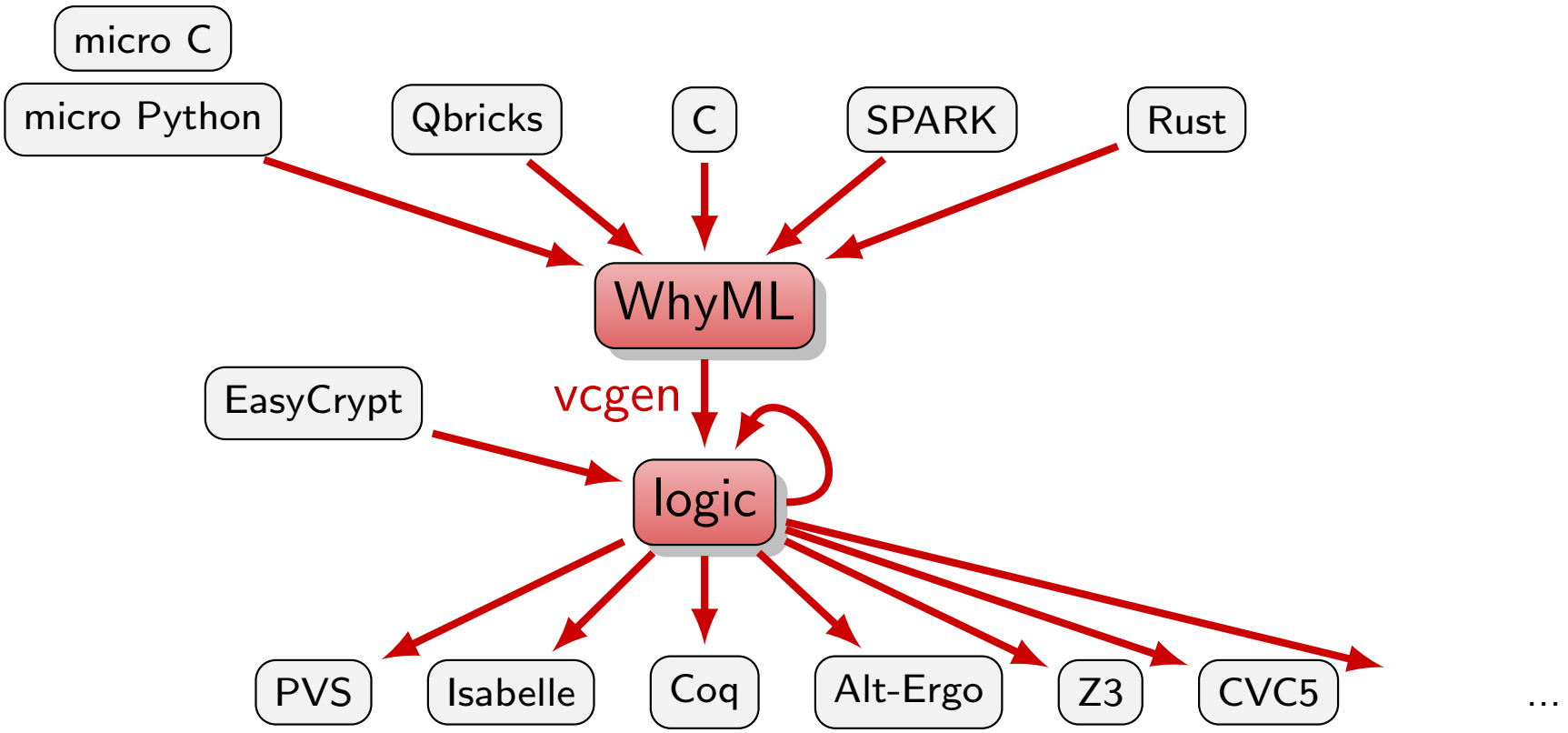
Why3, a versatile tool



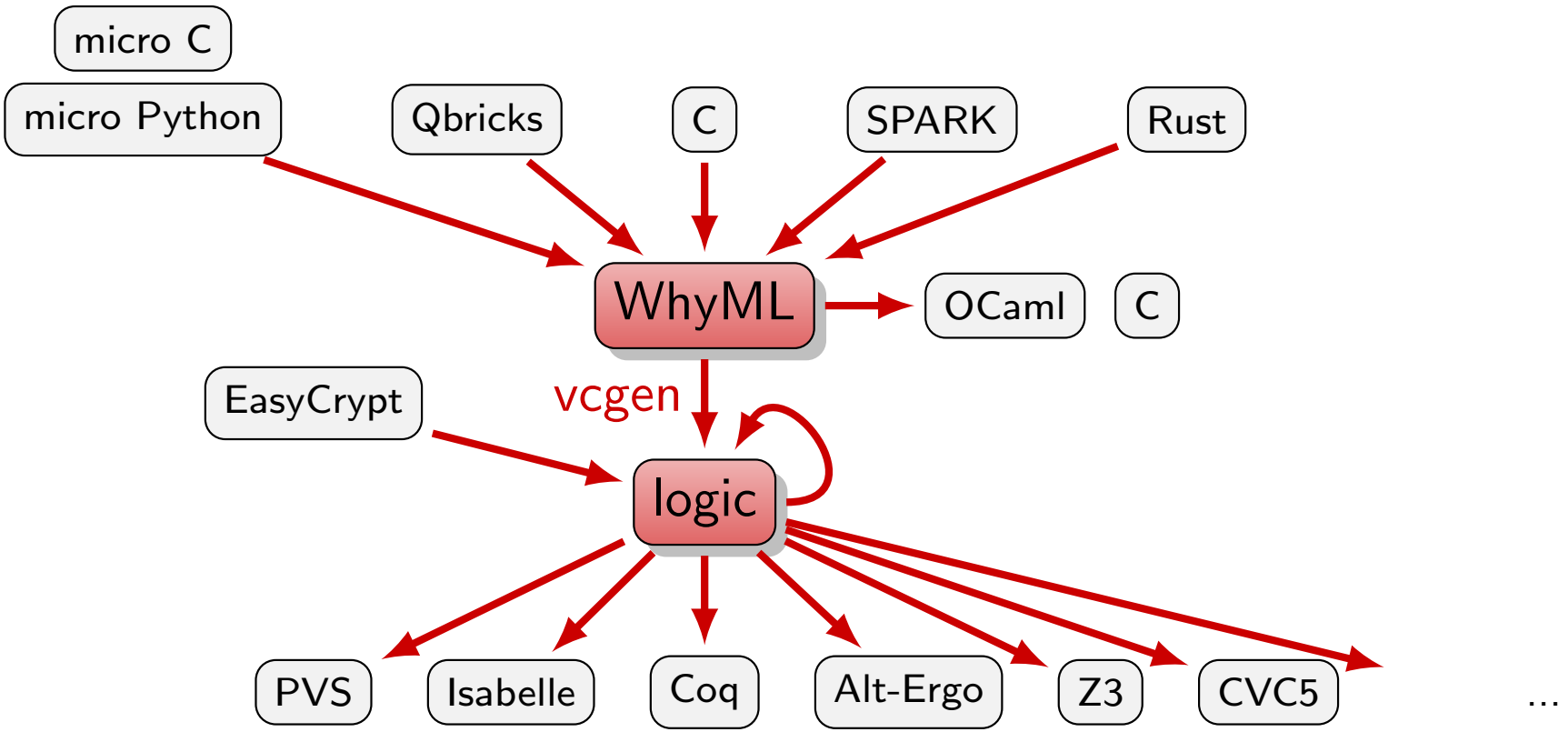
Why3, a versatile tool



Why3, a versatile tool



Why3, a versatile tool



WhyML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- `break`, `continue`, and `return`
- ghost code and ghost data
- mutable data with controlled aliasing
- contracts • loop and type invariants

WhyML, a specification language

- polymorphism & algebraic types
- limited support for higher order
- inductive predicates
- recursive definitions

example (Okasaki's random access lists)

```
let rec lookup (i: int) (l: ral 'a) : 'a
  requires { 0 <= i < length (elements l) }
  ensures  { nth i (elements l) = Some result }
  variant  { i, l }
  =
  match l with
  | Empty    -> absurd
  | One x s  -> if i=0 then x else lookup (i-1) (Zero s)
  | Zero s   -> let (x0, x1) = lookup (i/2) s in
                if i%2 = 0 then x0 else x1
  end
```

for 200+ more examples, see
https://toccata.gitlabpages.inria.fr/toccata/gallery/

Why3, a program verification tool

- VC generation using WP or fast WP
- 70+ VC *transformations* ($\approx$ tactics)
- support for 25+ ATP and ITP systems
- GUI for autoactive verification
- recorded proof sessions, batch replay
- Why3 in your browser

three focuses

I do not think it means what you think it means

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { ... r is an index where v appears,
                 or -1 is there is no such index ... }
```

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { (0 <= r < length a -> a[r] = v)
    /\ (r = -1 -> forall i. 0 <= i < length a -> a[i] <> v) }
```

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { (0 <= r < length a -> a[r] = v)
    /\ (r = -1 -> forall i. 0 <= i < length a -> a[i] <> v) }
```

the program can return -2 and yet be proved correct!

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { -1 <= r < length a
    /\ 0 <= r < length a -> a[r] = v
    /\ r = -1 -> forall i. 0 <= i < length a -> a[i] <> v }
```

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { -1 <= r < length a
    /\ 0 <= r < length a -> a[r] = v
    /\ r = -1 -> forall i. 0 <= i < length a -> a[i] <> v }
```

the program can return 42 and yet be proved correct!
(missing parentheses)

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { 0 <= r < length a /\ a[r] = v
    \/ r = -1 /\ forall i. 0 <= i < length a -> a[i] <> v }
```

this time this is fine!

before you do any proof, get the specification right

then have the reader agree with you on the spec

otherwise, the whole proof is a waste of time

write and verify some client code

```
let test () =
  let a = Array.init 5 (fun i -> i) in
  let r = binary_search a 0 in assert { r =  0 };
  let r = binary_search a 5 in assert { r = -1 };
  ...
```

using an exception to signal the unsuccessful search is much better

```
let binary_search (a: array int) (v: int) : (r: int)
  requires { ... the array a is sorted ... }
  ensures  { 0 <= r < length a /\ a[r] = v }
  raises   { Not_found -> forall i. 0<=i<length a -> a[i]<>v }
```

termination

Why3 logic is total

- recursive functions and predicates must terminate
- Why3 figures out a lexicographic structural variant

programs, however, may not terminate

- the user may supply variants
- Why3 tracks non-terminating code

depending on whether you supply a variant or not, you prove

partial correctness

if the precondition holds
and if the program terminates
then its postcondition holds

or

total correctness

if the precondition holds
then the program terminates
and its postcondition holds

partial correctness is a rather weak property,
since non-termination can turn your whole proof into something meaningless

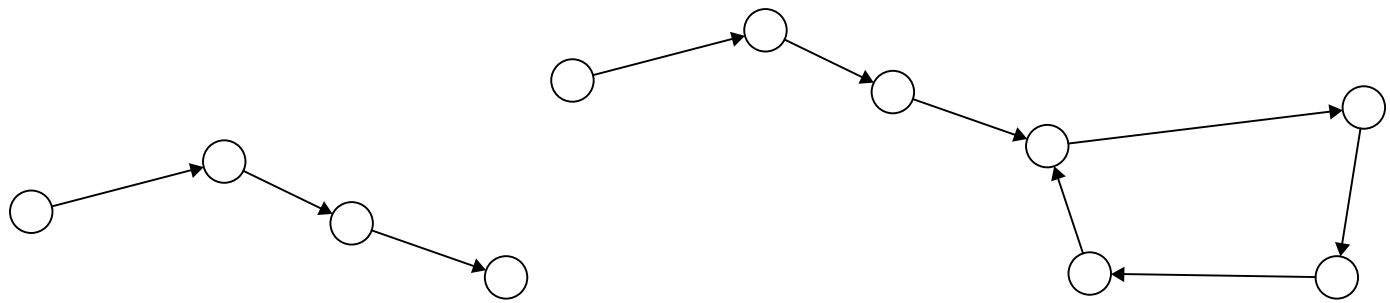
```
let oups ()
  diverges
  =
  while true do () done;
  ... this is ‘‘safe’’ ...
  assert { ... this is ‘‘verified’’ ... }
```

```
let rec f91 (n: int) : (r: int)


= if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

```
let rec f91 (n: int) : (r: int)

  variant { 100 - n }
= if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

```
let rec f91 (n: int) : (r: int)
  ensures { r = if n <= 100 then 91 else n - 10 }
  variant { 100 - n }
= if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

Floyd's tortoise and hare algorithm



defining the variant requires information we don't have
(that's precisely what the code is looking for)

ghost code

data and code added to the program
to make the proof simpler

we search the smallest Fibonacci number greater than n

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. fib i <= n < fib (i+1) = r }
=
  let ref a = 0 in
  let ref b = 1 in
  while b <= n do


    a, b <- b, a+b
  done;
  return b
```

we may want to introduce a loop invariant as follows

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. fib i <= n < fib (i+1) = r }
=
  let ref a = 0 in
  let ref b = 1 in
  while b <= n do
    invariant { exists i. i >= 0 /\
                    a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b
  done;
  return b
```

instead, we can keep track of the value of i with a ghost variable

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. fib i <= n < fib (i+1) = r }
=
  let ref a = 0 in
  let ref b = 1 in
  let ref i = 0 in
  while b <= n do
    invariant { i >= 0 /\ a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b;
    i <- i+1
  done;
  return b
```

instead, we can keep track of the value of *i* with a ghost variable

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. fib i <= n < fib (i+1) = r }
=
  let ref a = 0 in
  let ref b = 1 in
  let ghost ref i = 0 in
  while b <= n do
    invariant { i >= 0 /\ a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b;
    i <- i+1
  done;
  return b
```

- ghost code may read regular data but can't modify it
- ghost code cannot modify the control flow of regular code
- regular code does not see ghost data

consequence: ghost code can be removed
without observable modification

we do so when we translate WhyML to OCaml/C

suppose we want to prove that, for all n,

$$n! \geq 1$$

we can make a program that proves it

```
let lemma fact_pos (n: int) : unit
  requires { n >= 0 }
  ensures  { fact n >= 1 }
= let ref f = 1 in
  for i = 1 to n do
    invariant { f = fact (i-1) >= 1 }
    f <- i * f
  done;
  assert { f = fact n }
```

the whole program is ghost; we do not intend to run it

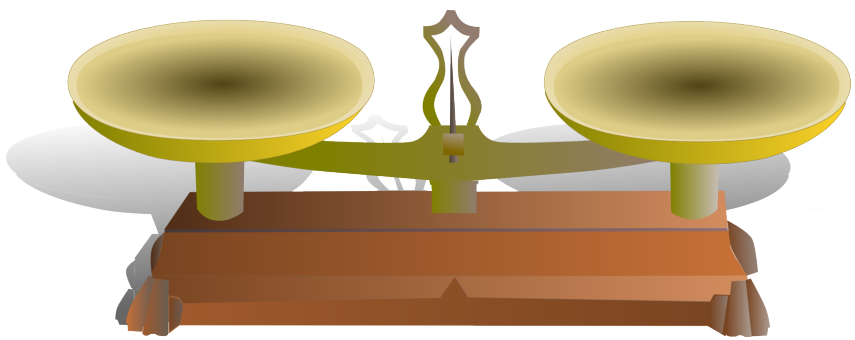
yet we can call it (within ghost code), for instance to get $42! \geq 1$

live demo: a small puzzle

---

you are given 8 balls
they have the same weight, apart from one, which is lighter

you are given a Roberval balance



using the balance at most twice, determine the lighter ball

- Guillaume Melquiond, Raphaël Rieu-Helft. WhyMP, a Formally Verified Arbitrary-Precision Integer Library. JSC, 2023.
- Xavier Denis, Jacques-Henri Jourdan, Claude Marché. Creusot: a Foundry for the Deductive Verification of Rust Programs. ICFEM 2022.
- JCF and Andrei Paskevich. Abstraction and Genericity in Why3. ISoLA 2020.
- JCF, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. FMSD, 2016.
- Francois Bobot, JCF, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. VSTTE 2013.