

An Introduction to Deductive Program Verification

Jean-Christophe Filliâtre
CNRS

Sixth Summer School on Formal Techniques
May 2016

<http://why3.lri.fr/ssft-16/>



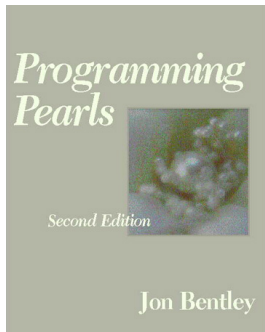
why?

- wrong interpretation of specifications
- coding in a hurry
- incompatible changes
- software = complex artifact
- etc.

a famous example: *binary search*

first publication in 1946

first publication **without bug** in 1962



Jon Bentley. Programming Pearls.
1986.

Writing correct programs

the challenge of binary search

and yet...

in 2006, a bug was found in Java standard library's *binary search*

Joshua Bloch, Google Research Blog

"Nearly All Binary Searches and Mergesorts are Broken"

it had been there for 9 years

```
...  
int mid = (low + high) / 2;  
int midVal = a[mid];  
...
```

may exceed the capacity of type `int`
then provokes an access out of array bounds

a possible fix

```
int mid = low + (high - low) / 2;
```

better programming languages

- better **syntax**
(e.g. avoid considering `DO 17 I = 1. 10` as an assignment)
- more **typing**
(e.g. avoid confusion between meters and yards)
- more **warnings** from the compiler
(e.g. do not forget some cases)
- etc.

systematic and rigorous **test** is another, complementary answer

but test is

- costly
- sometimes difficult to perform
- and **incomplete** (except in some rare cases)

formal methods propose a **mathematical approach** to software correctness

there are several aspects

- **what** we compute
- **how** we compute it
- **why** it is correct to compute it this way

the code is only one aspect (“how”) and nothing else

“what” and “why” are not part of the code

there are informal requirements, comments, web pages, drawings, research articles, etc.

- how: 2 lines of C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",  
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- **how**: 2 lines of C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",  
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- **what**: 15,000 decimals of π
- **why**: lot of maths, including

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2 2^{i+1}}{(2i+1)!}$$

formal methods propose a rigorous approach to programming, where we manipulate

- a **specification** written in some mathematical language
- a **proof** that the program satisfies this specification

what do we intend to prove?

- **safety**: the program does not crash
 - no illegal access to memory
 - no illegal operation, such as division by zero
 - termination
- **functional correctness**
 - the program does what it is supposed to do

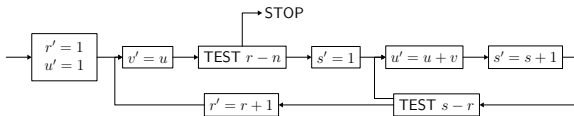
model checking, abstract interpretation, etc.

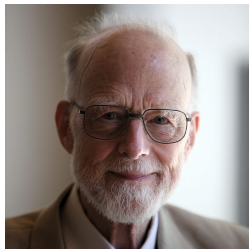
this lecture: **deductive verification**





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





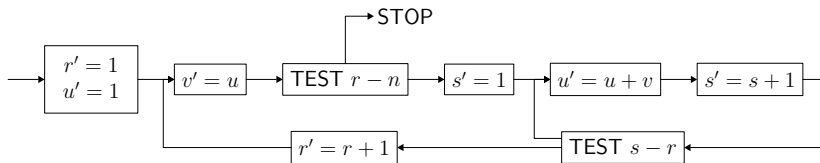
Tony Hoare.

An Axiomatic Basis for Computer Programming.
Commun. ACM, 1969.

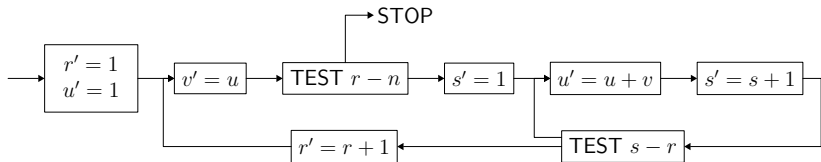
Proof of a program: FIND.
Commun. ACM, 1971.

k		
$\leq v$	v	$\geq v$

checking a large routine (Turing, 1949)

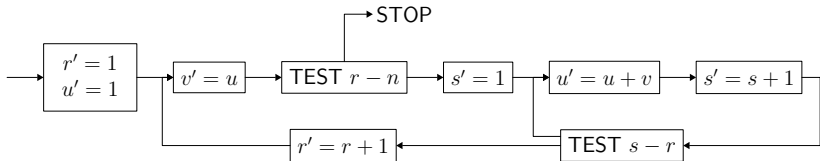


checking a large routine (Turing, 1949)



```
u ← 1
for r = 0 to n - 1 do
  v ← u
  for s = 1 to r do
    u ← u + v
```

checking a large routine (Turing, 1949)



precondition $\{n \geq 0\}$

$u \leftarrow 1$

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

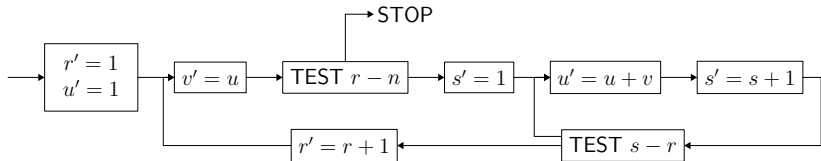
$v \leftarrow u$

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

$u \leftarrow u + v$

postcondition $\{u = \text{fact}(n)\}$

checking a large routine (Turing, 1949)



precondition $\{n \geq 0\}$

$u \leftarrow 1$

for $r = 0$ to $n - 1$ do invariant $\{u = \text{fact}(r)\}$

$v \leftarrow u$

for $s = 1$ to r do invariant $\{u = s \times \text{fact}(r)\}$

$u \leftarrow u + v$

postcondition $\{u = \text{fact}(n)\}$

```

function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn:  $\forall n:\text{int}. n \geq 1 \rightarrow \text{fact}(n) = n * \text{fact}(n-1)$ 

```

```

goal vc:  $\forall n:\text{int}. n \geq 0 \rightarrow$ 
  (0 > n - 1  $\rightarrow$  1 = fact(n))  $\wedge$ 
  (0  $\leq$  n - 1  $\rightarrow$ 
    1 = fact(0)  $\wedge$ 
    ( $\forall u:\text{int}.$ 
      ( $\forall r:\text{int}. 0 \leq r \wedge r \leq n - 1 \rightarrow u = \text{fact}(r) \rightarrow$ 
        (1 > r  $\rightarrow u = \text{fact}(r + 1)$ )  $\wedge$ 
        (1  $\leq$  r  $\rightarrow$ 
          u = 1 * fact(r)  $\wedge$ 
          ( $\forall u1:\text{int}.$ 
            ( $\forall s:\text{int}. 1 \leq s \wedge s \leq r \rightarrow u1 = s * \text{fact}(r) \rightarrow$ 
              ( $\forall u2:\text{int}.$ 
                u2 = u1 + u  $\rightarrow u2 = (s + 1) * \text{fact}(r)$ ))  $\wedge$ 
                (u1 = (r + 1) * fact(r)  $\rightarrow u1 = \text{fact}(r + 1)$ ))))  $\wedge$ 
              (u = fact((n - 1) + 1)  $\rightarrow u = \text{fact}(n)$ )))

```

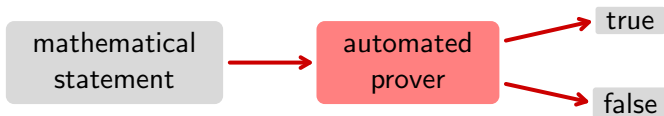
```
function fact(int) : int  
axiom fact0: fact(0) = 1
```

```
goal vc:  $\forall n:\text{int}. n \geq 0 \rightarrow$   
       $(0 > n - 1 \rightarrow 1 = \text{fact}(n)) \wedge$ 
```

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



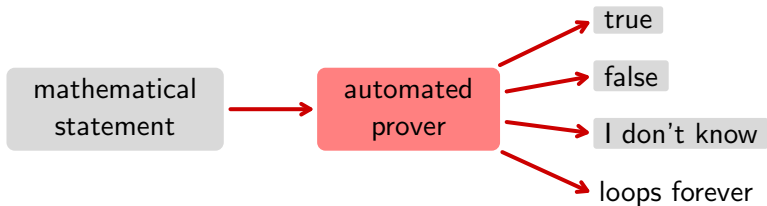
it is not possible to implement such a
program

(Turing/Church, 1936, from Gödel)

full employment theorem for
mathematicians

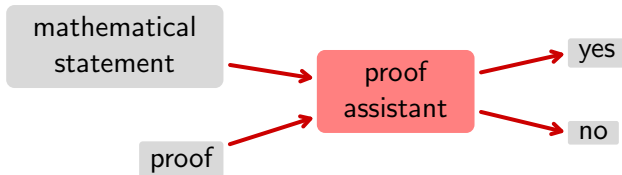


Kurt Gödel



examples: Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

if we only intend to **check** a proof, this is decidable



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

a tool for this lecture

there are many deductive verification tools
(see the lecture web page)

in this lecture, we use **Why3**

but the concepts are broader
(similar to programming languages / learning programming)

Why3 is joint work with
François Bobot, Martin Clochard, Léon Gondelman, Claude
Marché, Guillaume Melquiond, Andrei Paskevich, Mário Pereira

demo

logic of Why3 = **polymorphic first-order logic**, with

- (mutually) recursive algebraic data types
- (mutually) recursive function/predicate symbols
- (mutually) (co)inductive predicates
- let-in, match-with, if-then-else

formal definition in

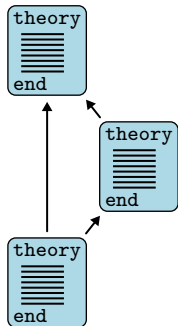
One Logic To Use Them All (**CADE 2013**)

- types
 - abstract: `type t`
 - alias: `type t = list int`
 - algebraic: `type list α = Nil | Cons α (list α)`
- function / predicate
 - uninterpreted: `function f int : int`
 - defined: `predicate non_empty (l: list α) = l \neq Nil`
- inductive predicate
 - `inductive trans t t = ...`
- axiom / lemma / goal
 - `goal G: $\forall x: \text{int}. x \geq 0 \rightarrow x*x \geq 0$`

logic declarations organized in theories

a theory T_1 can be

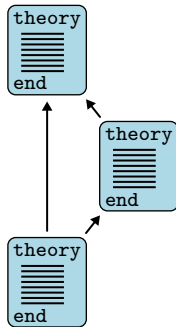
- used (**use**) in a theory T_2
- cloned (**clone**) in another theory T_2



logic declarations organized in theories

a theory T_1 can be

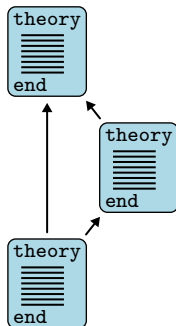
- used (**use**) in a theory T_2
 - symbols of T_1 are **shared**
 - axioms of T_1 remain axioms
 - lemmas of T_1 become axioms
 - goals of T_1 are ignored
- cloned (**clone**) in another theory T_2



logic declarations organized in theories

a theory T_1 can be

- used (**use**) in a theory T_2
- cloned (**clone**) in another theory T_2
 - declarations of T_1 are **copied** or **substituted**
 - axioms of T_1 remain axioms or become lemmas/goals
 - lemmas of T_1 become axioms
 - goals of T_1 are ignored



there are **many** theorem provers

- SMT solvers: Alt-Ergo, Z3, CVC3, Yices, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- proof assistants: Coq, PVS, Isabelle, etc.
- dedicated provers, e.g. Gappa

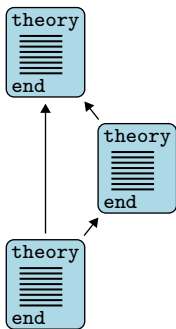
we want to use **all of them** if possible

a technology to talk to provers

central concept: **task**

- a context (a list of declarations)
- a goal (a formula)

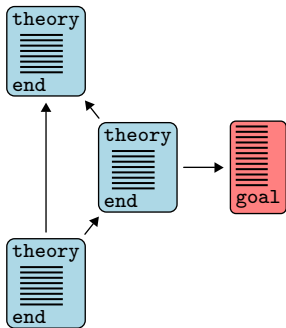




Alt-Ergo

Z3

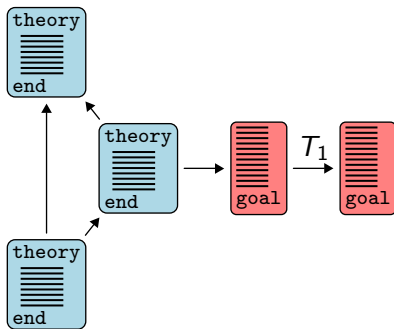
Vampire



Alt-Ergo

Z3

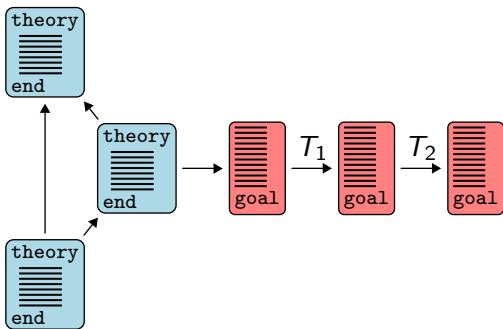
Vampire



Alt-Ergo

Z3

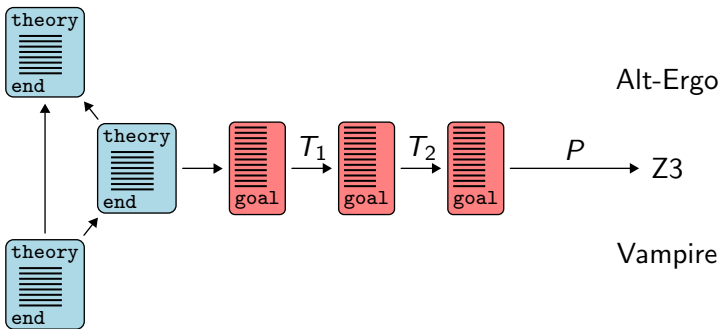
Vampire



Alt-Ergo

Z3

Vampire



- eliminate algebraic data types and match-with
- eliminate inductive predicates
- eliminate if-then-else, let-in
- encode polymorphism, encode types
- etc.

efficient: results of transformations are memoized

a task journey is driven by a file

- transformations to apply
- prover's input format
 - syntax
 - predefined symbols / axioms
- prover's diagnostic messages

more details:

Expressing Polymorphic Types in a Many-Sorted Language (FroCos 2011)

Why3: Shepherd your herd of provers (Boogie 2011)

example: Z3 driver (excerpt)

```
printer "smtv2"
valid "^unsat"
invalid "^sat"

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic"
transformation "simplify_formula"
transformation "discriminate"
transformation "encoding_smt"

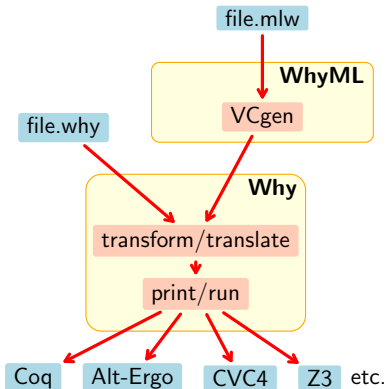
prelude "(set-logic AUFNIRA)"

theory BuiltIn
  syntax type int "Int"
  syntax type real "Real"
  syntax predicate (=) "(= %1 %2)"
  meta "encoding : kept" type int
end
```

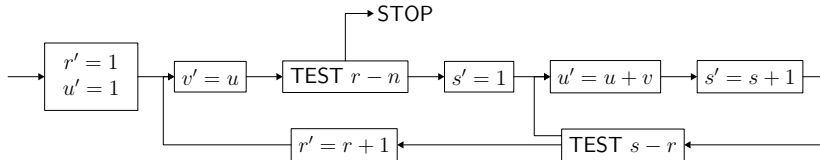
program verification

a programming language, WhyML

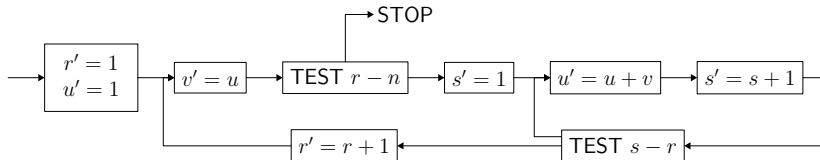
- ML-like syntax
- polymorphism
- pattern-matching
- exceptions
- mutable data structures



A. M. Turing. **Checking a Large Routine.** 1949.



A. M. Turing. *Checking a Large Routine*. 1949.



```

u ← 1
for r = 0 to n - 1 do
  v ← u
  for s = 1 to r do
    u ← u + v
  
```

demo (access code)

$$VC(\text{let } f \text{ x requires } \{ P \} \text{ ensures } \{ Q \} = e) = \\ \forall x. P \Rightarrow WP(e, Q)$$

where $WP(e, Q)$ is the **weakest precondition** for program e to satisfy postcondition Q

$$WP(t, Q) =$$
$$Q[\text{result} \leftarrow t]$$
$$WP(x := t, Q) =$$
$$Q[x \leftarrow t]$$
$$WP(e1; e2, Q) =$$
$$WP(e1, WP(e2, Q))$$
$$WP(\text{if } b \text{ then } e1 \text{ else } e2, Q) =$$
$$\text{if } b \text{ then } WP(e1, Q) \text{ else } WP(e2, Q)$$
$$WP(\text{while } b \text{ do invariant } \{ I \} \text{ e done}, Q) =$$
$$I \wedge \forall x_1, \dots, x_n. I \Rightarrow \text{if } b \text{ then } WP(e, I) \text{ else } Q$$

- instead of substituting, introduce new variables

$$\begin{array}{ll} x := !x + 1; & \forall x_1. x_1 = x_0 + 1 \Rightarrow \\ x := !x * !y; & \forall x_2. x_2 = x_1 \times y \Rightarrow \\ \dots & \dots \end{array}$$

- many other constructs: function application, pattern-matching, for loop, etc.
- exceptional postconditions

computing WPs this way can lead to exponential explosion
e.g.

```
if b1 then ... else ...;  
if b2 then ... else ...;  
if b3 then ... else ...;  
...
```

there are better ways to compute WPs, that are linear in practice
(Flanagan and Saxe, POPL 2001)

termination

Why3 requires all functions in the logic to be terminating

this is *one way* to ensure consistency

e.g. it rules out

```
function f (x: int) : int = 1 + f(x)
```

that would introduce an inconsistency

what about programs?

you can prove either

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

another historical example

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f(f(n + 11)) & \text{sinon.} \end{cases}$$

demo (access code)

another historical example

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f(f(n + 11)) & \text{sinon.} \end{cases}$$

demo (access code)

```
e ← 1
while e > 0 do
  if n > 100 then
    n ← n - 10
    e ← e - 1
  else
    n ← n + 11
    e ← e + 1
return n
```

demo (access code)

termination of a loop / recursive function is ensured by a variant

variant $\{t_1, \dots, t_n\}$

- lexicographic order
- the order relation for t_i
 - is $y \prec x \stackrel{\text{def}}{=} y < x \wedge 0 \leq x$ if t_i has type `int`
 - is immediate sub-term if t_i has some algebraic type
 - is user-given otherwise (e.g. `variant {t with r}`)

as shown with function 91, proving termination may require to establish functional properties as well

another example:

- Floyd's cycle detection (tortoise and hare algorithm)

partial correctness is a rather weak property,
since non-termination can turn your whole proof into something
meaningless (we'll see an example later)

non-termination is an **effect**

Why3 tracks it and warns you about it
(unless an explicit **diverges** is given)

- Euclidean division (`ex1_eucl_div.mlw`)
- factorial (`ex2_fact.mlw`)
- Egyptian multiplication (`ex3_multiplication.mlw`)

arrays

only one kind of mutable data structure:

records with mutable fields

for instance, references are defined this way

```
type ref  $\alpha$  = { mutable contents :  $\alpha$  }
```

and ref, !, and := are regular functions

the library introduces arrays as follows:

```
type array  $\alpha$  model {  
    length: int;  
    mutable elts: map int  $\alpha$   
}
```

where

- map is the logical type of purely applicative maps
- keyword **model** means type array α is an abstract data type in programs

we cannot define operations over type array α
(it is abstract) but we can **declare** them

examples:

```
val ([]) (a: array  $\alpha$ ) (i: int) :  $\alpha$ 
  requires {  $0 \leq i < \text{length } a$  }
  ensures { result = Map.get a.elts i }

val ([] $\leftarrow$ ) (a: array  $\alpha$ ) (i: int) (v:  $\alpha$ ) : unit
  requires {  $0 \leq i < \text{length } a$  }
  writes { a.elts }
  ensures { a.elts = Map.set (old a.elts) i v }
```

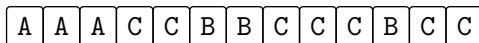
and other operations such as create, append, sub, copy, etc.

when we write `a[i]` in the logic

- it is mere syntax for `Map.get a.elts i`
- we do not prove that `i` is within array bounds
(`a.elts` is a map over all integers)

demo: Boyer-Moore's majority

given a multiset of N votes



determine the majority, if any

due to Boyer & Moore (1980)

linear time

uses only three variables

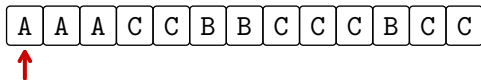
MJRTY—A Fast Majority Vote Algorithm¹

Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

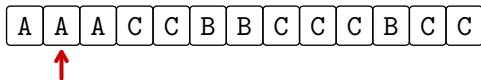
Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.



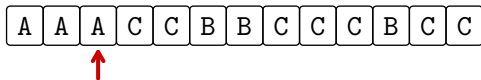
cand = A

k = 1



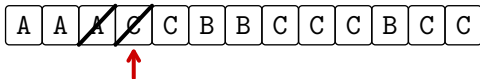
cand = A

k = 2

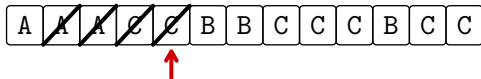


cand = A

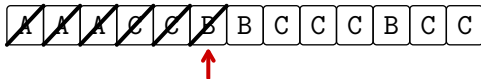
k = 3



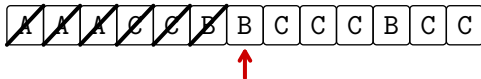
cand = A
k = 2



cand = A
k = 1



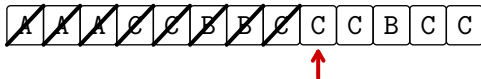
cand = A
k = 0



cand = B
k = 1

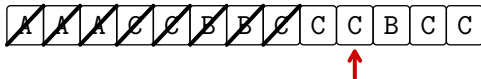


cand = B
k = 0



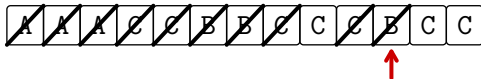
cand = C

k = 1

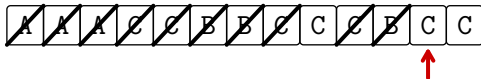


cand = C

k = 2



cand = C
k = 1



cand = C
k = 2



cand = C
k = 3



cand = C

k = 3

then we check if C indeed has majority, with a second pass
(in that case, it has: $7 > 13/2$)

```

SUBROUTINE MJRTY(A, N, BOOLE, CAND)
  INTEGER N
  INTEGER A
  LOGICAL BOOLE
  INTEGER CAND
  INTEGER I
  INTEGER K
  DIMENSION A(N)
  K = 0
C   THE FOLLOWING DO IMPLEMENTS THE PAIRING PHASE. CAND IS
C   THE CURRENTLY LEADING CANDIDATE AND K IS THE NUMBER OF
C   UNPAIRED VOTES FOR CAND.
  DO 100 I = 1, N
    IF ((K .EQ. 0)) GOTO 50
    IF ((CAND .EQ. A(I))) GOTO 75
    K = (K - 1)
    GOTO 100
50   CAND = A(I)
    K = 1
    GOTO 100
75   K = (K + 1)
100  CONTINUE
    IF ((K .EQ. 0)) GOTO 300
    BOOLE = .TRUE.
    IF ((K .GT. (N / 2))) RETURN
C   WE NOW ENTER THE COUNTING PHASE. BOOLE IS SET TO TRUE
C   IN ANTICIPATION OF FINDING CAND IN THE MAJORITY. K IS
C   USED AS THE RUNNING TALLY FOR CAND. WE EXIT AS SOON
C   AS K EXCEEDS N/2.
    K = 0
    DO 200 I = 1, N
      IF ((CAND .NE. A(I))) GOTO 200
      K = (K + 1)
      IF ((K .GT. (N / 2))) RETURN
200  CONTINUE
300  BOOLE = .FALSE.
      RETURN
      END

```

```

let mjrty (a: array candidate) =
  let n = length a in
  let cand = ref a[0] in let k = ref 0 in
  for i = 0 to n-1 do
    if !k = 0 then begin cand := a[i]; k := 1 end
    else if !cand = a[i] then incr k else decr k
  done;
  if !k = 0 then raise Not_found;
  try
    if 2 * !k > n then raise Found; k := 0;
    for i = 0 to n-1 do
      if a[i] = !cand then begin
        incr k; if 2 * !k > n then raise Found
      end
    done;
    raise Not_found
  with Found →
    !cand
end

```

- precondition

```
let mjrty (a: array candidate)  
  requires {  $1 \leq \text{length } a$  }
```

- postcondition in case of success

```
  ensures  
    {  $2 * \text{numeq } a \text{ result } 0 \ (\text{length } a) > \text{length } a$  }
```

- postcondition in case of failure

```
  raises { Not_found  $\rightarrow$   
     $\forall c: \text{candidate.}$   
     $2 * \text{numeq } a \ c \ 0 \ (\text{length } a) \leq \text{length } a$  }
```

first loop

```

for i = 0 to n-1 do
  invariant {  $0 \leq !k \leq \text{numeq } a \text{ !cand } 0 \ i$  }
  invariant {  $2 * (\text{numeq } a \text{ !cand } 0 \ i - !k) \leq i - !k$  }
  invariant {  $\forall c: \text{candidate. } c \neq !cand \rightarrow$   

                $2 * \text{numeq } a \ c \ 0 \ i \leq i - !k$  }
  ...

```

second loop

```

for i = 0 to n-1 do
  invariant {  $!k = \text{numeq } a \text{ !cand } 0 \ i$  }
  invariant {  $2 * !k \leq n$  }
  ...

```


verification conditions express

- safety
 - access within array bounds
 - termination
- user annotations
 - loop invariants are initialized and preserved
 - postconditions are established

fully automated proof

WhyML code can be translated to OCaml code

```
why3 extract -D ocaml64 -D mjrty -T mjrty.Mjrty -o .
```

two drivers used here

- a library driver for 64-bit OCaml
(maps type `int` to `Zarith`, type `array` to OCaml's arrays, etc.)
- a custom driver for this example, namely

```
module mjrty.Mjrty
  syntax type candidate "char"
end
```

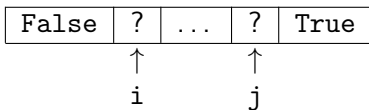
then we can link extracted code with hand-written code

```
ocamlopt ... zarith.cmxa why3extract.cmx  
          mjrty__Mjrty.ml test_mjrty.ml
```

exercise (lab): two-way sort

sort an array of Boolean, using the following algorithm

```
let two_way_sort (a: array bool) =  
  let i = ref 0 in  
  let j = ref (length a - 1) in  
  while !i < !j do  
    if not a[!i] then  
      incr i  
    else if a[!j] then  
      decr j  
    else begin  
      let tmp = a[!i] in  
      a[!i] ← a[!j];  
      a[!j] ← tmp;  
      incr i;  
      decr j  
    end  
  done
```



exercise: ex4_two_way.mlw

exercise (lab): Dutch national flag

an array contains elements of the following enumerated type

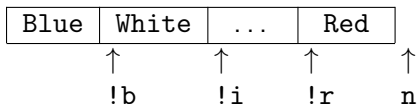
```
type color = Blue | White | Red
```

sort it, in such a way we have the following final situation:

... Blue White Red ...
--------------	---------------	-------------

exercise (lab): Dutch national flag

```
let dutch_flag (a:array color) (n:int) =  
  let b = ref 0 in  
  let i = ref 0 in  
  let r = ref n in  
  while !i < !r do  
    match a[!i] with  
    | Blue →  
      swap a !b !i;  
      incr b;  
      incr i  
    | White →  
      incr i  
    | Red →  
      decr r;  
      swap a !r !i  
  end  
done
```

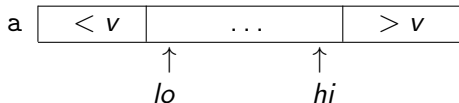


exercise: ex5_flag.mlw

I do not think it means what you think it means



```
lo ← 0
hi ← len(a) - 1
while lo ≤ hi do
    m ← lo + (hi - lo) / 2
    if a[m] < v
        lo ← m + 1
    else if a[m] > v
        hi ← m - 1
    else
        return m
return -1
```




```
let binary_search (a : array int) (v : int) : int
  requires {  $\forall i\ j. 0 \leq i \leq j < \text{length } a \rightarrow a[i] \leq a[j]$  }
  ensures {  $0 \leq \text{result} < \text{length } a \wedge a[\text{result}] = v$ 
    ||  $\text{result} = -1 \wedge \forall i. 0 \leq i < \text{length } a \rightarrow a[i] \neq v$  }
```

if we write instead

```
let binary_search (a: array int) (v: int) : int
  requires {  $\forall i\ j. 0 \leq i \leq j < \text{length } a \rightarrow a[i] \leq a[j]$  }
  ensures {  $(0 \leq \text{result} < \text{length } a \rightarrow a[\text{result}] = v)$ 
    &&  $(\text{result} = -1 \rightarrow \forall i. 0 \leq i < \text{length } a \rightarrow a[i] \neq v)$  }
```

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

and if we write instead

```
let binary_search (a: array int) (v: int) : int
  requires {  $\forall i\ j. 0 \leq i \leq j < \text{length } a \rightarrow a[i] \leq a[j]$  }
  ensures  {  $0 \leq \text{result} < \text{length } a \rightarrow a[\text{result}] = v$ 
    &&  $\text{result} = -1 \rightarrow \forall i. 0 \leq i < \text{length } a \rightarrow a[i] \neq v$  }
```

(note the missing parentheses)

the program can now return 42 and yet be proved correct!

before you do any proof, get the specification right

even more, have the reviewer agree with you on the spec

otherwise, the whole proof is a waste of time

there are many ways of writing the same specification

some are good for humans, others are good for theorem provers

instead of specifying sortedness like this

requires $\{ \forall i\ j. 0 \leq i \leq j < \text{length } a \rightarrow a[i] \leq a[j] \}$

we could do it like this

requires $\{ \forall i. 0 \leq i < \text{length } a - 1 \rightarrow a[i] \leq a[i + 1] \}$

though they are equivalent, the latter now requires **induction** to discharge the VCs of binary search
(and ATPs typically do not perform induction)

it is easy to get binary search wrong when it comes to termination
(e.g. writing $lo \leftarrow m$ instead of $lo \leftarrow m + 1$)

if you are not proving termination, you can still prove the program correct but this is **partial correctness**

ghost code

data and code that is added to the program
for the purpose of specification and/or proof only

we search the smallest Fibonacci number equal to or greater than n

```
 $a, b \leftarrow 0, 1$   
while  $a < n$  do  
     $a, b \leftarrow b, a + b$   
return  $a$ 
```

we could have a loop invariant as follows

```
let a = ref 0 in
let b = ref 1 in
while !a < n do
  invariant {  $\exists i. i \geq 0 \ \&\& \ !a = \text{fib } i \ \&\& \ !b = \text{fib } (i+1)$  }
  ...
```

but existential quantifiers make VCs that are difficult to discharge

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

```
let a = ref 0 in
let b = ref 1 in
let ghost i = ref 0 in (* ghost data *)
while !a < n do
  invariant { !i ≥ 0 && !a = fib !i && !b = fib (!i+1) }
  ...
  i := !i + 1 (* ghost statement *)
done
```

instead of having the ATP guessing the right value, we provide it

- 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
(and is removed during OCaml extraction)

a function

```
let f (x: t) : unit
  requires { P }
  ensures  { Q }
= ...
```

that is pure (i.e. does not modify global data) and terminating
can be turned into a lemma automatically

```
lemma f:  $\forall x: t. P \rightarrow Q$ 
```

the declaration `let lemma` tells Why3 to do that

```
let rec lemma fib_pos (n: int) : unit
  requires { n ≥ 1 }
  variant { n }
  ensures { fib n ≥ 1 }
  =
  if n > 2 then begin fib_pos (n - 2); fib_pos (n - 1) end
```

- we have performed a **proof by induction**
(thanks to the variant and the WP calculus)
- we can make **explicit calls** to a lemma function

```
fib_pos 42;    (* this is ghost code *)  
...
```

it saves ATPs the burden of instantiating the lemma

ghost code can be used to **model** the contents of a data structure

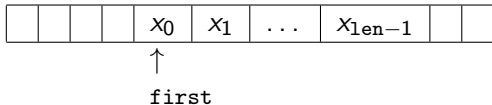
say we want to implement a queue with bounded capacity

```
type queue  $\alpha$   
val create: int  $\rightarrow$  queue  $\alpha$   
val push:  $\alpha \rightarrow$  queue  $\alpha \rightarrow$  unit  
val pop: queue  $\alpha \rightarrow \alpha$ 
```

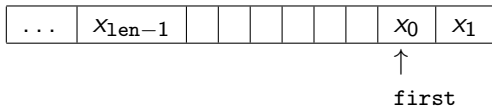
it can be implemented with an array

```
type buffer  $\alpha$  = {
  mutable first: int;
  mutable len  : int;
  data : array  $\alpha$ ;
}
```

len elements are stored, starting at index first



they may wrap around the array bounds



to give a specification to queue operations, we would like to **model** the queue contents, say, as a sequence of elements

one way to do it is to use **ghost code**

we add two **ghost fields** to model the queue contents

```
type queue  $\alpha$  = {  
  ...  
  ghost          capacity: int;  
  ghost mutable sequence: Seq.seq  $\alpha$ ;  
}
```

then we use them in specifications

```
val create (n: int) (dummy:  $\alpha$ ) : queue  $\alpha$ 
  requires { n > 0 }
  ensures { result.capacity = n }
  ensures { result.sequence = Seq.empty }

val push (q: queue  $\alpha$ ) (x:  $\alpha$ ) : unit
  requires { Seq.length q.sequence < q.capacity }
  writes { q.sequence }
  ensures { q.sequence = Seq.snoc (old q.sequence) x }

val pop (q: queue  $\alpha$ ) :  $\alpha$ 
  requires { Seq.length q.sequence > 0 }
  writes { q.sequence }
  ensures { result = (old q.sequence)[0] }
  ensures { q.sequence = (old q.sequence)[1 ..] }
```

we are already able to prove some **client code** using the queue

```
let harness () =  
  let q = create 10 0 in  
  push q 1;  
  push q 2;  
  push q 3;  
  let x = pop q in assert { x = 1 };  
  let x = pop q in assert { x = 2 };  
  let x = pop q in assert { x = 3 };  
  ()
```

we link the regular fields and the ghost fields with a **type invariant**

```

type buffer  $\alpha$  =
  ...
invariant {
    self.capacity = Array.length self.data  $\wedge$ 
     $0 \leq \text{self.first} < \text{self.capacity} \wedge$ 
     $0 \leq \text{self.len} \leq \text{self.capacity} \wedge$ 
    self.len = Seq.length self.sequence  $\wedge$ 
     $\forall i: \text{int}. 0 \leq i < \text{self.len} \rightarrow$ 
      (self.first + i < self.capacity  $\rightarrow$ 
        Seq.get self.sequence i = self.data[self.first + i])  $\wedge$ 
      ( $0 \leq \text{self.first} + i - \text{self.capacity} \rightarrow$ 
        Seq.get self.sequence i = self.data[self.first + i
          - self.capacity])
  }

```


such a type invariant holds at **function boundaries**

thus

- it is **assumed** at function entry
- it must be **ensured**
 - when a function is called
 - at function exit, for values returned or modified

ghost code is added to set ghost fields accordingly

example:

```
let push (b: buffer  $\alpha$ ) (x:  $\alpha$ ) : unit
=
  ghost b.sequence  $\leftarrow$  Seq.snoc b.sequence x;
  let i = b.first + b.len in
  let n = Array.length b.data in
  b.data[if i  $\geq$  n then i - n else i]  $\leftarrow$  x;
  b.len  $\leftarrow$  b.len + 1
```

implement other operations

- clear
- head
- pop

on ring buffers and prove them correct

exercise: `ex6_buffer.mlw`

purely applicative programming

a **key idea** of Hoare logic:

*any types and symbols from the logic
can be used in programs*

note: we already used type `int` this way

we can do so with algebraic data types

in the library, we find

```
type bool = True | False           (in bool.Bool)
type option  $\alpha$  = None | Some  $\alpha$  (in option.Option)
type list  $\alpha$  = Nil | Cons  $\alpha$  (list  $\alpha$ ) (in list.List)
```

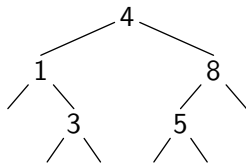
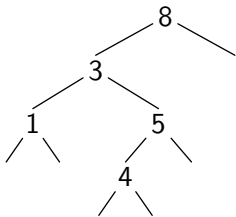
let us consider binary trees

```
type elt
```

```
type tree =  
  | Empty  
  | Node tree elt tree
```

and the following problem

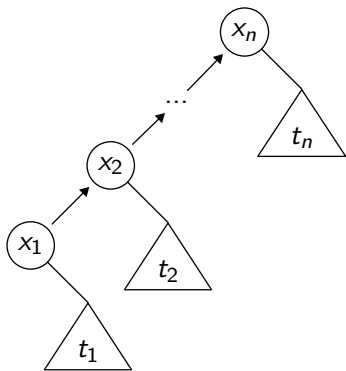
given two binary trees,
do they contain the same elements when traversed in order?



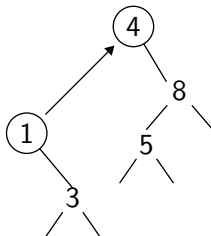
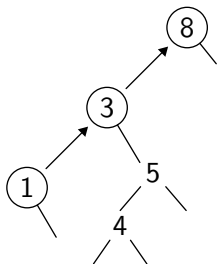
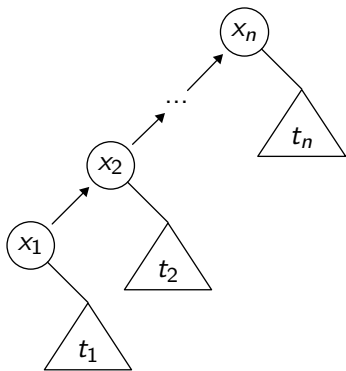

```
function elements (t: tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end
```

```
let same_fringe (t1 t2: tree) : bool
  ensures { result=True  $\leftrightarrow$  elements t1 = elements t2 }
  =
  ...
```

one solution: look at the left branch as
a list, from bottom up



one solution: look at the left branch as
a list, from bottom up



demo (access code)

exercise (lab): inorder traversal

```
type elt
type tree = Null | Node tree elt tree
```

inorder traversal of t , storing its elements in array a

```
let rec fill (t: tree) (a: array elt) (start: int) : int =
  match t with
  | Null →
      start
  | Node l x r →
      let res = fill l a start in
      if res ≠ length a then begin
        a[res] ← x;
        fill r a (res + 1)
      end else
        res
  end
```

exercise: ex7_fill.mlw

machine arithmetic

let us model signed 32-bit arithmetic

two possibilities:

- ensure absence of arithmetic overflow
- model machine arithmetic faithfully (i.e. with overflows)

a **constraint**:

we do not want to lose the arithmetic capabilities of SMT solvers

we introduce a new type for 32-bit integers

```
type int32
```

its integer value is given by

```
function toint int32 : int
```

main idea: within annotations, we only use type `int`
(thus a program variable `x : int32` always appears as `toint x` in annotations)

we define the range of 32-bit integers

```
function min_int: int = - 0x8000_0000 (* -2^31 *)
function max_int: int =  0x7FFF_FFFF (* 2^31-1 *)
```

when we use them...

```
axiom int32_domain:
   $\forall x: \text{int32}. \text{min\_int} \leq \text{to\_int } x \leq \text{max\_int}$ 
```

... and when we build them

```
val ofint (x: int) : int32
  requires { min_int ≤ x ≤ max_int }
  ensures { toint result = x }
```


then each program expression such as

$$x + y$$

is translated into

```
ofint (toint x + toint y)
```

this ensures the absence of arithmetic overflow
(but we get a large number of additional verification conditions)

let us show the absence of arithmetic overflow in binary search

demo (access code)

we found a bug

the computation

```
let m = (!l + !u) / 2 in
```

may provoke an arithmetic overflow
(for instance with a 2-billion elements array)

a possible fix is

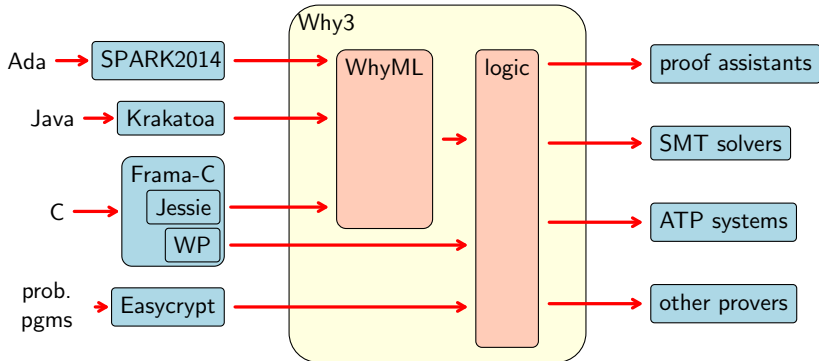
```
let m = !l + (!u - !l) / 2 in
```

conclusion

three different ways of using Why3

- as a logical language
(a convenient front-end to many theorem provers)
- as a programming language to prove algorithms
(currently 120 examples in our gallery)
- as an intermediate language
(for the verification of C, Java, Ada, etc.)

some systems using Why3



- how aliases are controlled
- how verification conditions are computed
- how formulas are sent to provers
- how pointers/heap are modeled
- how floating-point arithmetic is modeled
- etc.

see <http://why3.lri.fr> for more details

there are many other ways to perform program verification, e.g.

- abstract interpretation
- model checking

including other ways to perform deductive verification

- dedicated program logic e.g. separation logic
- symbolic evaluation

see <http://why3.lri.fr/ssft-16/>

uses a simpler version of the Why3 GUI running in your browser

only one prover available (Alt-Ergo 1.00)