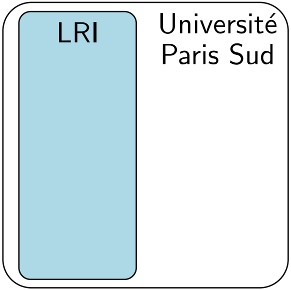
Deductive Program Verification with Why3

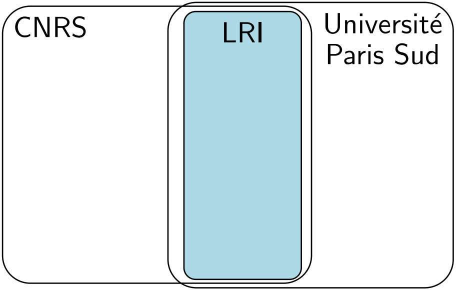
Jean-Christophe Filliâtre
CNRS

EJCP
June 25, 2015

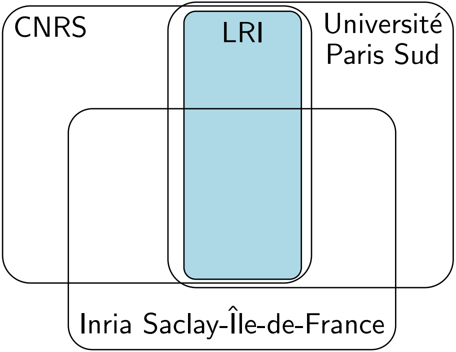
http://why3.lri.fr/ejcp-2015/

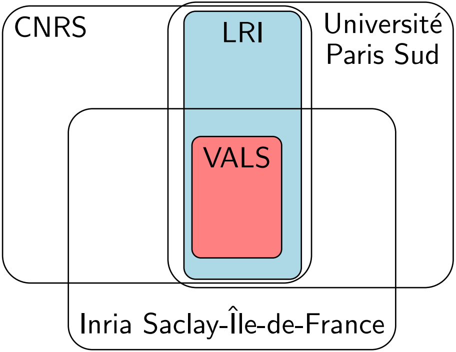
Université
Paris Sud


LRI
Université
Paris Sud


CNRS
LRI
Université
Paris Sud

team VALS — http://vals.lri.fr/



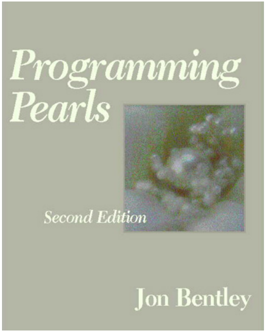

CNRS
LRI
Université
Paris Sud
VALS
Inria Saclay-Île-de-France

Software is hard. – Don Knuth

why?

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

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

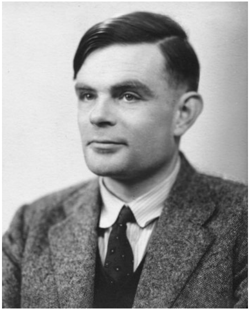
what do we intend to prove?

- 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

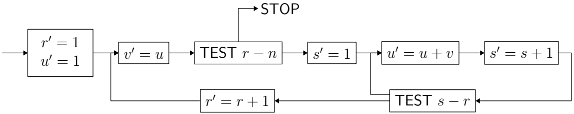
model checking, abstract interpretation, etc.

this lecture introduces deductive verification

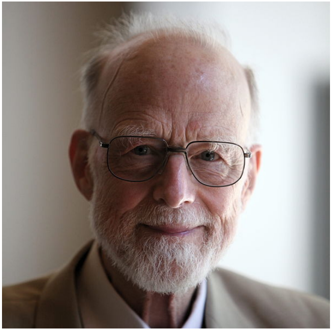
this is not new



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



this is not new

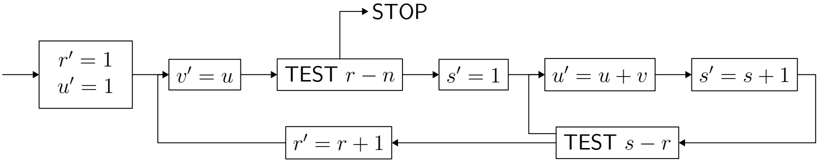


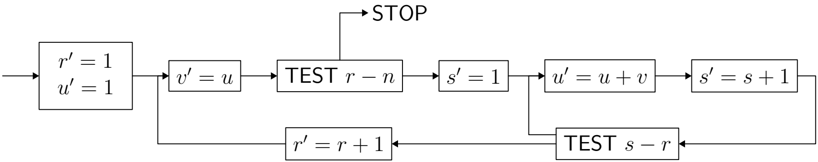
Tony Hoare.
Proof of a program: FIND.
Commun. ACM, 1971.

k

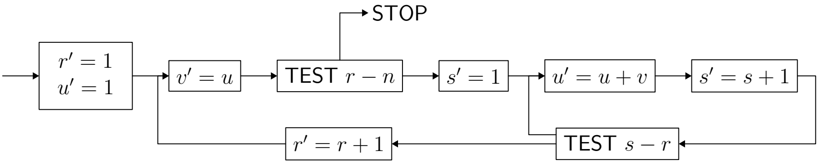
$\leq v$	v	$\geq v$

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
```



precondition $\{n \geq 0\}$

$u \leftarrow 1$

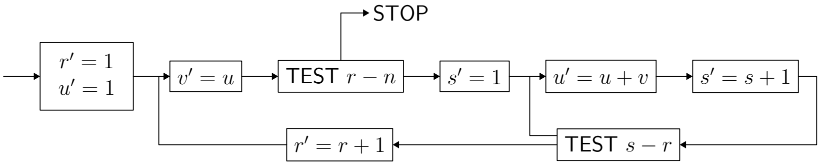
for $r = 0$ **to** $n - 1$ **do**

    $v \leftarrow u$

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

        $u \leftarrow u + v$

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



precondition $\{n \geq 0\}$

$u \leftarrow 1$

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

    $v \leftarrow u$

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

        $u \leftarrow u + v$

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

```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                  u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
            (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
      (u = fact((n - 1) + 1) → u = fact(n))))
```

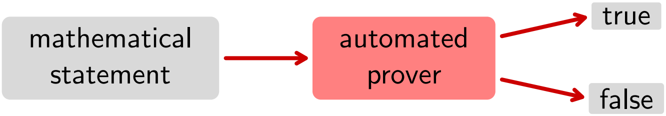
```
function fact(int) : int
axiom fact0: fact(0) = 1
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(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

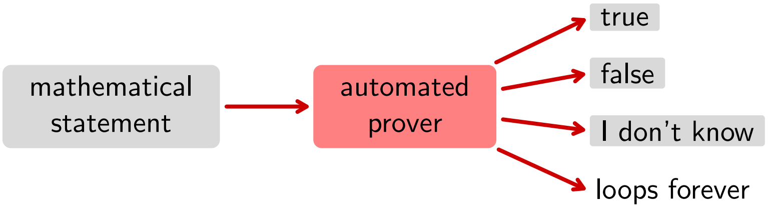
so we turn to tools that mechanize mathematical reasoning


mathematical
statement
automated
prover
true
false

it is not possible to implement such a program
(Turing/Church, 1936, from Gödel)

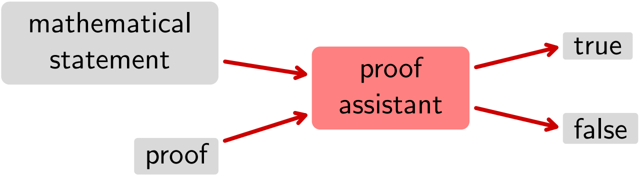
full employment theorem for mathematicians

Kurt Gödel



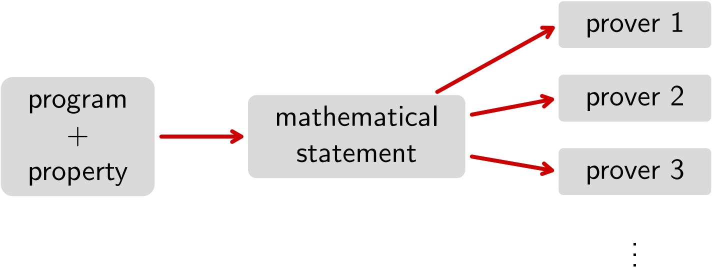
examples: Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

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



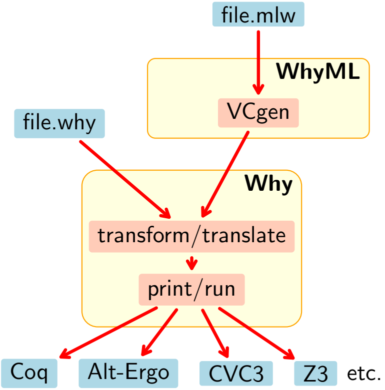
examples: Coq, Isabelle, PVS, HOL Light, etc.

main idea: use as many theorem provers as possible
(both automated and interactive)



- a programming language, WhyML
  - polymorphism
  - pattern-matching
  - exceptions
  - mutable data structures, with controlled aliasing
- a polymorphic logic
  - algebraic data types
  - recursive definitions
  - (co)inductive predicates

http://why3.lri.fr/

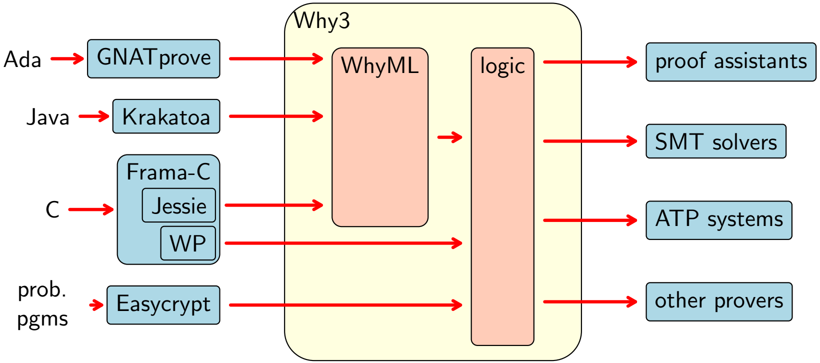


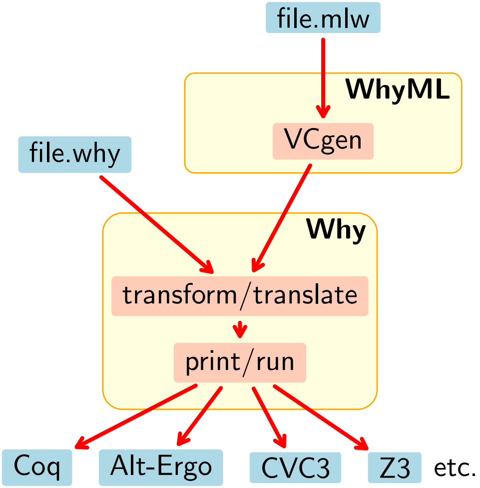
three different ways of using Why3

- as a logical language
  (a convenient front-end to many theorem provers)

- as a programming language to prove algorithms
  (many examples in our gallery)

- as an intermediate language,
  to verify programs written in C, Java, Ada, etc.


Why3
Ada
GNATprove
Java
Krakatoa
C
Frama-C
Jessie
WP
prob.
pgms
Easycrypt
WhyML
logic
proof assistants
SMT solvers
ATP systems
other provers


file.mlw
WhyML
VCgen
file.why
Why
transform/translate
print/run
Coq
Alt-Ergo
CVC3
Z3
etc.

Part I

one logic to use them all

demo 1: the logic of Why3

logic of Why3 = polymorphic logic, with

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

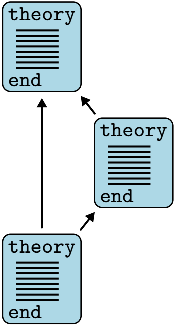
formal definition in
One Logic To Use Them All (CADE 2013)

- 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: int. x` $\geq$ `0` $\rightarrow$ `x*x` $\geq$ `0`

logic declarations organized in theories

a theory T_1 can be

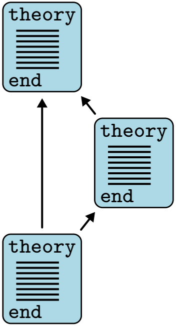
- used (`use`) in a theory T_2
- cloned (`clone`) in another theory T_2



logic declarations organized in theories

a theory T_1 can be

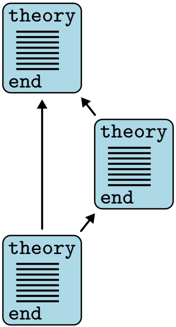
- used (`use`) in a theory T_2
  - symbols of T_1 are shared
  - axioms of T_1 remain axioms
  - lemmas of T_1 become axioms
  - goals of T_1 are ignored
- cloned (`clone`) in another theory T_2



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

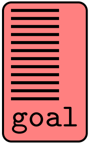
- SMT solvers: Alt-Ergo, Z3, CVC3, Yices, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- proof assistants: Coq, PVS, Isabelle, etc.
- dedicated provers, e.g. Gappa

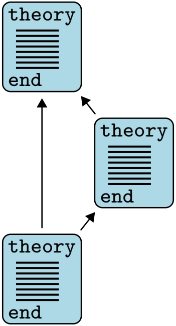
we want to use all of them if possible

a technology to talk to provers

central concept: task

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

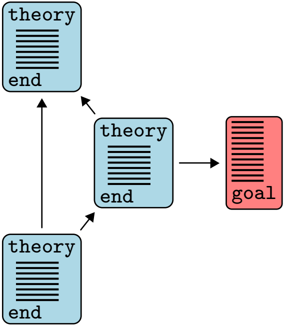




Alt-Ergo

Z3

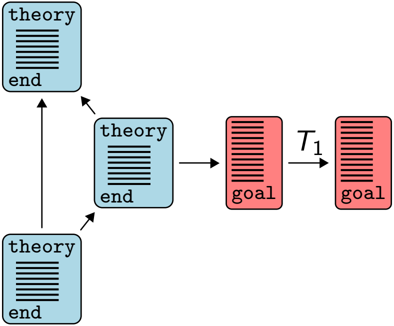
Vampire



Alt-Ergo

Z3

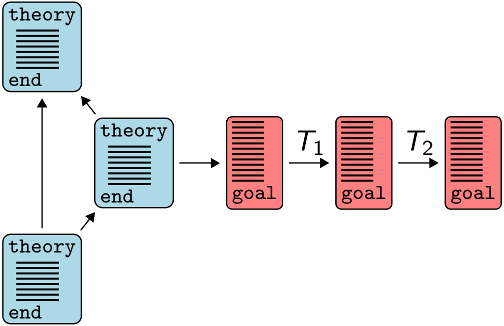
Vampire



Alt-Ergo

Z3

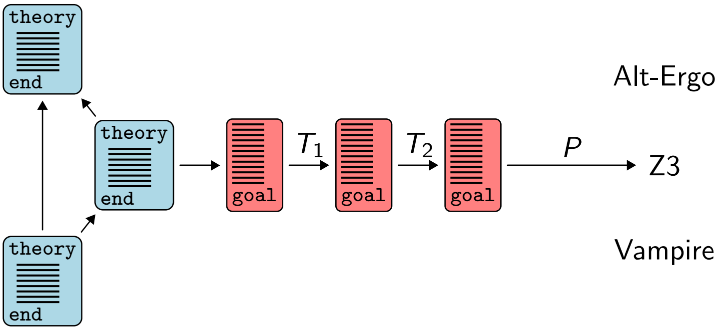
Vampire



Alt-Ergo

Z3

Vampire


theory
end
theory
end
theory
end
goal
T_1
goal
T_2
goal
P
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)

```
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
```

Why3 has an OCaml API

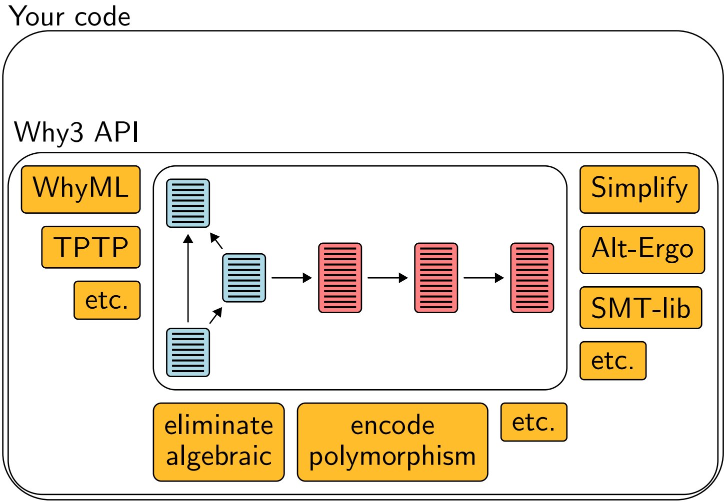
- to build terms, declarations, theories, tasks
- to call provers

defensive API

- well-typed terms
- well-formed declarations, theories, and tasks

Why3 can be extended via three kinds of plug-ins

- parsers (new input formats)
- transformations (to be used in drivers)
- printers (to add support for new provers)


Your code
Why3 API
WhyML
TPTP
etc.
Simplify
Alt-Ergo
SMT-lib
etc.
eliminate algebraic
encode polymorphism
etc.

- numerous theorem provers are supported
  - SMT, TPTP, proof assistants, etc.
- user-extensible system
  - input languages
  - transformations
  - output syntax
- proofs
  - are preserved
  - can be replayed

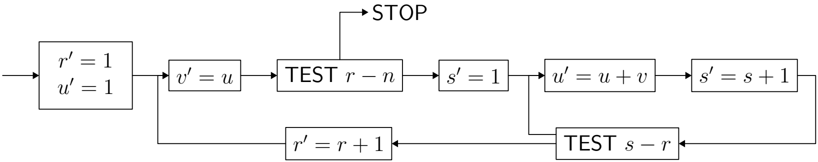
more details:

Preserving User Proofs Across Specification Changes (VSTTE 2013)

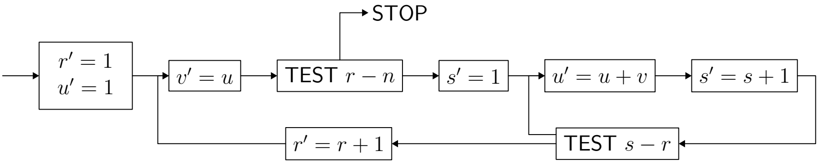
Part II

program verification

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



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



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

demo (access code)

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

demo (access code)

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

demo (access code)

$e \leftarrow 1$
while $e > 0$ do
  if $n > 100$ then
    $n \leftarrow n - 10$
    $e \leftarrow e - 1$
  else
    $n \leftarrow n + 11$
    $e \leftarrow e + 1$
return n

demo (access code)

- pre/postcondition

```
let foo x y z
  requires { P } ensures { Q }
  = ...
```

- loop invariant

```
while ... do invariant { I } ... done

for i = ... do invariant { I(i) } ... done
```

termination of a loop (resp. a recursive function) is ensured by a variant

$$\texttt{variant}\ \{t\}\ \texttt{with}\ R$$

- R is a well-founded order relation
- t decreases for R at each step
  (resp. each recursive call)

by default, t is of type `int` and R is the relation

$$y \prec x \stackrel{\text{def}}{=} y < x \land 0 \leq x$$

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) now, it's up to you

suggested exercises

- Euclidean division (`exo_eucl_div.mlw`)
- Factorial (`exo_fact.mlw`)
- Fast exponentiation (`exo_power.mlw`)

Part III

arrays

only one kind of mutable data structure:

records with mutable fields

for instance, references are defined this way

```
type ref α = { mutable contents : α }
```

and `ref`, `!`, and `:=` are regular functions the library introduces arrays as follows:

```
type array α model {
        length: int;
  mutable elts: map int α
}
```

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 α) (i: int) : α
  requires { 0 ≤ i < length a }
  ensures  { result = Map.get a.elts i }

val ([]←) (a: array α) (i: int) (v: α) : unit
  requires { 0 ≤ i < 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)

given a multiset of N votes

A	A	A	C	C	B	B	C	C	C	B	C	C

determine the majority, if any

an elegant solution

due to Boyer & Moore (1980)

linear time

uses only three variables

MJRTY—A Fast Majority Vote Algorithm[1]

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.

A	A	A	C	C	B	B	C	C	C	B	C	C

```
cand = A
k    = 1
```

A	A	A	C	C	B	B	C	C	C	B	C	C

```
cand = A
k    = 2
```

A	A	A	C	C	B	B	C	C	C	B	C	C

```
cand = A
k    = 3
```

A	A	~~A~~	~~C~~	C	B	B	C	C	C	B	C	C

```
cand = A
k    = 2
```

A	~~A~~	~~A~~	~~C~~	~~C~~	B	B	C	C	C	B	C	C

```
cand = A
k    = 1
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	B	C	C	C	B	C	C

```
cand = A
k    = 0
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	B	C	C	C	B	C	C

```
cand = B
k    = 1
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	~~B~~	~~C~~	C	C	B	C	C

```
cand = B
k    = 0
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	~~B~~	~~C~~	C	C	B	C	C

```
cand = C
k    = 1
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	~~B~~	~~C~~	C	C	B	C	C

```
cand = C
k    = 2
```

principle

~~A~~ ~~A~~ ~~A~~ ~~C~~ ~~C~~ ~~B~~ ~~B~~ ~~C~~ C ~~C~~ ~~B~~ C C

```
cand = C
k    = 1
```

~~A~~ ~~A~~ ~~A~~ ~~C~~ ~~C~~ ~~B~~ ~~B~~ ~~C~~ C ~~C~~ ~~B~~ C C

```
cand = C
k    = 2
```

~~A~~ ~~A~~ ~~A~~ ~~C~~ ~~C~~ ~~B~~ ~~B~~ ~~C~~ C ~~C~~ ~~B~~ C C

```
cand = C
k    = 3
```

~~A~~	~~A~~	~~A~~	~~C~~	~~C~~	~~B~~	~~B~~	~~C~~	C	~~C~~	~~B~~	C	C

```
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
```

demo (access code)

- precondition

```
let mjrty (a: array candidate)
  requires { 1 ≤ length a }
```

- postcondition in case of success

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

- postcondition in case of failure

```
  raises { Not_found →
             ∀ c: candidate.
             2 * numeq a c 0 (length a) ≤ length a }
```

first loop

```
for i = 0 to n-1 do
  invariant { 0 ≤ !k ≤ numeq a !cand 0 i }
  invariant { 2 * (numeq a !cand 0 i - !k) ≤ i - !k }
  invariant { ∀ c: candidate. c ≠ !cand →
                2 * numeq a c 0 i ≤ i - !k  }
  ...
```

second loop

```
for i = 0 to n-1 do
  invariant { !k = numeq a !cand 0 i }
  invariant { 2 * !k ≤ 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.cmxa
                 mjrty__Mjrty.ml test_mjrty.ml
```

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
```

False	?	...	?	True
	↑ i		↑ j	

exercise: `exo_two_way.mlw`

an array contains elements of the following enumerated type

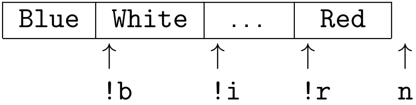
```
type color = Blue | White | Red
```

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

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

exercise: 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: exo_flag.mlw

Part IV

specifying / implementing a data structure

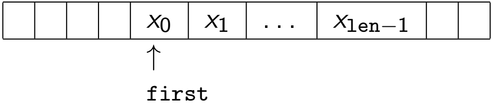
say we want to implement a queue with bounded capacity

```
type queue α
val create: int → queue α
val push: α → queue α → unit
val pop: queue α → α
```

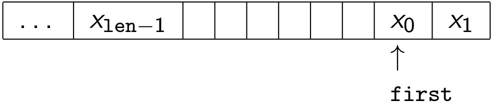
it can be implemented with an array

```
type buffer α = {
  mutable first: int;
  mutable len  : int;
          data : array α;
}
```

`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

may be inserted for the purpose of specification and/or proof

rules are:

- 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

in particular, ghost code can be removed without observable modification (and is removed during OCaml extraction) we add two ghost fields to model the queue contents

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

then we use them in specifications

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

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

val pop (q: queue α) : α
  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 α =
  ...
invariant {
  self.capacity = Array.length self.data ∧
  0 ≤ self.first <  self.capacity ∧
  0 ≤ self.len   ≤ self.capacity ∧
  self.len = Seq.length self.sequence ∧
  ∀ i: int. 0 ≤ i < self.len →
    (self.first + i < self.capacity →
      Seq.get self.sequence i = self.data[self.first + i]) ∧
    (0 ≤ self.first + i - self.capacity →
      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 α) (x: α) : unit
  =
  ghost b.sequence ← Seq.snoc b.sequence x;
  let i = b.first + b.len in
  let n = Array.length b.data in
  b.data[if i ≥ n then i - n else i] ← x;
  b.len ← b.len + 1
```

implement other operations

- `length`
- `clear`
- `head`

on ring buffers and prove them correct

Part V

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

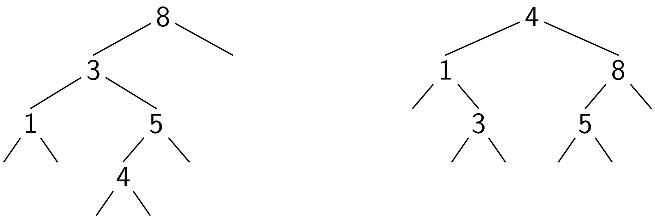
```
type bool = True | False                      (in bool.Bool)
type option α = None | Some α                 (in option.Option)
type list α = Nil | Cons α (list α)           (in list.List)
```

let us consider binary trees

```
type elt

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

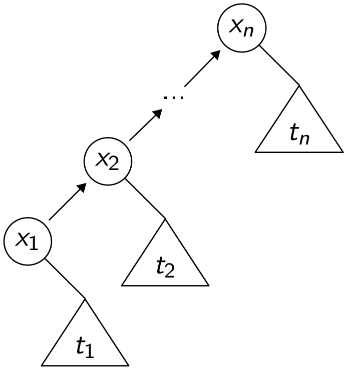
and the following problem

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

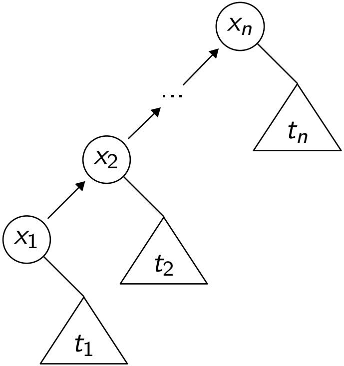


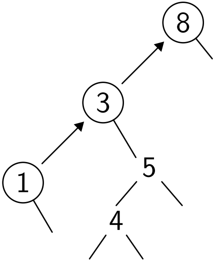
```
function elements (t: tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end

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

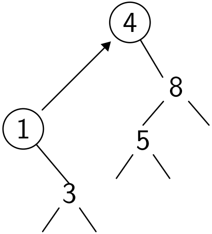
one solution: look at the left branch as a list, from bottom up



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







demo (access code)

```
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: exo_fill.mlw

Part VI

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 loose 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:
  ∀ x: int32. min_int ≤ toint x ≤ 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

$$\texttt{ofint}\ (\texttt{toint}\ x)\ (\texttt{toint}\ y)$$

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

let us consider searching for a value in a sorted array using binary search

let us show the absence of arithmetic overflow

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.)

- 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