An Introduction to Deductive Program Verification

Jean-Christophe Filliâtre
CNRS

Summer School on Verification Technology, Systems & Applications
August 2018

http://why3.lri.fr/vtsa-18/
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


*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
... 
```java
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;
```
what can we do?

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
what is a program?

there are several aspects

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

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

```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 \(\pi\)

• **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
several approaches

model checking, abstract interpretation, etc.

this lecture: deductive verification
Tony Hoare.


\[
\begin{array}{c|c|c}
  k & \leq v & \geq v \\
\end{array}
\]
checking a large routine (Turing, 1949)

\begin{align*}
  r' &= 1 \\
  u' &= 1 \\
  v' &= u \\
  \text{TEST } r - n \\
  s' &= 1 \\
  u' &= u + v \\
  s' &= s + 1 \\
  r' &= r + 1 \\
  \text{TEST } s - r
\end{align*}
checking a large routine (Turing, 1949)

\[ r' = 1 \]
\[ u' = 1 \]
\[ v' = u \]
\[ \text{TEST } r - n \]
\[ s' = 1 \]
\[ u' = u + v \]
\[ s' = s + 1 \]
\[ r' = r + 1 \]
\[ \text{TEST } s - r \]

\[ u ← 1 \]
\[ \text{for } r = 0 \text{ to } n - 1 \text{ do} \]
\[ \quad v ← u \]
\[ \quad \text{for } s = 1 \text{ to } r \text{ do} \]
\[ \quad \quad u ← u + v \]
precondition \{ n \geq 0 \}

\textbf{for} r = 0 \textbf{to} n - 1 \textbf{do}

\hspace{1em} \textbf{v} \leftarrow u

\hspace{2em} \textbf{for} s = 1 \textbf{to} r \textbf{do}

\hspace{3em} u \leftarrow u + v

\textbf{postcondition} \{ u = \text{fact}(n) \}
precondition \( \{ n \geq 0 \} \)

\[
\begin{align*}
  u & \leftarrow 1 \\
  \text{for } r = 0 \text{ to } n - 1 \text{ do } & \quad \text{invariant } \{ u = \text{fact}(r) \} \\
  & \quad v \leftarrow u \\
  & \quad \text{for } s = 1 \text{ to } r \text{ do } \quad \text{invariant } \{ u = s \times \text{fact}(r) \} \\
  & \quad \quad u \leftarrow u + v \\
  \text{postcondition } & \quad \{ u = \text{fact}(n) \}
\end{align*}
\]
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: forall n:int. n >= 1 -> fact(n) = n * fact(n-1)

goal vc: forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = fact(n)) /
  (0 <= n - 1 ->
    1 = fact(0) /
    (forall u:int /\ 
     (forall r:int. 0 <= r /\ r <= n - 1 -> u = fact(r) ->
      (1 > r -> u = fact(r + 1)) /
      (1 <= r ->
       u = 1 * fact(r) /
       (forall u1:int /\ 
        (forall s:int. 1 <= s /\ s <= r -> u1 = s * fact(r) ->
         (forall 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: forall n:int. n >= 0 ->
    (0 > n - 1 -> 1 = fact(n)) /
...

verification condition
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
automated theorem proving

- 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
automated theorem proving

examples: Z3, CVC4, Alt-Ergo, Vampire, E, 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)
joint work with

Françoise Bobot

Claude Marché

Guillaume Melquiond

Andrei Paskevich
Why3 — the big picture

WhyML

Why3 logic

Alt-Ergo  Z3  CVC4

...  ...  ...
Why3 — the big picture

WhyML

Why3 logic

OCaml  C

Alt-Ergo  Z3  CVC4

...
Why3 — the big picture

WhyML

Java → WhyML
C → WhyML
Ada → WhyML
Python → WhyML
OCaml → WhyML
C → WhyML

Why3 logic

Alt-Ergo → Why3 logic
Z3 → Why3 logic
CVC4 → Why3 logic
Why3 — the big picture

- WhyML
  - Why3 logic
    - Alt-Ergo
    - Z3
    - CVC4
  - OCaml
  - C
- Java
- C
- Ada
- Python
- your language
- your VCs
Why3 — the big picture

WhyML → Why3 logic

your VCs → your language

Java → WhyML

C → WhyML

Ada

Python → WhyML

OCaml → Why3 logic

C

Alt-Ergo → Why3 logic

Z3 → Why3 logic

CVC4 → Why3 logic

...
goal
rich enough to make your life easier,
simple enough to be sent to ATPs
goal
rich enough to make your life easier,
simple enough to be sent to ATPs

our solution
a total, polymorphic first-order logic with

- algebraic types & pattern matching
- recursive definitions
- (co)inductive predicates
- mapping type $\alpha \rightarrow \beta$, $\lambda$-notation, application

[FroCos 2011, CADE 2013, VSTTE 2014]
a first demo
logic declarations

- **types**
  - abstract: `type t`
  - alias: `type t = list int`
  - algebraic: `type list 'a = Nil | Cons 'a (list 'a)`

- **function / predicate**
  - uninterpreted: `function f int : int`
  - defined: `predicate non_empty (l: list 'a) = l <> Nil`

- **inductive predicate**
  - `inductive trans t t = ...`

- **axiom / lemma / goal**
  - `goal G: forall x: int. x >= 0 -> x*x >= 0`
logic declarations can be organized in **modules**

module $M_1$ can be used in module $M_2$ in two ways

- imported (**use**)

- copied, possibly with a substitution (**clone**)

*modules*
logic declarations can be organized in modules

module $M_1$ can be used in module $M_2$ in two ways

- imported (use)
  - symbols of $M_1$ are shared
  - axioms of $M_1$ remain axioms
  - lemmas of $M_1$ become axioms
  - goals of $M_1$ are ignored

- copied, possibly with a substitution (clone)
logic declarations can be organized in modules

module $M_1$ can be used in module $M_2$ in two ways

- imported (use)

- copied, possibly with a substitution (clone)
  - declarations of $M_1$ are copied or substituted
  - axioms of $M_1$ remain axioms or become lemmas/goals
  - lemmas of $M_1$ become axioms
  - goals of $M_1$ are ignored
there are many theorem provers

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

we want to use as many of them as possible

Why3 currently supports 25+ ITPs and ATPs
a technology to talk to provers

central concept: task
  • a context (a list of declarations)
  • a goal (a formula)
to transmit a task to a prover, we apply a sequence of logical transformations, until it fits into the logic of the prover

then we print in the syntax of the prover

good $T_1$ $T_2$ $T_3$ $P$ some prover
• eliminate algebraic data types and match-with
• eliminate inductive predicates
• eliminate if-then-else, let-in
• encode polymorphism, encode types
• etc.
each prover is controlled by a “driver” 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

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)"
end
...
Why3 can be extended with

- user OCaml plugins declaring
  - new logical transformations
  - new pretty-printers
  - new input formats
- user driver files
program verification
WhyML, a language for program verification

- ML-like syntax
- polymorphism
- algebraic data types, pattern matching
- exceptions, break, continue, return
- mutable data types, with static control of aliases
- ghost code and ghost data

- contracts, loop and type invariants

\[
\begin{align*}
    r' &= 1 \\
    u' &= 1 \\
    v' &= u \\
    r &= r + 1 \\
    u &= u + v \\
    s &= s + 1 \\
\end{align*}
\]

\[
\begin{align*}
    u' &= u + v \\
    s' &= s + 1 \\
    r' &= r + 1 \\
    \text{for } s &= 1 \text{ to } r \text{ do } \\
    u &= u + v \\
\end{align*}
\]

\[
\begin{align*}
    u' &= u \\
    s' &= 1 \\
    \text{for } r &= 0 \text{ to } n - 1 \text{ do } \\
\end{align*}
\]

demo (access code)
computing verification conditions

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

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

\[ WP(t, Q) = \]
\[ Q[result \leftarrow t] \]

\[ WP(x := t, Q) = \]
\[ Q[x \leftarrow t] \]

\[ WP(e_1; e_2, Q) = \]
\[ WP(e_1, WP(e_2, Q)) \]

\[ WP(\text{if } b \text{ then } e_1 \text{ else } e_2, Q) = \]
\[ \text{if } b \text{ then } WP(e_1, Q) \text{ else } WP(e_2, Q) \]

\[ WP(\text{while } b \text{ do invariant } \{ I \} \text{ e done}, Q) = \]
\[ I \land \forall x_1, \ldots, x_n. I \Rightarrow \text{if } b \text{ then } WP(e, I) \text{ else } Q \]
• instead of substituting, introduce new variables

\[
\begin{align*}
x & := !x + 1; & \forall x_1. x_1 &= x_0 + 1 \Rightarrow \\
x & := !x \times !y; & \forall x_2. x_2 &= x_1 \times y \Rightarrow \\
\end{align*}
\]

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

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

```plaintext
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]

Why3 implements both styles,
and they can even be mixed within the same function
termination
Why3 requires all functions in the logic to be terminating

this is one way to ensure consistency

e.g. it rules out

```plaintext
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
on a per-function basis, either

- add \textit{diverges} to the contract, or
- provide a \textit{variant} for each loop / recursive (sub-)function

a variant is a list of terms

\[
\text{variant} \ \{t_1, \ldots, t_n\}
\]

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

\[ f(n) = \begin{cases} 
  n - 10 & \text{if } n > 100 \\
  f(f(n + 11)) & \text{otherwise}
\end{cases} \]

demo (access code)
another historical example

\[ f(n) = \begin{cases} 
    n - 10 & \text{if } n > 100 \\
    f(f(n + 11)) & \text{otherwise}
\end{cases} \]

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)
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 ensures you either prove termination or put an explicit diverges in the contract
• Euclidean division (ex1_eucl_div.mlw)
• factorial (ex2_fact.mlw)
• Russian multiplication (ex3_multiplication.mlw)
arrays
only one kind of mutable data structure:

records with mutable fields

for instance, references are defined this way

\[
\text{type ref } 'a = \{ \text{mutable contents : } 'a \}
\]

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

```
type array 'a = private {
  mutable ghost elts: int -> 'a;
  length: int
} invariant { 0 <= length }
```

where

- **private** means we cannot build records of that type
- **ghost** means field `elts` can only be used in the logic
we cannot define operations over type array 'a (it is private) but we can declare some

designated examples:

val ([])(a: array 'a) (i: int): 'a
   requires { 0 <= i < length a }
   ensures { result = Map.get a.elts i }

val ([]<-) (a: array 'a) (i: int) (v: 'a): 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 $\text{Map.get} \ a.\text{elts} \ i$
- we do not prove that $i$ is within array bounds ($a.\text{elts}$ is a map over all integers)
given a multiset of $N$ votes

[A A A C C C B B B C C C C C B C C C]

determine the majority, if any
an elegant solution

due to Boyer & Moore (1980)

linear time

uses only three variables
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: \( 7 > \frac{13}{2} \))
SUBROUTINE MJRTY(A, N, BOOLE, CAND)
  INTEGER N
  INTEGER A
  LOGICAL BOOLE
  INTEGER CAN
  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) : 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;
  if 2 * !k > n then return !cand;
  k := 0;
  for i = 0 to n - 1 do
    if a[i] = !cand then begin
      incr k;
      if 2 * !k > n then return !cand
    end
  done;
  raise Not_found
• **precondition**

```ocaml
let mjrty (a: array candidate) : candidate
  requires { 1 <= length a }
```

• **postcondition in case of success**

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

• **postcondition in case of failure**

```ocaml
raises { Not_found ->
  forall c: candidate.
  2 * num a c 0 (length a) <= length a }
```
first loop

for i = 0 to n - 1 do
  invariant { 0 <= !k <= num a !cand 0 i }
  invariant { 2 * (num a !cand 0 i - !k) <= i - !k }
  invariant { forall c: candidate. c <> !cand ->
    2 * num a c 0 i <= i - !k }
...

second loop

for i = 0 to n - 1 do
  invariant { !k = num a !cand 0 i }
  invariant { 2 * !k <= n }
...

loop invariants
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 -L . -D ocaml64 -D mjrty.drv \ 
mjrty.Mjrty -o mjrty.ml
```

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

```ocaml
module mjrty.Mjrty

  syntax type candidate "char"
  syntax val (=) "%1 = %2"

end
```
then we can link extracted code with hand-written code

```
ocamlfind ocamlopt -package zarith -linkpkg \\
mjrty.ml test_mjrty.ml -o test_mjrty
```

and run it

```
./test_mjrty
candidate 'c' has absolute majority
no one has absolute majority
```
sort an array of Boolean, using the following algorithm

```ocaml
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
```

<table>
<thead>
<tr>
<th>False</th>
<th>?</th>
<th>...</th>
<th>?</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exercise: ex4_two_way.mlw
exercise (lab): Dutch national flag

an array contains elements of the following enumerated type

\texttt{type color = Blue \mid White \mid Red}

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

\begin{tabular}{|c|c|c|}
  \hline
  \ldots & Blue & \ldots \\
  \hline
  \ldots & White & \ldots \\
  \hline
  \ldots & Red & \ldots \\
  \hline
\end{tabular}
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

<table>
<thead>
<tr>
<th>Blue</th>
<th>White</th>
<th>...</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td></td>
<td>↓b</td>
<td>↓i</td>
<td>↓r</td>
</tr>
</tbody>
</table>

exercise: ex5_flag.mlw
I do not think it means what you think it means
$lo \leftarrow 0$

$hi \leftarrow \text{len}(a) - 1$

\textbf{while} $lo \leq hi$ \textbf{do}

\hspace{1em} $m \leftarrow lo + (hi - lo)/2$

\hspace{1em} \textbf{if} $a[m] < v$

\hspace{2em} $lo \leftarrow m + 1$

\hspace{1em} \textbf{else if} $a[m] > v$

\hspace{2em} $hi \leftarrow m - 1$

\hspace{1em} \textbf{else}

\hspace{2em} \textbf{return} $m$

\textbf{return} -1
let binary_search (a : array int) (v : int) : int
requires {
    forall i j. 0 <= i <= j < length a -> a[i] <= a[j] }
ensures {
    0 <= result < length a \ a[result] = v
\ result = -1 \ forall i. 0 <= i < length a -> a[i] <> v }
if we write instead

```ocaml
let binary_search (a: array int) (v: int) : int
  requires {
    forall i j. 0 <= i <= j < length a -> a[i] <= a[j] }
  ensures {
    (0 <= result < length a -> a[result] = v)
  /
    (result = -1 -> forall i. 0 <= i < length a -> a[i] <> v) }
```

if we write instead

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

the program can now return \(-2\) and yet be proved correct
and if we write instead

```ocaml
let binary_search (a: array int) (v: int) : int
  requires {
    forall i j. 0 <= i <= j < length a -> a[i] <= a[j] }
  ensures {
    0 <= result < length a -> a[result] = v
    /
    result = -1 -> forall i. 0 <= i < length a -> a[i] <> v }
```

(note the missing parentheses)
and if we write instead

```ml
let binary_search (a: array int) (v: int) : int
  requires {
    forall i j. 0 <= i <= j < length a -> a[i] <= a[j] 
  } 
  ensures {
    0 <= result < length a -> a[result] = v
    \(\forall\) result = -1 -> forall i. 0 <= i < length a -> a[i] <> 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

\[
\text{requires } \left\{ \begin{array}{l}
\text{forall } i \ j. \ 0 \leq i \leq j < \text{length } a \rightarrow a[i] \leq a[j] \\
\end{array} \right. 
\]

we could do it like this

\[
\text{requires } \left\{ \begin{array}{l}
\text{forall } i. \ 0 \leq i < \text{length } a - 1 \rightarrow a[i] \leq a[i + 1] \\
\end{array} \right. 
\]

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 $n$

$$
a, b \leftarrow 0, 1 \\
\text{while } a < n \text{ do} \\
\quad a, b \leftarrow b, a + b \\
\text{return } a$$
we could have a loop invariant as follows

```ocaml
let a = ref 0 in
let b = ref 1 in
while !a < n do
  invariant {
    exists i. i >= 0 \ a = fib i \ !b = 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

```ocaml
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 = \text{fib \( !i \)} /\ !b = \text{fib \( (!i+1) \)} \)}
  ...
  i := !i + 1 (* ghost statement *)
done
```

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

- ghost code may read regular data but can’t modify it
- ghost code cannot modify the control flow of regular code
  - must terminate, does not raise exceptions
- regular code does not see ghost data

[The Spirit of Ghost Code, CAV 2014]

consequence: ghost code can be removed without observable modification (and is removed during OCaml extraction)
a function

```ocaml
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

```ocaml
lemma f: forall x: t. P -> 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

we already saw it with arrays
say we want to implement a queue with bounded capacity

```ocaml
type queue 'a
val create: int -> queue 'a
val push: 'a -> queue 'a -> unit
val pop: queue 'a -> 'a
```
it can be implemented with an array

```ocaml
type buffer 'a = {
  mutable first: int;
  mutable len : int;
  data : array 'a;
}
```

len elements are stored, starting at index first

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X0</td>
<td>X1</td>
<td>...</td>
<td>X_{len-1}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>first</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

they may wrap around the array bounds

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>X_{len-1}</td>
<td></td>
<td></td>
<td></td>
<td>X0</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>first</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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 'a = {
  ...
  ghost capacity: int;
  ghost mutable sequence: Seq.seq 'a;
}

then we use them in specifications

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

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

val pop (q: queue 'a) : 'a
  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

```ocaml
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 regular fields and ghost fields with a type invariant

type buffer 'a =
  ...

invariant {  
capacity = Array.length data /
0 <= first <  capacity /
0 <= len  <= capacity /
len = Seq.length sequence /
forall i. 0 <= i < len ->
  (first + i < capacity ->
   Seq.get sequence i = data[first + i]) /
  (0 <= first + i - capacity ->
   Seq.get sequence i = data[first + i - capacity])
}
such a type invariant holds at **function and annotation 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
  - when referred to within an annotation
ghost code is added to set ghost fields accordingly

example:

```ocaml
let push (b: buffer 'a) (x: 'a) : 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
```

(keyword `ghost` can be omitted; Why3 would infer it is ghost code)
implement other operations

- clear
- head
- pop

on ring buffers and prove them correct

exercise: ex6_buffer.mlw
pointer data structures
mutable recursive data structures are beyond the reach of Why3’s static control of aliases

example

define llist 'a = ref (node 'a)
with node 'a = Nil | Cons 'a (llist 'a)

This field has non-pure type, it cannot be used in a recursive type definition
one solution consists in **modeling the heap**

- within Why3, we have an explicit heap and explicit pointers
- when translating to OCaml, it is mapped to OCaml types

let’s illustrate this with a union-find data structure
type elem
val make : unit -> elem
val union : elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union : elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union : elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union : elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
**union-find**

```ocaml
type elem
val make : unit -> elem
val union : elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
type elem

type uf = {
    mutable dom: set elem;
    mutable rep: elem -> elem;
}

val ghost create () : uf
val make (ghost uf: uf) () : elem
val union (ghost uf: uf) (x y: elem) : unit
val find (ghost uf: uf) (x : elem) : elem
val same (ghost uf: uf) (x y: elem) : bool
type elem

type uf = {
  mutable dom: set elem;
  mutable rep: elem -> elem;
}

invariant { forall x. mem x dom ->
  mem (rep x) dom && rep (rep x) = rep x }

val ghost create () : uf
  ensures { result.dom = empty }
val make \((\text{ghost uf: uf})\) () : elem
  writes \{ uf.dom, uf.rep \}
  ensures \{ \text{not (mem result (old uf.dom))} \}
  ensures \{ uf.dom = \text{add result (old uf.dom)} \}
  ensures \{ uf.rep = (\text{old uf.rep})[\text{result <- result}] \}

val find \((\text{ghost uf: uf})\) (x: elem) : elem
  requires \{ \text{mem x uf.dom} \}
  ensures \{ \text{result = uf.rep x} \}
val union (ghost uf: uf) (x y: elem) : ghost elem
  requires { mem x uf.dom }
  requires { mem y uf.dom }
  writes   { uf.rep }
  ensures  { result = old (uf.rep x) ||
             result = old (uf.rep y) }
  ensures  { forall z. mem z uf.dom ->
             uf.rep z = if old (uf.rep z = uf.rep x ||
                             uf.rep z = uf.rep y)
                             then result
                             else old (uf.rep z) }
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
type elem =
    content ref

and content =
    | Link of elem
    | Root of int
type elem =
    content ref

and content =
    | Link of elem
    | Root of int

x \[0\] y \[0\] z \[0\]
**implementation**

type elem =
  content ref

and content =
  | Link of elem
  | Root of int

\[
\begin{array}{c}
\text{x} & \text{1} \\
\text{y} \\
\text{z} & \text{0}
\end{array}
\]
type elem =
  content ref
and content =
  | Link of elem
  | Root of int
type elem =
    content ref

and content =
    | Link of elem
    | Root of int
type elem =
    content ref

and content =
    | Link of elem
    | Root of int
**implementation**

```ocaml
type elem =
    content ref

and content =
    | Link of elem
    | Root of int
```

![Diagram of a tree structure](image)
type elem =
    content ref
and content =
    | Link of elem
    | Root of int
type elem = 
  content ref

and content = 
  | Link of elem
  | Root of int
type elem =
  content ref
and content =
  | Link of elem
  | Root of int

let’s verify this with Why3
OCaml

```ocaml
type elem =
  content ref
and content =
  | Link of elem
  | Root of int
```

Why3

```why3
type loc

type elem =
  loc

type content =
  | Link loc
  | Root Peano.t

type heap = {
  ghost mutable
  refs: loc -> option content;
}
```
heap operations

val alloc_ref (ghost h: heap) (v: content) : loc
  ...
val get_ref (ghost h: heap) (l: loc) : content
  ...
val set_ref (ghost h: heap) (l: loc) (c: content) : unit
  ...
val (==) (x y: loc) : bool
  ...

predicate allocated (h: heap) (x: loc) =
  h.refs x <> None
type uf = {
  heap: heap;
  mutable dom : set elem;
  mutable rep : elem -> elem;
}
...
invariant { forall x. mem x dom <-> allocated heap x }
type uf =
    ...

invariant { forall x. match heap.refs x with
    | Some (Link y) -> x <> y \ allocated heap y \ rep x = rep y
    | Some (Root _) -> rep x = x
    | None -> true end }

invariant { forall x. mem x dom ->
    match heap.refs (rep x) with
    | Some (Root _) -> true
    | _ -> false end }
it would be very tempting to introduce an inductive notion of path

\[
\text{inductive \ path \ (h: \ heap) \ (x \ y: \ elem) =}
\]

\[
| \ \text{Path0: forall \ x \ y \ k.} \\
| \ h.\text{refs} \ x = \text{Some} \ (\text{Root} \ k) \rightarrow \\
| \ \text{path} \ h \ x \ x \\
\]

\[
| \ \text{Path1: forall \ x \ y \ z.} \\
| \ h.\text{refs} \ x = \text{Some} \ (\text{Link} \ y) \rightarrow \\
| \ \text{path} \ h \ y \ z \rightarrow \ \text{path} \ h \ x \ z
\]

this way, we would have \text{path heap} \ x \ (\text{rep} \ x) \ as \ an \ invariant
and this would ensure the termination of \text{find}
but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along Link
- a maximum distance for the whole union-find structure
but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along Link
- a maximum distance for the whole union-find structure

\[
\text{maxd} = 1
\]
but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

• a distance to each node, increasing along Link
• a maximum distance for the whole union-find structure

\[ \text{maxd} = 2 \]
but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along Link
- a maximum distance for the whole union-find structure

\[
\text{maxd} = 2
\]
• type Peano.t is mapped to OCaml's type int
• our custom mini-heap is mapped to OCaml’s references

```ocaml
module uf.UnionFind

  syntax type loc = "content ref"
  syntax val (==) = "%1 == %2"
  syntax val alloc_ref = "ref %1"
  syntax val get_ref = "!%1"
  syntax val set_ref = "%1 := %2"

end
```
Charguéraud & Pottier did a Coq proof of a similar OCaml code, using CFML

- includes a proof of complexity!
- maps OCaml’s type `int` to Coq’s type `Z` (unsound)
- more than 4k lines
1. modeling the heap can be easy
   - can be local
   - incurs a small TCB

2. avoid recursive/inductive definitions for better automation

   two other examples:
   - heap stored in an array
   - inverting a permutation in-place
heap stored in an array

a = [1, 3, 4, 4, 7, 5, ...]

It would be tempting to introduce trees, but a universal, local invariant $\forall i. a[i] \leq a[2i+1]$, $a[2i+2]$ is all you need.
heap stored in an array

it would be tempting to introduce trees

but a universal, local invariant

\[ \forall i. \ a[i] \leq a[2i + 1], a[2i + 2] \]

is all you need
Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
4 & 3 & 0 & 1 & 5 & 2 \\
\end{array}
\]
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

| 4 | 3 | 0 | 1 | 5 | -1 |
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
4 & 3 & -6 & 1 & 5 & -1 \\
\end{array}
\]
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
-3 & 3 & -6 & 1 & 5 & -1 \\
\end{array}
\]
Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[ \begin{array}{cccccc}
-3 & 3 & -6 & 1 & -1 & -1 \\
\end{array} \]
Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
-3 & 3 & -6 & 1 & -1 & 4 \\
\end{array}
\]
Algorithm I in TAOCP [Sec. 1.3.3, page 176]
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
-3 & 3 & -6 & -1 & 0 & 4 \\
\end{array}
\]
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3  -4  -6  -1  0  4
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
-3 & -4 & -6 & 1 & 0 & 4
\end{array}
\]
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

```
-3  -4  5  1  0  4
```
inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
-3 & 3 & 5 & 1 & 0 & 4 \\
\end{array}
\]
Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
2 & 3 & 5 & 1 & 0 & 4
\end{array}
\]
Algorithm I in TAOCP [Sec. 1.3.3, page 176]

\[
\begin{array}{cccccc}
2 & 3 & 5 & 1 & 0 & 4 \\
\end{array}
\]

again it would tempting to introduce paths, orbits, cycles, etc.

but again a universal, local invariant suffices
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

for convenience, we declare

```
meta coercion function toint
```
32-bit arithmetic

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: 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 }
```
addition without overflow

\[
\text{val} \ (\ + \ ) \ (x: \ \text{int32}) \ (y: \ \text{int32}) : \ \text{int32} \\
\quad \text{requires} \ { \ min\_\text{int} \ <= \ x + y <= \ max\_\text{int} } \\
\quad \text{ensures} \ { \ \text{result} = x + y } 
\]

(here we use coercions on \( x \), \( y \), and \( \text{result} \))

this ensures the \textit{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
we found a bug

the computation

\[
\text{let } m = (\!l + \!u) / 2 \text{ in}
\]

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

a possible fix is

\[
\text{let } m = \!l + (\!u - \!l) / 2 \text{ in}
\]
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 150 examples in our gallery)

- as an intermediate language
  (for the verification of C, Java, Ada, etc.)
things not covered in this lecture

- how aliases are controlled  
  [hal-01256434]
- Why3’s OCaml API  
  [BOOGIE 2011]
- proof by reflection  
  including imperative programs  
  [VSTTE 2016]  
  [IJCAR 2018]
- logical connectives by and so to encode proofs  
  [ARITH 2007]
- floating-point arithmetic  
  [VSTTE 2013]
- checking the consistency of libraries using Coq
- preserving proofs across changes

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
thank you
see http://why3.lri.fr/vtsa-18/

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

only one prover available (Alt-Ergo 2.0.0)