Deductive Program Verification with Why3

Jean-Christophe Filliâtre
CNRS

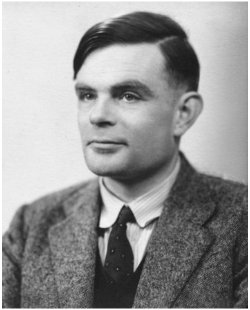
Digicosme Spring School
April 22, 2013

http://why3.lri.fr/digicosme-spring-school-2013/

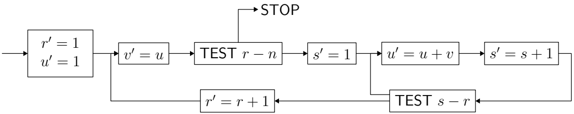
definition

this is not new

program + specification → verification conditions → proof

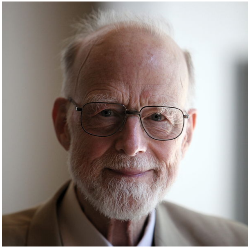


A. M. Turing. Checking a large routine. 1949.



this is not new

program + specification → verification conditions → proof



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

proving

program + specification → verification conditions → proof

a lot of theorem provers

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

- too rich: we can't use automated theorem provers
- too poor: we can't model programming languages and we can't specify programs

typically, a compromise

- first-order logic
- a bunch a theories: arithmetic, arrays, bit vectors, etc.

extracting verification conditions for a realistic programming language is a lot of work

as in a compiler, we rather translate to some intermediate language from which we extract VCs developed since 2001 at ProVal (LRI / INRIA)

rewritten from scratch, started Feb 2010 $\Rightarrow$ Why3

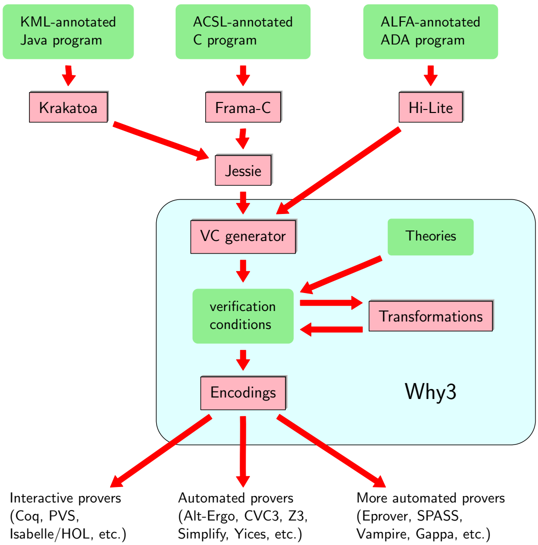
authors: F. Bobot, JCF, C. Marché, G. Melquiond, A. Paskevich

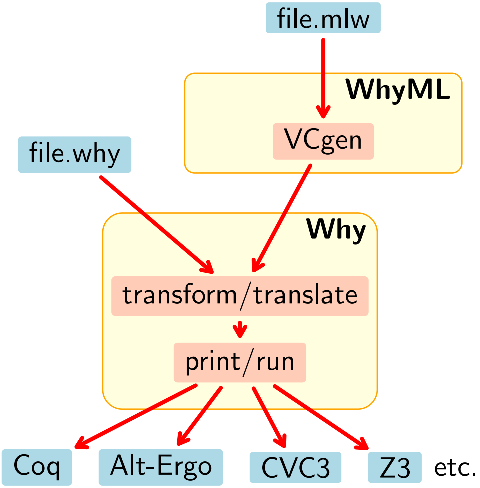
open source software (LGPL)

http://why3.lri.fr/

a similar tool: Boogie (Microsoft Research)

- Java programs: Krakatoa (Marché Paulin Urbain)
- C programs: Caduceus (Filliâtre Marché) formerly, Jessie plug-in of Frama-C (Marché Moy) today
- Ada programs: Hi-Lite (Adacore)
- algorithms
- probabilistic programs (Barthe et al.)
- cryptographic programs (Vieira)


KML-annotated
Java program
ACSL-annotated
C program
ALFA-annotated
ADA program
Krakatoa
Frama-C
Hi-Lite
Jessie
VC generator
Theories
verification
conditions
Transformations
Encodings
Why3
Interactive provers
(Coq, PVS,
Isabelle/HOL, etc.)
Automated provers
(Alt-Ergo, CVC3, Z3,
Simplify, Yices, etc.)
More automated provers
(Eprover, SPASS,
Vampire, Gappa, etc.)


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

Part I

the logic of Why3

logic of Why3 = polymorphic first-order logic, with

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

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

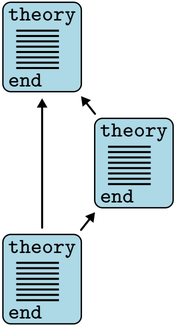
Demo 1: the logic of Why3

- 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

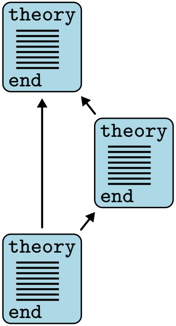
- used (`use`) in a theory T_2
- cloned (`clone`) in another theory T_2



logic declarations organized in theories

a theory T_1 can be

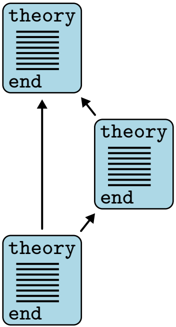
- used (`use`) in a theory T_2
  - symbols of T_1 are shared
  - axioms of T_1 remain axioms
  - lemmas of T_1 become axioms
  - goals of T_1 are ignored
- cloned (`clone`) in another theory T_2



logic declarations organized in theories

a theory T_1 can be

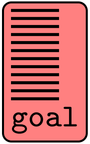
- used (`use`) in a theory T_2
- cloned (`clone`) in another theory T_2
    - declarations of T_1 are copied or substituted
    - axioms of T_1 remain axioms or become lemmas/goals
    - lemmas of T_1 become axioms
    - goals of T_1 are ignored

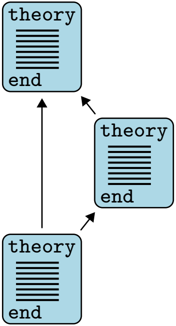


a technology to talk to provers

central concept: task

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

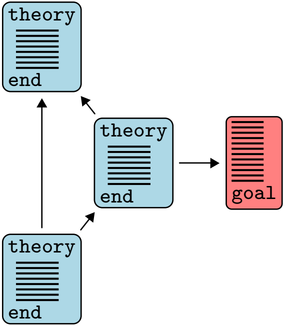




Alt-Ergo

Z3

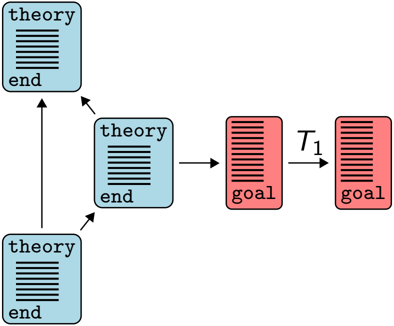
Vampire



Alt-Ergo

Z3

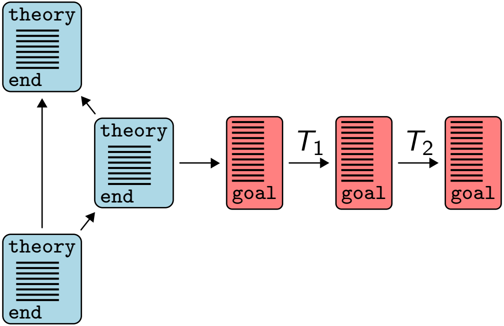
Vampire



Alt-Ergo

Z3

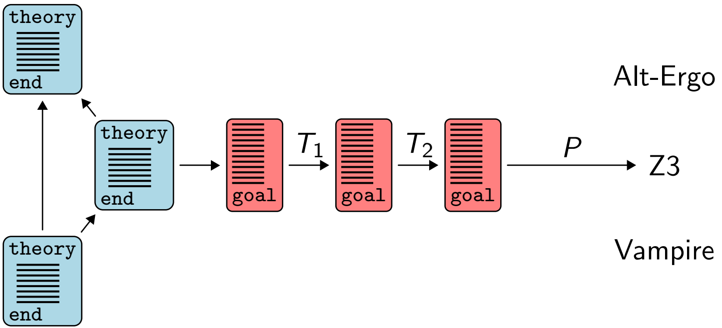
Vampire



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: *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
```

API

Why3 has an OCaml API

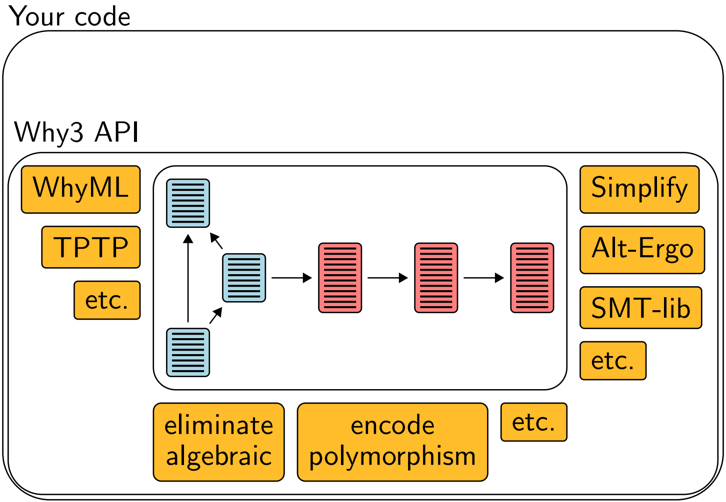
- to build terms, declarations, theories, tasks
- to call provers

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
  - Coq, SMT, TPTP, Gappa
- user-extensible system
  - input languages
  - transformations
  - output syntax
- efficient
  - e.g. transformations are memoized

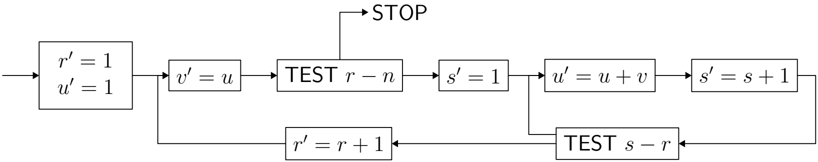
more details:

- *Why3: Shepherd your herd of provers.* (Boogie 2011)

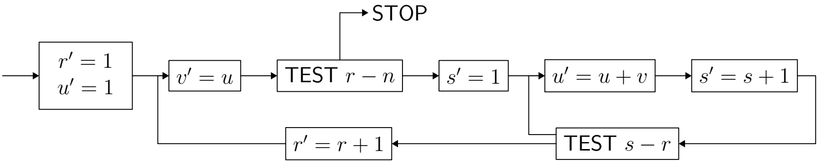
Part II

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)

Demo 3: another historical example

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

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

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

- 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

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

as show with function 91, proving termination may require to establish behavioral properties as well

another example:

- Floyd's cycle detection (Hare and Tortoise algorithm) up to now, we have only used integers

let us consider more complex data structures

- arrays
- algebraic data types

Why3 standard library provides arrays

```
use import array.Array
```

that is

- a polymorphic type

  `array` α

- an access operation, written

  ```
  a[e]
  ```

- an assignment operation, written

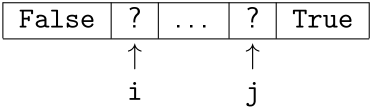
  `a[e1]` $\leftarrow$ `e2`

- operations `create`, `append`, `sub`, `copy`, etc.

Demo 4: 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
```



demo (access code)

Exercise 1: 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 . . .

```
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`

as for termination, proving safety (such as absence of array access our of bounds) may be arbitrarily difficult

an example:

- Knuth's algorithm for N first primes (TAOCP vol. 1)

Demo 5: Boyer-Moore's majority

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

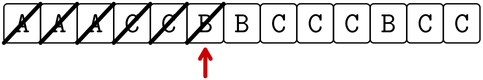
```
cand = A
k    = 3
```

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

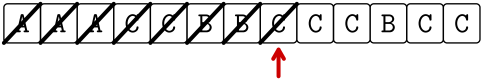
```
cand = A
k    = 1
```




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

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

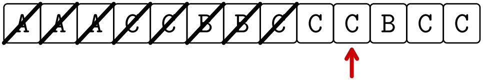
```
cand = B
k    = 1
```




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

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

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




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

~~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 * numof a result 0 (length a) > length a }
```

- postcondition in case of failure

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

each loop is given a loop invariant

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

for i = 0 to n-1 do
  invariant { !k = numof a !cand 0 i ∧ 2 * !k ≤ n }
  ...
```

the verification condition expresses

- safety
  - array access within bounds
  - termination
- validity of annotations
  - invariants are initialized and preserved
  - postconditions are established

automatically discharged by SMT solvers

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 may be removed without observable modification a circular buffer is implemented within an array

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

`len` elements are stored, starting at index `first`

				x_1	x_2	...	$x_{\texttt{len}}$		
				↑ `first`					

they may wrap around the array bounds

...	$x_{\texttt{len}}$							x_1	x_2
								↑ `first`	

we add an extra ghost field to model the buffer contents

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

ghost code is added to set this ghost field accordingly

example:

```
let push (b: buffer α) (x: α) : unit
  =
  ghost b.sequence ← b.sequence ++ Cons x Nil;
  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
```

we link the array contents and the ghost field with a type invariant

```
type buffer α =
  ...
invariant {
  let size = Array.length self.data in
  0 ≤ self.first < size ∧
  0 ≤ self.len   ≤ size ∧
  self.len = L.length self.sequence ∧
  ∀ i: int. 0 ≤ i < self.len →
    (self.first + i < size →
       nth i self.sequence =
       Some self.data[self.first + i]) ∧
    (0 ≤ self.first + i - size →
       nth i self.sequence =
       Some self.data[self.first + i - size])
}
```

such a type invariant

- is assumed at function entry
- must be ensured for values returned or modified

alternatively, we could have introduced a logical function mapping the buffer to a list

```
function buffer_model (b: buffer α) : list α
(* + suitable axioms *)
```

but ghost code

- is more compact
- results in simpler proof (it provides explicit witnesses)

a key idea of Hoare logic:

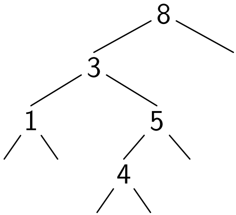
> *any types and symbols from the logic*
> *can be used in programs*

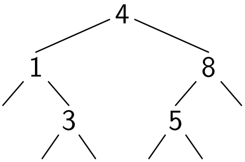
note: we already used type `int` this way we can do so with algebraic data types

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)
```

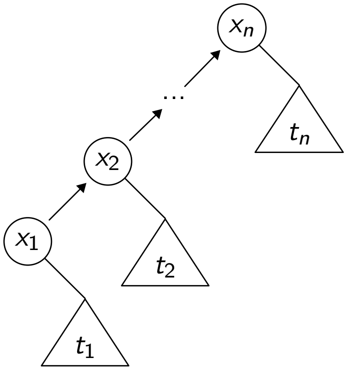
given two binary trees,
do they contain the same elements when traversed in order?



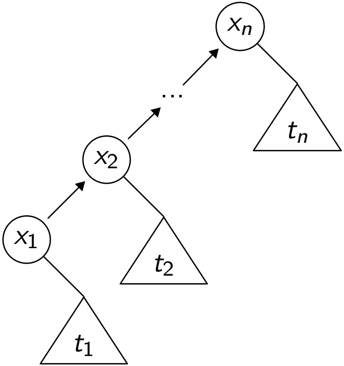


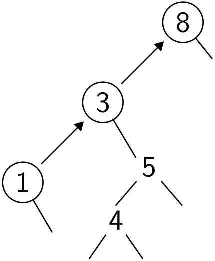
```
type elt

type tree =
  | Empty
  | Node tree elt tree

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 }
  =
  ...
```

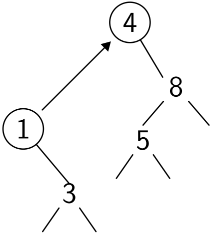
one solution: look at the left branch as a list, from bottom up



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 III

Modeling

in the library, we find

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

two meanings

- in programs, an abstract data type:

  ```
  type array α
  ```

- in the logic, an immutable record type:

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

one cannot define operations over type `array` α
(it is abstract) but one may declare them

examples:

```
val ([]) (a: array α) (i: int) : α reads {a}
  requires { 0 ≤ i < length a }
  ensures  { result = a[i] }

val ([]←) (a: array α) (i: int) (v: α) : unit writes {a}
  requires { 0 ≤ i < length a }
  ensures  { a.elts = M.set (old a.elts) i v }
```

one can model this way many data structures (be they implemented or not)

examples: stacks, queues, priority queues, graphs, etc.

```
type key

type t 'a

val create: int -> t 'a

val clear: t 'a -> unit

val add: t 'a -> key -> 'a -> unit

exception Not_found

val find: t 'a -> key -> 'a
```

```
type key

type t α model { mutable contents: map key (list α) }

val add (h: t α) (k: key) (v: α) : unit writes {h}
  ensures { h[k] = Cons v (old h)[k] }
  ensures { ∀ k': key. k' ≠ k → h[k'] = (old h)[k'] }

...
```

it is also possible to implement hash tables

```
type t α = { mutable size: int;
             mutable data: array (list (key, α)); }
  invariant ...
```

but it is (currently) not possible to prove that it implements the model from the previous slide

Another example: 32-bit 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
```

the integer value is given by

```
function toint int32 : int
```

within annotations, we only use type `int`

an expression x : `int32` appears, in annotations, as `toint` x

we define the range of 32-bit integers

```
function min_int: int = -2147483648
function max_int: int =  2147483647
```

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
```

modeling the heap

the second key idea of Hoare logic is

> *one can statically identify the various memory locations (absence of aliasing)*

in particular, memory locations are not first-class values

to handle programs with pointers,
one has to model the memory heap

consider for instance C programs with pointers of type `int*`

a possible model is

```
type pointer
val memory: ref (map pointer int)
```

the C expression

```
*p
```

is translated into the Why3 expression

```
!memory[p]
```

there are more subtle models
such as the *component-as-array* model (Burstall / Bornat)

each structure field is modeled as a separate map

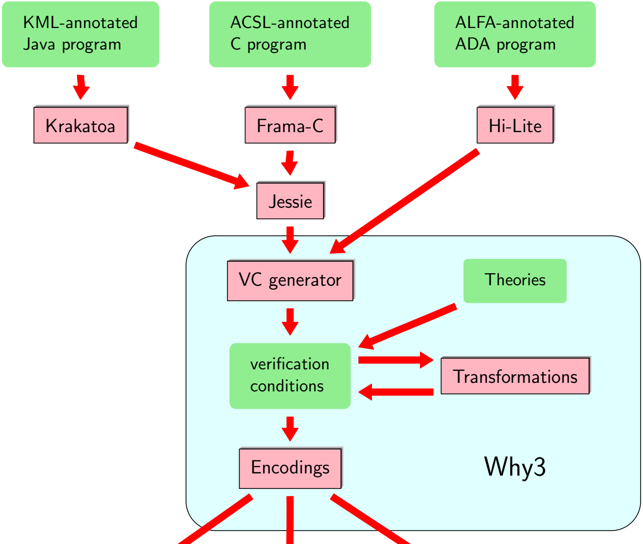
the C type

```
struct List {
  int head;
  struct List *next;
};
```

is modeled as

```
type pointer
val head: ref (map pointer int)
val next: ref (map pointer pointer)
```

such models are used in aforementioned tools for C, Java, and Ada



conclusion

Things not covered in this lecture

- how aliases are excluded
- how verification conditions are computed
- how formulas are sent to provers
- how floating-point arithmetic is modeled
- etc.

we saw 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 78 examples in our gallery)

- as an intermediate language
  (for the verification of C, Java, Ada, etc.)