Deductive Program Verification with Why3

Jean-Christophe Filliâtre
CNRS

Digicosme Spring School
April 22, 2013

http://why3.lri.fr/digicosme-spring-school-2013/
program + specification → verification conditions → proof
Tony Hoare.
Proof of a program: FIND.
which programs? which specs?

**Program** + **Specification** → **Verification Conditions** → **Proof**

**Programs**
- pseudo code / mainstream languages / DSL
- small / large

**Specs**
- safety, i.e. the program does not crash
- absence of arithmetic overflow
- complex behavioral property, e.g. “sorts an array”
which logic?

- too rich: we won’t be able to automate proofs
- too poor: we can’t model programming languages and we can’t specify programs

typically, a compromise
- e.g. first-order logic + equality + arithmetic
what about proofs?

program + specification \rightarrow verification conditions \rightarrow proof

a gift: theorem provers

- proof assistants: Coq, PVS, Isabelle, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- SMT solvers: CVC3, Z3, Yices, Alt-Ergo, etc.
- dedicated provers
a well-known technique: weakest preconditions
(Dijkstra 1971, Barnett/Leino 2005)

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

a well-known technique: weakest preconditions (Dijkstra 1971, Barnett/Leino 2005)

yet doing it for a realistic programming language is a lot of work

instead, we design an simpler language from which we extract VCs

two examples:
- Boogie (Microsoft Research)
- Why3 (Univ. Paris Sud / Inria)
Why3 in a nutshell

- a programming language, WhyML
  - polymorphism
  - pattern-matching
  - exceptions
  - mutable data structures, with controlled aliasing
  - a polymorphic first-order logic
    - algebraic data types
    - recursive definitions
    - inductive and coinductive 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.
some systems using Why3
Why3, bottom up

file.why -> Why -> transform/translate -> print/run

file.mlw -> WhyML -> VCgen

Coq, Alt-Ergo, CVC4, Z3, etc.
Part I

one logic to use them all
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

we make a compromise
logic of Why3 = polymorphic first-order logic, with

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

formal definition in

*Expressing Polymorphic Types in a Many-Sorted Language* (FroCos 2011)
*One Logic To Use Them All* (CADE 2013)
demo 1: the logic of Why3
• 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 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 (\texttt{use}) in a theory $T_2$
  • symbols of $T_1$ are \texttt{shared}
  • axioms of $T_1$ remain axioms
  • lemmas of $T_1$ become axioms
  • goals of $T_1$ are ignored

• cloned (\texttt{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
under the hood

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

Vampire

Z3

T1

T2

P
• 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
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.

eliminate algebraic
encode polymorphism
etc.

Simplify
Alt-Ergo
SMT-lib
etc.

API and plug-ins
• 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)

• *Preserving User Proofs Across Specification Changes* (VSTTE 2013)
Part II

program verification
demo 2: an historical example

demo 2: an historical example


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

\[ u \leftarrow 1 \]
for \( r = 0 \) to \( n - 1 \) do
\[ v \leftarrow u \]
for \( s = 1 \) to \( r \) do
\[ u \leftarrow u + v \]
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)
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)

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)
• **pre/postcondition**

\[
\text{let foo } x \ y \ z
\]
\[
\text{requires } \{ P \} \text{ ensures } \{ Q \}
\]
\[
= \ldots
\]

• **loop invariant**

\[
\text{while } \ldots \text{ do invariant } \{ I \} \ldots \text{ done}
\]
\[
\text{for } i = \ldots \text{ do invariant } \{ I(i) \} \ldots \text{ done}
\]
Recapitulation

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

\[ \text{variant } \{ t \} \text{ with } R \]

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

by default, \( t \) is of type \( \text{int} \) and \( R \) is the relation

\[ y \prec x \overset{\text{def}}{=} y < x \land 0 \leq x \]
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

```why3
use import array.Array
```

that is

- a polymorphic type
  ```why3
  array 'a
  ```

- an access operation, written
  ```why3
  a[e]
  ```

- an assignment operation, written
  ```why3
  a[e1] <- e2
  ```

- operations create, append, sub, copy, etc.
sort an array of Boolean, using the following algorithm

```ocaml
define 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

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

an example:
- Knuth’s algorithm for $N$ first primes (TAOCP vol. 1)
given a multiset of $N$ votes

\[ \text{A A A C C B B C C C C B C C} \]

determine the majority, if any
due to Boyer & Moore (1980)
cand = A
k = 1
cand = A
k = 2
cand = A
k = 3
cand = A
k = 2
cand = A
k = 1
cand = A
k = 0
cand = B
k = 1
cand = B
k = 0
cand = C
k = 1
cand = C
k = 2
cand = C
k = 1
cand = C
k = 2
cand = C
k = 3
cand = C
k = 3

then we check if C indeed has majority, with a second pass
(in that case, it has: 7 > 13/2)
SUBROUTINE MJRTY(A, N, BOOLE, CAND)
INTEGER N
INTEGER A
LOGICAL BOOLE
INTEGER CAND
INTEGER I
INTEGER K
DIMENSION A(N)
K = 0
C THE FOLLOWING DO IMPLEMENTS THE PAIRING PHASE. CAND IS
C THE CURRENTLY LEADING CANDIDATE AND K IS THE NUMBER OF
C UNPAIRED VOTES FOR CAND.
DO 100 I = 1, N
IF ((K .EQ. 0)) GOTO 50
IF ((CAND .EQ. A(I))) GOTO 75
K = (K - 1)
GOTO 100
50 CAND = A(I)
K = 1
GOTO 100
75 K = (K + 1)
100 CONTINUE
IF ((K .EQ. 0)) GOTO 300
BOOLE = .TRUE.
IF ((K .GT. (N / 2))) RETURN
C WE NOW ENTER THE COUNTING PHASE. BOOLE IS SET TO TRUE
C IN ANTICIPATION OF FINDING CAND IN THE MAJORITY. K IS
C USED AS THE RUNNING TALLY FOR CAND. WE EXIT AS SOON
C AS K EXCEEDS N/2.
K = 0
DO 200 I = 1, N
IF ((CAND .NE. A(I))) GOTO 200
K = (K + 1)
IF ((K .GT. (N / 2))) RETURN
200 CONTINUE
300 BOOLE = .FALSE.
RETURN
END
let mjrty (a: array candidate) =
    let n = length a in
    let cand = ref a[0] in let k = ref 0 in
    for i = 0 to n-1 do
        if !k = 0 then begin cand := a[i]; k := 1 end
        else if !cand = a[i] then incr k else decr k
    done;
    if !k = 0 then raise Not_found;
try
    if 2 * !k > n then raise Found; k := 0;
    for i = 0 to n-1 do
        if a[i] = !cand then begin
            incr k; if 2 * !k > n then raise Found
        end
    done;
    raise Not_found
with Found ->
    !cand
end
• precondition

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

• postcondition in case of success

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

• postcondition in case of failure

```plaintext
raises { Not_found ->
  forall c: candidate.
    2 * numof a c 0 (length a) <= length a }
```
each loop is given a loop invariant

\[
\text{for } i = 0 \text{ to } n-1 \text{ do }
\]
\[
\text{invariant } \{ 0 \leq !k \leq i \wedge
\]
\[
\text{numof a !cand 0 i } \geq !k \wedge
\]
\[
2 \times (\text{numof a !cand 0 i } - !k) \leq i - !k \wedge
\]
\[
\text{forall } c: \text{ candidate.}
\]
\[
\text{c } \not\leftrightarrow \text{ !cand } \rightarrow 2 \times \text{numof a c 0 i } \leq i - !k
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\[
\}
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
\]
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

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

len elements are stored, starting at index first

they may wrap around the array bounds
we add an extra ghost field to model the buffer contents

type buffer 'a = {
    mutable first: int;
    mutable len : int;
    data : array 'a;
    ghost mutable sequence: list 'a;
}
ghost code is added to set this ghost field accordingly

example:

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

```ocaml
type buffer 'a =
  ...

invariant { let size = Array.length self.data in
  0 <= self.first < size /
  0 <= self.len <= size /
  self.len = L.length self.sequence /
  forall 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 \texttt{function} mapping
the buffer to a list

\begin{verbatim}
function buffer_model (b: buffer 'a) : list 'a
(* + suitable axioms *)
\end{verbatim}

but ghost code

- is more compact
- results in simpler proofs (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

```plaintext
type bool = True | False  (in bool.Bool)
type option 'a = None | Some 'a  (in option.Option)
type list 'a = Nil | Cons 'a (list 'a)  (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
exercise 2: inorder traversal

```ml
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

controlled aliasing
only one kind of mutable data structure: records with **mutable fields**

for instance, references are defined this way

```ocaml
type ref 'a = { mutable contents : 'a }
```

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

type array 'a model { length: int; mutable elts: map int 'a }

keyword model instead of = makes a distinction

• in programs, array 'a is an abstract data type
• in the logic, array 'a is a (immutable) record type
one cannot define operations over type array ’a (it is abstract) but one may declare them

examples:

val ([]) (a: array ’a) (i: int) : ’a
requires { 0 <= i < length a }
ensures { result = a[i] }

val ([]<-) (a: array ’a) (i: int) (v: ’a) : unit
requires { 0 <= i < length a }
writes { a.elts }
ensures { a.elts = M.set (old a.elts) i v }
mutable data structures can be nested

example: hash tables

```ocaml
type t 'a = {
    mutable size: int;
    mutable data: array (list (key, 'a));
}
```

field data is mutable to allow resizing
but WhyML imposes a static control of aliasing

why? to get simpler verification conditions
how? using regions (internally)
consider hash tables again

```ocaml
type t 'a = {
  mutable size: int;
  mutable data: array (list (key, 'a));
}
```

a function resize (called from add) enlarges the bucket array

```ocaml
let resize (h: t 'a) : unit
  writes { h.data }
  =
    let nsize = 2 * Array.length h.data + 1 in
    let ndata = Array.make nsize Nil in
    ... rehash all values ...
    h.data <- ndata
```
then the following code is rejected

```ocaml
let alias (h: t int) (k: key) : unit =
  let old_data = h.data in
  add h k 42;
  old_data[0] <- Nil
```

with error

This expression prohibits further usage of variable old_data

indeed, add may call resize, and thus may invalidate old_data

more details:
*Why3 — Where Program Meet Provers (ESOP 2013)*
to use Why3 to verify programs with aliasing, you have to come up with a memory model

```plaintext
type loc
type value = ...
type state = map loc value
...
```

this is what is done for C, Java, Ada, etc.
consider for instance C programs with pointers of type int *

a possible model is

```plaintext
type loc
val memory: ref (map loc int)
```

the C expression

```plaintext
*p
```

is translated into the Why3 expression

```plaintext
!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

```c
struct List {
    int head;
    struct List *next;
};
```
is modeled as

```ocaml
type loc
val head: ref (map loc int)
val next: ref (map loc loc)
```
such models are used in tools for C, Java, and Ada
conclusion
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.)
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.

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