

an intermediate language for deductive program verification

Jean-Christophe Filliâtre

CNRS Orsay, France

AFM workshop

Grenoble, June 27, 2009









let us assume we want to do **deductive program verification** on realistic programs

- deductive verification means that we want to prove safety but also behavioral correctness, with arbitrary proof complexity
- realistic programs means pointers, aliases, dynamic allocation, arbitrary data structures, etc.

how do we proceed?

since **Hoare logic** (1968), we know how to turn a program correctness into logical formulas, the so-called verification conditions

we could

- design Hoare logic rules for a real programming language
- choose an interactive theorem prover

the Why approach: don't do that!

instead,

- design a small language dedicated to program verification and compile complex programs into it
- use as **many theorem provers** as possible (interactive or automatic)

there is another such tool: the **Boogie** tool developed at Microsoft Research, initially in the context of the SPEC# project (Barnett, Leino, Schulte)

there are differences but the main idea is the same: verification conditions should be computed on a small, dedicated language

O the WHY language

and its application to the verification of algorithms

 WHY as an intermediate language for program verification examples with C and Java the essence of Hoare logic fits in the rule for assignment

$$\overline{\{P[x \leftarrow E]\} \ x := E \ \{P\}}$$

two key ideas

- there is **no** alias, since only variable x is substituted
- the **pure** expression *E* belongs to both **logic** and **program**

 $\operatorname{W}{}_{\operatorname{HY}}$ captures these ideas

- programs can manipulate pure values (i.e. logical terms) arbitrarily
- the sole data structures are mutable variables containing pure values
- any program that would create an alias is rejected

- a $\operatorname{W}{}_{\operatorname{H}{\operatorname{Y}}}$ file contains
 - logical declarations

logic a : int logic f : int, int -> int axiom A : forall x: int. ... type set

variable/program declarations

parameter x : int ref
parameter p1 : a: int -> ...

program implementations

let p2 (x: int) (y: int) = ...

- a few types and symbols are predefined
 - a type int of arbitrary precision integers, with usual infix syntax
 - a type real of real numbers
 - a type **bool** of booleans
 - a singleton type unit

one nice idea inspired by ML:

no distinction between expressions and statements

- \Rightarrow less constructs, thus less rules
- \Rightarrow side-effects in expressions for free

but $\mathrm{W}\mathrm{H}\mathrm{Y}$ is not at all a functional language

let us check that n is even with the following (rather stupid) code

while $n \ge 1$ $n \leftarrow n-2$ return n = 0 we first introduce the predicate even, as an uninterpreted predicate with two axioms

```
logic even : int -> prop
axiom even0 :
    even(0)
axiom even2 :
    forall n: int. n >= 0 -> even(n) -> even(n+2)
```

the program $\texttt{is_even}$ is a function with n as argument

its body is a Hoare triple

```
let is_even (n: int) =
  { n >= 0 }
  ...
  { result=true -> even(n) }
```

in the postcondition, **result** is the returned value, i.e. the value of the function body, which simply is an expression

we introduce a local mutable variable x initialized to n

```
let is_even (n: int) =
  { n >= 0 }
  let x = ref n in
   ...
  { result=true -> even(n) }
```

finally, we add the while loop and its invariant

```
let is_even (n: int) =
  { n >= 0 }
  let x = ref n in
  while !x >= 2 do
      { invariant even(x) -> even(n) }
      x := !x - 2
  done;
  !x = 0
  { result=true -> even(n) }
```

A First Example

we are ready for program verification

two options

command line tool

why --smtlib even.why why --pvs even.why

• GUI to display verification conditions and launch provers



termination can be proved by adding a variant to the loop annotation

```
let is_even (n: int) =
  { n >= 0 }
  let x = ref n in
  while !x >= 2 do
      { invariant even(x) -> even(n)
        variant x }
      x := !x - 2
   done;
  !x = 0
  { result=true -> even(n) }
```

to get completeness, we add the axiom

```
axiom even_inv :
  forall n: int. even(n) -> n=0 or (n >= 2 and even(n-2))
```

and we turn the postcondition (and the invariant) into an equivalence

```
let is_even (n: int) =
  { n >= 0 }
  ...
  { result=true <-> even(n) }
```

a function argument can be a mutable variable

here, it simplifies the code

```
let is_even2 (n: int ref) =
  while !n >= 2 do
    n := !n - 2
  done;
  !n = 0
```

but it complicates the specification, since values of n at different program steps are now involved

in a postcondition, n@ stands for the value of n in the pre-state

```
let is_even2 (n: int ref) =
  { n >= 0 }
  ...
  { result=true <-> even(n@) }
```

more generally, a program point can be labelled (like for a goto) and then x@L stands for the value of x at point L

here it is used to refer to the value of n before the loop

```
let is_even2 (n: int ref) =
  { n >= 0 }
L:
  while !n >= 2 do
    { invariant even(n) <-> even(n@L) }
    ...
```

 $\rm WHY$ favors the use of labels instead of the traditional auxiliary variables, since it simplifies the VCs

note that it is yet possible to use auxiliary variables, if desired: simply add extra arguments to functions

 $\ensuremath{\mathrm{WHY}}$ supports recursive functions

```
let rec is_even_rec (n: int) : bool {variant n} =
  { n >= 0 }
  if n >= 2 then is_even_rec (n-2) else n=0
  { result=true <-> even(n) }
```

 WHY also features

- polymorphism, in both logic and programs
- exceptions in programs, and corresponding annotations
- Iocal assertions
- modularity, i.e. verification only depends on specifications

all of these features are illustrated in the following

let us consider a more complex program: Dijkstra's algorithm for single-source shortest path in a weighted graph

we are going to use W_{HY} to verify the **algorithm** i.e. a high-level pseudo-code, e.g. from the Cormen-Leiserson-Rivest, **not an actual implementation** in a given programming language

single-source shortest path in a weighted graph

 $S \leftarrow \emptyset$ $Q \leftarrow \{src\};$ $d[src] \leftarrow 0$ while $Q \setminus S$ not empty do extract u from $Q \setminus S$ with minimal distance d[u] $S \leftarrow S \cup \{u\}$ for each vertex v such that $u \xrightarrow{w} v$ $d[v] \leftarrow \min(d[v], d[u] + w)$ $Q \leftarrow Q \cup \{v\}$ we need finite sets for the program and its specification

- set of vertices V
- set of successors of *u*
- sets S and Q

all we need is

- the empty set \emptyset
- addition $\{x\} \cup s$
- subtraction $s \setminus \{x\}$
- membership predicate $x \in s$

let us axiomatize polymorphic sets

```
type 'a set
```

```
logic set_empty : 'a set
logic set_add : 'a, 'a set -> 'a set
logic set_rmv : 'a, 'a set -> 'a set
logic In : 'a, 'a set -> prop
```

```
predicate Is_empty(s : 'a set) =
  forall x: 'a. not In(x, s)
```

```
predicate Incl(s1 : 'a set, s2 : 'a set) =
  forall x: 'a. In(x, s1) -> In(x, s2)
```

```
axiom set_empty_def :
    Is_empty(set_empty)
```

```
axiom set_add_def :
  forall x: 'a. forall y: 'a. forall s: 'a set.
  In(x, set_add(y,s)) <-> (x = y or In(x, s))
```

```
axiom set_rmv_def :
  forall x: 'a. forall y: 'a. forall s: 'a set.
  In(x, set_rmv(y,s)) <-> (x <> y and In(x, s))
```

Dijkstra's Shortest Path: the Weighted Graph

the graph is introduced as follows

type vertex

logic V : vertex set

logic g_succ : vertex -> vertex set

axiom g_succ_sound : forall x: vertex. Incl(g_succ(x), V)

logic weight : vertex, vertex -> int (* a total function *)

axiom weight_nonneg : forall x,y: vertex. weight(x,y) >= 0

the set S of visited vertices is introduced as a global variable containing a value of type vertex set

```
parameter S : vertex set ref
```

to modify S, we could use assignment (:=) directly, but we can equivalently declare a function

```
parameter S_add :
    x: vertex -> {} unit writes S { S = set_add(x, S0) }
```

which reads as "function S_add takes a vertex x, has no precondition, returns nothing, modifies the contents of S and has postcondition $S = \text{set}_add(x, S@)$ "

Dijkstra's Shortest Path: the Priority Queue

we proceed similarly for the priority queue

```
parameter Q : vertex set ref
parameter Q_is_empty :
    unit ->
    { }
    bool reads Q
    { if result then Is_empty(Q) else not Is_empty(Q) }
```

```
parameter init :
   src: vertex -> {} ...
```

```
parameter relax :
    u: vertex -> v: vertex -> {} ...
```

17 VCs are generated

they are all automatically discharged, with the help of two lemmas

these two lemmas are proved using an interactive proof assistant (they require induction)

demo

using Why as an intermediate language

let us say we want to verify programs written in a language such as C or Java; what do we need?

- to cope with complex **data structures** (arrays, pointers, records, objects, etc.) and possible **aliasing**
- to cope with **new control statements** such as for loops, abrupt return, gotos, etc.
- to cope with function pointers, dynamic binding, etc.

 $\ensuremath{\mathrm{W}\mathrm{HY}}$ can be used conveniently to handle most of these aspects

two main parts

• we design a **memory model**, that is a set of logical types and operations to describe the memory layout

• we design a **compilation** process to translate programs in WHY constructs

A Simple Example

let us consider the following C code

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  while (1 <= u) {
    int m = (1 + u) / 2:
    if (t[m] < v)
      1 = m + 1:
    else if (t[m] > v)
    u = m - 1;
    else
      return m;
  }
  return -1;
```

two (simple) problems with this code

- C pointers (but no pointer arithmetic, i.e. arrays)
 int binary_search(int* t, int n, int v) { ...
- an abrupt return in the while loop

```
while (l <= u) {
    if ...
    else
        return m;
}</pre>
```

we consider a very simple memory model here

type pointer

type memory

logic get : memory, pointer, int -> int

parameter mem : memory ref
 (* the current state of the memory *)

some remarks at this point

- we assume a word (4 bytes) access of the memory; accessing the same portion of memory using a char* pointer would require a finer model
- \bullet C local variables can be translated into W_{HY} local variables, unless their address is taken

thus the code looks like

```
let binary_search (t: pointer) (n: int) (v: int) =
{ ... }
let l = ref 0 in
let u = ref (n-1) in
while !l <= !u do
    let m = (!l + !u) / 2 in
    if get !mem t m < v then l := m + 1
    else if get !mem t m > v then u := m - 1
    else ...
done
```

. . .

to interpret the return statement we introduce an exception

```
exception Return_int of int
```

the whole function body is put into a try/with statement

```
let binary_search (t: pointer) (n: int) (v: int) =
  try
   ...
  with Return_int r ->
    r
   end
```

and any return e is translated into

```
raise (Return_int e)
```

with suitable annotations for correctness, completeness and termination, we get 17 VCs

with the help of the axiom

axiom mean_1 : forall x,y: int. $x \le y \rightarrow x \le (x+y)/2 \le y$

all VCs are discharged automatically

demo

let us say we want to add array bound checking

we need to refine our model with a notion of block size

```
logic block_size : memory, pointer -> int
```

it is then convenient to introduce a function to access memory

```
parameter get_ :
    p: pointer -> ofs: int ->
    { 0 <= ofs < block_size(mem, p) }
    int reads mem
    { result = get(mem, p, ofs) }</pre>
```

so that its precondition introduces the suitable VC

we get 2 additional VCs, easily proved once we add the suitable requirement

let binary_search (t: pointer) (n: int) (v: int) =
 { n >= 0 and block_size(mem, t) >= n and ... }

. . .

finally, let us model 32 bit integers,

two possibilities

- to prove that there is no arithmetic overflow
- to model modulo arithmetic faithfully

one requirement:

we do not want to loose the arithmetic capabilities of the provers

we introduce a new type for 32 bit integers

type int32

the value of an int32 is given by

logic to_int: int32 -> int

annotations only use arbitrary prevision integers, i.e. if x of type int32 appears in an annotation, it is actually $to_{int}(x)$ we need to set the range of 32 bit integers

when using them...

axiom int32_domain: forall x: int32. -2147483648 <= to_int(x) <= 2147483647</pre>

... and when building them

```
parameter of_int :
    x: int ->
    { -2147483648 <= x <= 2147483647 }
    int32
    { to_int(result) = x }</pre>
```

and that's it!

let us prove the absence of integer overflow in binary search

demo

we found a bug (that was the purpose, after all)

indeed, when computing

int m = (1 + u) / 2;

the addition 1+u may overflow

(on a 32 bit architecture with arrays of billions of elements for instance)

it can be fixed as follows

int m = 1 + (u - 1) / 2;

Conclusion

regarding $\operatorname{W}\!\operatorname{H}\!\operatorname{Y}$ itself

- how to exclude aliases
- how to send VCs to all provers (typing systems differ)
- how to compute VCs efficiently

regarding the use of $\rm W{\ensuremath{\rm HY}}$

- how to design a high-level specification language
- how to design a more subtle memory model (component-as-array, regions, etc.)

in the **ProVal** team, we develop the following softwares

- Jessie, another intermediate language on top of WHY
- Krakatoa, a tool to verify JML-annotated Java programs

we also collaborate to **Frama-C**, a platform to verify C programs (which subsumes the tool Caduceus formerly developed at ProVal)



thank you