Proof Assistants
Proving Imperative Programs

Guillaume Melquiond
(Christine Paulin & Jean-Christophe Filliâtre)

INRIA Saclay – Île-de-France
Laboratoire de Recherche en Informatique

2011-02-01

Outline

1. Monads and Coq
2. Hoare Logic
   - Principles
   - Coq Implementation
   - The Why Tool
3. High-Level Datatypes
   - Handling Arrays
   - Advanced Memory Models

Monads

Used in Haskell to simulate imperative features.

A monad is given by

- M : Type → Type,
  M A represents computations of type A;
- return : $A \to$ M A (aka unit)
  return v represents the value v seen as the result of a computation;
- bind : M $A \to (A \to$ M $B) \to$ M B,
  bind passes the result of the first computation to the second one.

Haskell syntax: "`do p <- e1; e2`" means bind e_1 $(\lambda p.\, e_2)$.

Axioms of Monads

Monads have to satisfy the following laws:

1. $\texttt{bind}\ (\texttt{return}\ x)\ f = f\ x$,
2. $\texttt{bind}\ m\ \texttt{return} = m$,
3. $\texttt{bind}\ m\ (\lambda x.\, \texttt{bind}\ (f\ x)\ g) = \texttt{bind}\ (\texttt{bind}\ m\ f)\ g$.

Additional operators are defined to provide specific features to monads.

Combining State Monad and Error Monad

While it is generally impossible to merge two monads,
there are two ways of combining the state and error monads:

1. $\texttt{M}\ A = S \to (\texttt{Error} \mid \texttt{Value of}\ (S \times A))$,
2. $\texttt{M}\ A = S \to (S \times (\texttt{Error} \mid \texttt{Value of}\ A))$.

```
Inductive res {A: Type} : Type :=
  | Error: res
  | OK: A -> state -> res.

Definition M (A: Type) : Type := state -> @res A.
```

Error&State Monad in Coq

```
Definition ret {A: Type} (x: A) : M A :=
  fun s => OK x s.

Definition error {A: Type} : M A :=
  fun s => Error.

Definition bind {A B: Type} (f: M A) (g: A->M B) : M B :=
  fun s =>
    match f s with
    | Error => Error
    | OK a s' => g a s'
    end.

Notation "'do' X <- A ; B" := (bind A (fun X => B))
  (at level 200, X ident, A at level 100, B at level 200).
```

Example: Global Counter

Example (Global Counter Using a Reference)

```
let fresh = let r = ref 0 in fun () -> incr r; !r
```

```
Record state := { r: nat }.
Definition fresh : M nat := fun s =>
  let n := S (r s) in OK n {|r := n|}.

Fixpoint list_n (n: nat) : M (list nat) :=
  match n with
  | O => ret nil
  | S n' =>
    do k <- fresh;
    do l <- list_n n';
    ret (k :: l)
  end.
```

Outline

1. Monads and Coq

2. Hoare Logic
   - Principles
   - Coq Implementation
   - The Why Tool

3. High-Level Datatypes
   - Handling Arrays
   - Advanced Memory Models

Hoare Logic

Definition (Small Imperative Language)

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e \; op \; e \\
op & ::= & + \mid - \mid \times \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \text{and} \mid \text{or} \\
i & ::= & \texttt{skip} \mid x := e \mid i;\; i \mid \texttt{if}\; e \;\texttt{then}\; i \;\texttt{else}\; i \mid \texttt{while}\; e \;\texttt{do}\; i \;\texttt{done}
\end{array}
$$

Example (Program ISQRT)

```
count := 0; sum := 1;
while sum <= n do
  count := count + 1; sum := sum + 2 * count + 1
done
```

Specification: at the end of execution, count contains $\lfloor \sqrt{n} \rfloor$.

Operational Semantic

State E: function assigning values to variables.
It is extended to any expression and logical formulas:

$$\begin{aligned} E(n) &= n \\ E(e_1 \ op \ e_2) &= E(e_1) \ op \ E(e_2) \end{aligned}$$

Definition (Operational Semantic)

$$\begin{array}{llll}
E & \xrightarrow[x := e]{} & E[x \leftarrow E(e)] & \\
E_1 & \xrightarrow[i_1;\ i_2]{} & E_3 & \text{if } E_1 \xrightarrow[i_1]{} E_2 \text{ et } E_2 \xrightarrow[i_2]{} E_3 \\
E_1 & \xrightarrow[\texttt{if } e \texttt{ then } i_1 \texttt{ else } i_2]{} & E_2 & \text{if } E_1(e) = \mathit{true} \text{ and } E_1 \xrightarrow[i_1]{} E_2 \\
E_1 & \xrightarrow[\texttt{if } e \texttt{ then } i_1 \texttt{ else } i_2]{} & E_2 & \text{if } E_1(e) = \mathit{false} \text{ and } E_1 \xrightarrow[i_2]{} E_2 \\
E_1 & \xrightarrow[\texttt{while } e \texttt{ do } i \texttt{ done}]{} & E_3 & \text{if } E_1(e) = \mathit{true},\ E_1 \xrightarrow[i]{} E_2 \text{ and } E_2 \xrightarrow[\texttt{while } e \texttt{ do } i \texttt{ done}]{} E_3 \\
E & \xrightarrow[\texttt{while } e \texttt{ do } i \texttt{ done}]{} & E & \text{if } E(e) = \mathit{false}
\end{array}$$

Hoare Triple

Definition (Hoare Triple)

$$\{P\}\ i\ \{Q\}$$

with P and Q first-order formulas
using the same expressions (and variables!) than programs

Definition (Validity)

Triple $\{P\}\ i\ \{Q\}$ is valid if, for any states E_1 and E_2
such that $E_1(P)$ holds and $E_1 \underset{i}{\longrightarrow} E_2$, $E_2(Q)$ holds.

Example (Specification of ISQRT)

$\{n \geq 0\}\ ISQRT\ \{count^2 \leq n < (count + 1)^2\}$

Rules of Hoare Logic

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{\{P \land e = \mathit{true}\}\ i_1\ \{Q\} \qquad \{P \land e = \mathit{false}\}\ i_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2\ \{Q\}}$$

$$\frac{}{\{P[x \leftarrow e]\}\ x := e\ \{P\}} \qquad \frac{\{I \land e = \mathit{true}\}\ i\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ i\ \texttt{done}\ \{I \land e = \mathit{false}\}}$$

$$\frac{\{P\}\ i_1\ \{Q\} \qquad \{Q\}\ i_2\ \{R\}}{\{P\}\ i_1;\ i_2\ \{R\}} \qquad \frac{\{P'\}\ i\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ i\ \{Q\}}$$

Soundness and Completeness

- Soundness: every triple derived from the rules is valid.
- Practical issue: one has to guess the intermediate annotations (e.g., for ISQRT, a loop invariant is needed).
- Theoretical issue: is Hoare logic complete?
  That is, can every valid triple be derived from the rules?

Completeness and Weakest Preconditions

Given i et Q, $\{ P \mid \{P\}\ i\ \{Q\} \text{ valid } \}$ has some minimal elements P': for any P such that $\{P\}\ i\ \{Q\}$ valid, $P \Rightarrow P'$.

The weakest precondition $WP(i, Q)$ can be computed by induction on i:

$$
\begin{array}{rcl}
WP(x := e, Q) & = & Q[x \leftarrow e] \\
WP(i_1;\ i_2, Q) & = & WP(i_1, WP(i_2, Q)) \\
WP(\texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2, Q) & = & (e = \mathit{true} \Rightarrow WP(i_1, Q)) \wedge \\
 & & (e = \mathit{false} \Rightarrow WP(i_2, Q)) \\
WP(\texttt{while}\ e\ \texttt{do}\ i\ \texttt{done}, Q) & = & \bigvee H_k \text{ with } H_0 = (e = \mathit{false} \wedge Q) \\
 & & \text{and } H_{k+1} = (H_0 \vee WP(i, H_k))
\end{array}
$$

Formula for `while` is infinite hence illegal, but there exists an equivalent yet finite formula as long as the underlying logic is sufficiently expressive.

Coq Implementation

How to implement Hoare logic in Coq?

1. deep embedding

   Formalization of Hoare logic: terms, code, rules are internalized.

   $\Rightarrow$ useful to prove soundness or completeness but hardly usable in practice.

2. shallow embedding

   Programs are interpreted by their semantic (`Prop`) and Hoare rules are just tactics/theorems.

   $\Rightarrow$ usable on specific programs, but metatheory is now out of scope.

Imperative Features

Extensions to the basic language:

- allow side effects in expressions,
- access previous values of a variable in a logical formula,
- handle function/procedure calls (modularity),
- express loop/program termination,
- allow control flow jumps and exceptions,
- extend datatypes: arrays, records, pointers, objects, and so on.

The Why Tool

http://why.lri.fr/

- Functional language with imperative features (references, exceptions).
- Aliasing between references syntactically forbidden.
- Effect analysis: accessed variables, raised exceptions.
- Modularity in programs and specifications.
- Weakest precondition computation.
- Generate verification conditions for
  - proof assistants (Coq, Isabelle, PVS, HOL light) and
  - automated provers (Alt-Ergo, Simplify, CVC3, Z3, Gappa).

Example: Square Root

```
let isqrt (n:int) =
  { n >= 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum <= n do
    { invariant count >= 0 and n >= count*count
                and sum = (count+1)*(count+1)
      variant n - sum }
    count := !count + 1;
    sum := !sum + 2 * !count + 1
  done;
  !count
  { result >= 0 and
    result * result <= n < (result+1)*(result+1) }
```

Outline

1. Monads and Coq
2. Hoare Logic
   - Principles
   - Coq Implementation
   - The Why Tool
3. High-Level Datatypes
   - Handling Arrays
   - Advanced Memory Models

High-Level Datatypes

Hoare's assignment rule

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

implicitly states that

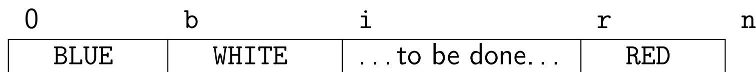
- expressions e are shared between programs and logical formulas,
- there is no aliasing between program variables.

Handling datatypes with Hoare logic requires some abstractions.

Example: Arrays and the Dutch National Flag

Dijkstra's problem: sort an array containing elements with only three possible values (blue, white, red).

Invariant for the algorithm:



Sorting Colors

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) {
  color c = t[i]; t[i] = t[j]; t[j] = c;
}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

Abstracting the C Code

The C code itself will not be checked directly.

1. Colors are replaced by an abstract type.
2. Arrays are replaced by references pointing to applicative arrays.

```
type color
logic blue : color
logic white : color
logic red : color

predicate is_color(c:color) =
  c = blue or c = white or c = red

parameter eq_color  :
  c1:color -> c2:color ->
    {} bool { if result then c1 = c2 else c1 <> c2 }
```

Abstract Type for Applicative Arrays

```
type color_array

logic acc : color_array, int -> color
logic upd : color_array, int, color -> color_array

axiom acc_upd_eq :
  forall a:color_array. forall i:int. forall c:color.
    acc(upd(a, i, c), i) = c

axiom acc_upd_neq :
  forall a:color_array. forall i,j:int. forall c:color.
    i <> j -> acc(upd(a, j, c), i) = acc(a, i)
```

Bounded Accesses to Arrays

```
logic length : color_array -> int

axiom length_update :
  forall a:color_array. forall i:int. forall c:color.
    length(upd(a, i, c)) = length(a)

parameter get :
  t:color_array ref -> i:int ->
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref -> i:int -> c:color ->
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

The swap Function

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@, i, acc(t@, j)), j, acc(t@, i)) }
```

Verification conditions are automatically verified by Alt-Ergo.

The Sorting Function

```
let dutch_flag (t:color_array ref) (n:int) =
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if eq_color (get t !i) blue then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if eq_color (get t !i) white then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
```

Specifying the Sort Function

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  forall k:int. i <= k < j -> acc(t, k) = c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    forall k:int. 0 <= k < n -> is_color(acc(t, k)) }
                            ⋮
  { exists b:int. exists r:int.
      monochrome(t, 0, b, blue) and
      monochrome(t, b, r, white) and
      monochrome(t, r, n, red) }
```

Loop Invariant

```
while !i < !r do
   { invariant 0 <= b <= i and i <= r <= n and
               monochrome(t, 0, b, blue) and
               monochrome(t, b, i, white) and
               monochrome(t, r, n, red) and
               length(t) = n and
               forall k:int. 0 <= k < n ->
                            is_color(acc(t, k))
     variant r - i }

                              ⋮

done
```

Verification Conditions

Verification conditions generated by WP:

1. The loop invariant is true before loop entry.
2. The loop invariant is preserved and the variant decreases.
3. Precondition of `swap` holds when it is called.
4. The array is not accessed out of bounds.
5. The postcondition holds at function exit.

All the conditions are automatically proved by Alt-Ergo!

Note: a better specification would state that the `dutch_flag` function does not modify the multiset of array elements.

Proving Java Programs: Global Memory Model

In Java, each object is a reference.
Two separate variables can point to the same memory location.

Simple memory model: memory is just a single huge array.
Attribute $e_1.a$ is modeled by $M[e_1 + \texttt{offset}(a)]$ with M a global memory.

Verification conditions grow too complex and are hardly provable,
because all memory accesses have to be checked for pointer equality.

Proving Java Programs: Burstall-Bornat Memory Model

In Java, if a et b are distinct class attributes,
then $e_1.a$ and $e_2.b$ cannot alias the same memory location.

Burstall-Bornat model: each attribute gets its own memory.
Attribute $e_1.a$ is modeled by $A[e_1]$ rather than $M[e_1 + \texttt{offset}(a)]$.
Similarly, integer arrays and object arrays can be separated.

A global table keeps track of allocated memory and dynamic types.

JML (Java Modeling Language)

```
class Purse {
    //@ public invariant balance >= 0;
    int balance;

    /*@ public normal_behavior
      @   requires s >= 0;
      @   modifiable balance;
      @   ensures balance == \old(balance) + s;
      @*/
    public void credit(int s) {
        balance += s;
    }
}
```

Exceptional Behaviors

```
/*@ public behavior
  @   requires s >= 0;
  @   modifiable balance;
  @   ensures s <= \old(balance) &&
  @           balance == \old(balance) - s;
  @   signals (NoCreditException)
  @           s > balance && balance == \old(balance);
  @*/

public void withdraw(int s) throws NoCreditException {
    if (balance >= s) { balance -= s; }
    else { throw new NoCreditException(); }
}
```

Translating Java to Why

- Modeling program and specification:
  - declarations and axioms backed by a Coq theory,
  - primitives describing the basic constructions of the language.
- Exceptions are used for describing control flow, e.g. `break`, `return`.

Coq Theory

```
Inductive value : Type :=
  | Null : value | Ref : adrObject -> value.

Inductive tag : Type :=
  | Obj : classId -> tag
  | Arr : Z -> Arrkind -> tag.

Definition alloc_table := pmap.t adrObject tag.

Definition memory A := map.t value A.

Definition mod_loc := value -> Prop.

Definition modifiable A (h:store) (m m':memory A)
                        (unchanged:mod_loc) : Prop :=
  forall v:value, alive h v -> unchanged v ->
  acc m v = acc m' v.
```

Why Theory: Declarations

```
type value
type tag
type classId
type javaType
type alloc_table
type 'a memory

logic acc : 'a memory, value -> 'a

logic Obj : classId -> tag
logic typeof : alloc_table, value, javaType -> prop

logic Null : -> value
```

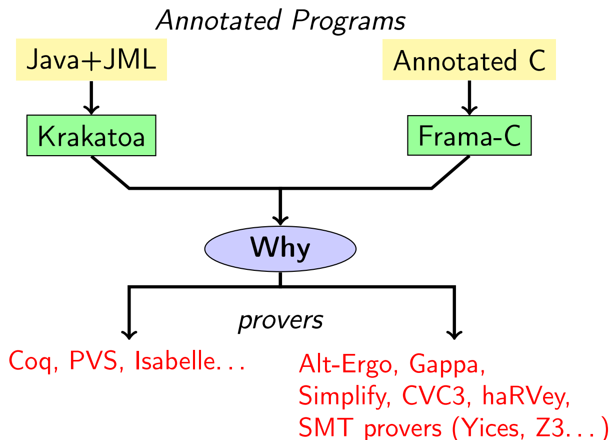
Why Theory: Language Primitives

```
exception Exception of value
exception Return_value of value

parameter alloc : alloc_table ref
parameter intA : int memarray ref
parameter objA : value memarray ref
parameter new_ref : alloc_table -> value
parameter allocate
   : alloc_table -> value -> tag -> alloc_table

parameter alloc_new_obj : c : classId ->
   { } value writes alloc
   { result <> Null
     and fresh(alloc@, result)
     and typeof(alloc, result, ClassType(c))
     and store_extends(alloc@, alloc) }
```

Why Architecture



Modeling C Code

- Structure fields act as Java attributes.
- Less type constraints in the language (pointer arithmetic, casts, etc).
- Either a lower-level memory model or improved static analyses.
- ACSL specification language.
- Handling `goto` requires assigning invariants to arbitrary program point.
- Some verified examples:
  - Schorr-Waite marking (pointer manipulation inside GC),
  - Safety of some Dassault code (70kloc, 116k conditions, 96% automatically proved),
  - Minix 3 string library, Verisec buffer overflow benchmark.