Verifying Imperative Programs with Coq

Jean-Christophe Filliâtre

SRI International

Kestrel Institute, December 11, 2000

Overview

1. Constructive proofs and extraction
2. How to deal with imperative programs
3. The tactic Correctness
4. Case studies
5. Comparisons with other works

Constructive proofs and Extraction

Coq implements a constructive logic, the Calculus of Inductive Constructions

A proof of $\exists x.P(x)$ is an explicit witness n together with a proof of $P(n)$

From each proof, one can extract a program corresponding to the informative part

Programs Extraction (C. Paulin, 89)

$$\pi : \forall x.P(x) \Rightarrow \exists y.Q(y) \xrightarrow{\text{Extraction}} \mathcal{E}(\pi) : X \rightarrow Y$$

realizability : $\forall x.P(x) \Rightarrow Q(\mathcal{E}(\pi)(x))$

Example

Inductive $Even$: nat $\to$ Prop :=

  | $EvenO$: ($Even$ O)

  | $EvenS$: (n:nat)($Odd\ n$) $\to$ ($Even$ (S n))

with Odd : nat $\to$ Prop :=

  | $OddS$: (n:nat)($Even\ n$) $\to$ (Odd (S n)).

Lemma $even\_odd$:

  (n:nat) { b:bool | (b=true $\wedge$ ($Even\ n$)) $\vee$ (b=false $\wedge$ ($Odd\ n$)) }.

Proof.

Induction n.

...

Example (cont'd)

Require Extraction.

Write Caml File "even_odd" [*even_odd*].

`even_odd.ml`

```
type bool = True | False
type nat = O | S of nat
let even_odd n =
  let rec f1 = function
     O → True
  | S n1 → (match f1 n1 with True → False | False → True)
  in f1 n
```

The tactic Program (C. Parent, 95)

$$
\begin{tikzcd}
\widehat{\pi} : \forall x.P(x) \Rightarrow \exists y.Q(y) \arrow[d, "\text{resolution of obligations}"'] \arrow[dr, bend left] & \\
\pi : \forall x.P(x) \Rightarrow \exists y.Q(y) \arrow[r, "\text{Extraction}"] & f : X \to Y \arrow[ul, bend right, "\text{Program}"']
\end{tikzcd}
$$

realizability : $\forall x.P(x) \Rightarrow Q(f(x))$

What about imperative programs?

Two main issues

- how to specify the programs?

  $\Rightarrow$ Floyd-Hoare logic

- how to establish the correctness?

  $\Rightarrow$ functional interpretation i.e. denotational semantics

Our approach

$$
\begin{tikzcd}
\widehat{e} : \forall x. P(x) \Rightarrow \exists y. Q(y) \arrow[d, "\text{resolution of obligations}"'] \arrow[dr] & \{P\}\ e\ \{Q\} \arrow[l, "\text{partial proof}"'] \arrow[d, "\text{functional translation}"] \\
\pi : \forall x. P(x) \Rightarrow \exists y. Q(y) \arrow[r, "\text{Extraction}"] & \overline{e} : X \rightarrow Y
\end{tikzcd}
$$

realizability : $\forall x. P(x) \Rightarrow Q(\overline{e}(x))$

A natural interpretation

Denotational semantics traditionally interprets imperative programs as stores transformers

This is not natural, and not suitable for reasoning about programs

We expect references to be interpreted as functional variables

Example

$i = 0;$

while $(i < n)$ $\{$

  if $(p(a[i]))$ $\{$ break; $\}$

  i++;

$\}$

let rec $lookup$ i $=$

  if $i = n$ || $(p\ a.[i])$ then i

  else $lookup$ $(i + 1)$

in

$lookup$ 0

Example (cont'd)

$i = 0;$

while $(i < n)$ {

  if $(p(a[i]))$ { break; }

  i++;

}

let rec *lookup* i =

  if $i = n \mid\mid (p\ a.[i])$ then i

  else *lookup* $(i + 1)$

in

lookup 0

$$\forall j.\ 0 \leqslant j < i \Rightarrow \neg p(a[j])$$

Effects

Standard typing with effect à la Talpin & Jouvelot

$$\Gamma \vdash e : (\tau, \epsilon)$$

Effects are sets of read and write variables $\epsilon = (\rho, \omega)$

$$\begin{array}{llll} \text{values} & \tau & ::= & \beta \mid \beta\ \mathsf{ref} \mid (x : \tau) \to \kappa \\ \text{computations} & \kappa & ::= & (\tau, \epsilon) \end{array}$$

Aliases

Aliases are excluded

$$f \quad \equiv \quad \mathsf{fun}\ x\ y \rightarrow x\ := !x + 1;\ \ y\ := !y + 1$$

$$\vdash (f\ \ r) : (y : \mathsf{int\ ref}) \rightarrow (\mathsf{unit}, \{r, y\})$$

but $\quad ((f\ \ r)\ \ r) \quad$ cannot be typed

$$\frac{\Gamma \vdash e : ((x : \tau_1\ \mathsf{ref}) \rightarrow (\tau_2, \epsilon_2), \epsilon) \qquad r : \tau_1\ \mathsf{ref} \in \Gamma \qquad r \not\in (\tau_2, \epsilon_2)}{\Gamma \vdash (e\ \ r) : (\tau_2[x \leftarrow r], \epsilon \sqcup \epsilon_2[x \leftarrow r])}$$

Monads parameterized by effects

Usual state monads are too coarse ($\mu\ \tau \equiv S \to S \times \tau$)

$$\begin{aligned} \mathsf{unit} &: \tau \to \mu\ \tau \\ \mathsf{star} &: \mu\ \tau \to (\tau \to \mu\ \tau') \to \mu\ \tau' \end{aligned}$$

We introduce a notion of monads parameterized by effects.
$\mu_\epsilon\ \tau$ represents a computation of type τ and effect ϵ

$$\begin{aligned} \mathsf{unit}_{\epsilon_1,\epsilon_2} &: \mu_{\epsilon_1}\ \tau \to \mu_{\epsilon_2}\ \tau \\ \mathsf{star}_{\epsilon_1,\epsilon_2} &: \mu_{\epsilon_1}\ \tau \to (\tau \to \mu_{\epsilon_2}\ \tau') \to \mu_{\epsilon_1 \sqcup \epsilon_2}\ \tau' \end{aligned}$$

The main steps

1. annotated program $\{P\}\ e\ \{Q\}$
2. type and effect inference $\vdash e : (\tau, \epsilon)$

   with $\epsilon = (\{x_1, \dots, x_n\}, \{y_1, \dots, y_m\})$
3. functional translation

   $\overline{e} : \mu_\epsilon\ \tau = X_1 \times \cdots \times X_n \to Y_1 \times \cdots \times Y_m \times T$
4. goal $\forall \vec{x}.\ P(\vec{x}) \Rightarrow \exists \vec{y}, v.\ Q(\vec{x}, \vec{y}, v)$

   refined using a partial proof term built upon $\overline{e}$

Example

$$\{\ (Even\ n)\ \}$$

$$n\ \ :=\ \ !n \,/\, 2;$$

$$k\ \ :=\ \ !k + 1;$$

$$\{\ n \cdot 2^{k} = \overleftarrow{n} \cdot 2^{\overleftarrow{k}}\ \}$$

$$\lambda(n,k).\ \lambda h : (Even\ n).$$

$$\text{let } n_1 = n/2 \text{ in}$$

$$\text{let } k_1 = k + 1 \text{ in}$$

$$((n_1, k_1), \text{void}, ? : n_1 \cdot 2^{k_1} = n \cdot 2^{k})$$

The proof obligation

After reduction of the proof term, we get

$$\frac{\begin{array}{l} n, k : \mathsf{nat} \\ h : (Even\ n) \end{array}}{n/2 \cdot 2^{k+1} = n \cdot 2^k}$$

Correctness

If the partial proof term is completed (i.e. if the obligations are proved) then the program terminates and satisfies the specification.

Completeness

If the program terminates and satisfies its specification, then it can be annotated in such a way that all the obligations are provable.

(loops and recursive functions are not considered here)

The tactic Correctness

- The programming language
- The specifications
- The proof obligations

The programming language

- any Coq type can be used as a datatype
  (shallow embedding)
- references and arrays of purely functional values
- sequences, conditionals, loops, local variables
- functions as first-class values, recursive functions
- call-by-value and call-by-reference
- restricted polymorphism

Specifications

- pre- and post-conditions anywhere, written $\{P\}\ e\ \{Q\}$
- loop invariants and variants

  while b do { [invariant I] variant ϕ [for R] } e done

- variants for recursive functions

  let rec f $(\overrightarrow{x : T})$: T { variant ϕ [for R] } $= e$

- assertions (for convenience)

  begin $e_1; \ldots;$ assert { P }; $e_n; \ldots$ end

Auxiliary variables

. . . are replaced by a notation for the successive values of a reference

- $x@$ stands for the value of x before

$$\ldots \{P\}\ e\ \{\ldots x@ \ldots\} \ldots$$

- $x@0$ stands for the initial value of x
- $x@L$ stands for the value of x at label L

$$\ldots;\ \text{label } L;\ \ \ldots;\ \ \ldots \{\ldots x@L \ldots\}\ \ \ldots$$

Proof obligations

are the usual ones of Floyd-Hoare logic

- conditional $\Rightarrow$ two proof obligations, each one under the right hypothesis
- loop $\Rightarrow$ three proof obligations : initial invariant, preservation of invariant, termination
- etc.

Expected context of hypotheses

$\{ \ (Even\ n) \ \}$

while $!n > 0$ do

$\{$ invariant $(Even\ n)$ variant n $\}$

$n \ := \ !n - 2;$

done

$\{ \ n = 0 \ \}$

Arrays

Arrays are seen as references on a functional and abstract notion of array

$$\begin{array}{lcl} \mathsf{access} & : & (\mathsf{array}\ N\ T) \to Z \to T \\ \mathsf{store} & : & (\mathsf{array}\ N\ T) \to Z \to T \to (\mathsf{array}\ N\ T) \end{array}$$

$$\frac{\begin{array}{l} t_0 : (\mathsf{array}\ N\ T) \\ t_1 : (\mathsf{array}\ N\ T) \end{array}}{(\mathsf{access}\ t_1\ i) < \underbrace{(\mathsf{access}\ (\mathsf{store}\ t_0\ i\ v)\ i)}_{reduced\ to\ v}}$$

Case studies

- Find (Hoare, 1971)
- Quicksort
- Heapsort
- Knuth-Pratt-Morris
- Binary Search

Statistics

	lines of code	number annot.	number oblig.	lines proof	total time
Find (Hoare, 71)	27 (1)	9	22	619	—
Quicksort	41 (4)	18	26	468	2 days
Heapsort	19 (2)	7	22	626	3 days
Knuth-Morris-Pratt	24 (2)	8	23	377	1,5 days
Binary Search	12 (1)	3	8	146	3 hours

Proving programs...

- is great fun!
- is not so hard (a few days for a complex algorithm)
- requires high mathematics, to specify and to prove
- automatic proofs seem hopeless
- the linear arithmetic decision procedure (Omega) was really useful

Use of inductive predicates

The $heap$ property is an inductive predicate

$$\frac{\begin{array}{c} 2k+1 \leq n \Rightarrow (t[k] \geq t[2k+1] \wedge (heap\ t\ n\ (2k+1))) \\ 2k+2 \leq n \Rightarrow (t[k] \geq t[2k+2] \wedge (heap\ t\ n\ (2k+2))) \end{array}}{(heap\ t\ n\ k)}$$

Some proof obligations were established using the corresponding induction principle, and others using the induction principle

$P(0)\ \wedge\ \forall n.\ P(n) \Rightarrow P(2 \cdot n+1) \wedge P(2 \cdot n+2) \Rightarrow \forall n.\ P(n)$

Other case studies

- Floyd's algorithm (all shortest paths in a weighted graph) (Christine Paulin)
- Petersson's mutual exclusion algorithm (Eduardo Giménez)
- Insertion sort (Nicolas Magaud)
- RSA (Jose-Carlos Almeida)

Comparisons with other works

Shallow embeddings

- Hoare logic in HOL (M. Gordon, 96)
- B method in PVS (C. Muñoz, 98)

Deep embeddings

- Hoare logic in Lego (T. Kleymann, 97)
- B method in Isabelle/HOL (P. Chartier, 98)
- Hoare logic in HOL (P. Homeier, 96)

Conclusion

We propose a method to certify imperative programs which

- combines a realistic programming language, a good specification language (CIC) and a good prover
- is implemented in Coq (distributed with the system)
- has been successfully applied to non-trivial case studies

Future work

- exceptions
- pattern-matching
- a purely functional case study
  (and to unify the tactics Program and Correctness)

Availability

coq.inria.fr

www.csl.sri.com/~filliatr

- a 40 pages english translation of my thesis
- a paper about generalized monads
- papers describings the case studies