Cameleer
a Deductive Verification Tool for OCaml

Mário Pereira

NOVA–LINCS, Nova School of Science and Technology, Portugal

Cambium Seminar, November 8th
“[…] it is a pity that we do not, today, have mature tools for checking the correctness of functional programs”

Régis-Gianas & Pottier, MPC’08

Many research tools on deductive verification of imperative languages: Boogie, Dafny, VeriFast, Frama-C, VerCors, ...

But seldom applied to the functional world

Pick OCaml, for instance:
• clear syntax
• well-defined execution model (operational semantics)
• multi-paradigm

Language of choice to implement “critical” software

And yet…
Our Quest

... there is no **usable** deductive verification tool for OCaml code
  X write code in a proof-aware language, then extract
  X use an interactive tool

Let us build a **deductive verification tool for OCaml programmers**
  • code and specification should live and evolve together

What do we need and where to go?
  • a **specification language for OCaml**
  • a VCGen to compute **verification conditions**
  • SMTs to automatically discharge these verification conditions
The Cameleer Project

**Cameleer**: Principles and Methods to Verify OCaml Programs

[https://cordis.europa.eu/project/id/897873](https://cordis.europa.eu/project/id/897873)

Marie Sklodowska-Curie individual fellowship (June’20 – May’22)

The first automated tool for the verification of OCaml programs

Key ingredients:

- GOSPEL, *Generic OCaml SPEcification Language*
- translation into the *Why3* framework  
  
  [Bobot et al.]
Our research expectations

Specification that OCaml programmers can **read**, and **even write**

Use a **proof environment**, but **abstract it**

Clear focus towards **proof automation**

Dance with Gospel: **evolve it**, make it a **mature proof language**
In this talk

Practical aspects regarding:

1. GOSPEL specification language

   \[(\text{ensures } r > x)\]

2. Deductive verification of OCaml programs: the Cameleer tool

   \[
   \text{let next } x = x + 42 \\
   (\text{ensures } r > x) \\
   \]

3. Cameleer as a research vehicle

   \[
   \text{let do_next } x = \\
   \text{next } x \ (\text{fun } o \rightarrow (\text{ensures } \text{result } > x) \ o + 42) \\
   \]
The Tale of the Specifying Programmer
(** The type of queues containing elements of type [’a]. *)

`type ’a t`

(** Return a new queue, initially empty. *)

`val create: unit -> ’a t`

(** [push x q] adds the element [x] at the end of the queue [q]. *)

`val push: ’a -> ’a t -> unit`

(** [pop] removes and returns the first element in queue [q]. *)

`val pop : ’a t -> ’a`
The type of queues containing elements of type ['a]. *)

** The type of queues containing elements of type ['a]. *)

```ml
type 'a t
(*@ predicate R: loc -> 'a list -> Heap -> Prop *)
```

** Return a new queue, initially empty. *)

```ml
val create: unit -> 'a t
(*@ create ()
    ensures λq. ∃L. (R q L) ⋆ [L = nil] *)
```

** [push x q] adds the element [x] at the end of the queue [q]. *)

```ml
val push: 'a -> 'a t -> unit
(*@ push v q
    requires (R q L)
    ensures λu. ∃L’. (R q L’) ⋆ [L’ = L ++ v::nil] *)
```

** [pop] removes and returns the first element in queue [q]. *)

```ml
val pop : 'a t -> 'a
(*@ pop q
    requires (R q L) ⋆ [L <> nil]
    ensures λv. ∃L’. (R q L’) ⋆ [L = v :: L’] *)
```
type 'a t
(*@ mutable model view: 'a list *)

val create: unit -> 'a t
(*@ q = create ()
   ensures q.view = [] *)

val push: 'a -> 'a t -> unit
(*@ push v q
   modifies q
   ensures q.view = old q.view ++ v :: [] *)

val pop : 'a t -> 'a
(*@ v = pop q
   requires q.view <> []
   modifies q
   ensures old q.view = v :: q.view *)
The GOSPEL specification language

- expressive
  - user writes specification in (an extension of) first-order logic
- understandable by OCaml programmers
  - Gospel terms are a subset of OCaml + quantifiers
- concise (improvement over Separation Logic)
- not tied to any particular verification tool
- formal semantics, via a translation to Separation Logic

A Chaguéraud, J-C Filliâtre, C. Lourenço, M. Pereira
“GOSPEL – Providing OCaml with a Formal Specification Language”

Recently released
Try it yourself: opam install gospel
Gospel to Separation Logic – an Example

Merge $q_1$ into $q_2$, then clearing $q_1$:

```ocaml
val in_place_concat: 'a t -> 'a t -> unit
(*@ concat q1 q2
  modifies q1, q2
  ensures q1.view = empty
  ensures q2.view = old q2.view ++ old q1.view *)
```

Translation into Separation Logic:

```
{ (R q1 L1) ∗ (R q2 L2) }
in_place_concat q1 q2
{ λ_. ∃L1’ L2’. (R q1 L1’) ∗ (R q2 L2’) ∗
  [L1’ = nil ∧ L2’ = L2++L1] }
```
Verifying OCaml Code With GOSPEL
The Three Musketeers

Why3

- first-order logic, weakest preconditions
- VCs sent to automated theorem provers
- targets imperative programs with limited mutability

Coq

- automated translation to OCaml
- targets purely applicative programming

CFML

- higher-order Separation Logic, within Coq
- targets pointer programs
Gospel used only in signature files.
Gospel used only in *signature files.*
Gospel used only in *signature files.*

None is Perfect

Why3

Coq

CFML

OCaml
The Fourth Musketeer: Cameleer
Why3, Ancien Régime

WhyML → extraction → OCaml

Why3 logic

... → Z3 → Alt-Ergo → CVC4 → Coq
La Révolution Cameleer

OCaml + Gospel (ppxlib)

automatic translation (Cameleer)

WhyML

Why3 logic

Z3  Alt-Ergo  CVC4  Coq
Support for core OCaml + functors

- no objects
- no GADTs or polymorphic variants

Limited support for

- higher-order functions
- mutability

Our translation is defined as a set of inference rules

Today: overview of translation via examples
The OCaml side:

```ocaml
let int_sqrt x =
    let count = ref 0 in
    let sum = ref 1 in
    while !sum <= x do
        (*@ invariant !count >= 0
        invariant x >= sqr !count
        invariant !sum = sqr (!count + 1)
        variant x - !count *)
        count := !count + 1;
        sum := !sum + (2 * !count + 1)
    done;
    !count

(*@ r = int_sqrt x
  requires x >= 0
  ensures int_sqrt_spec x r *)
```

The OCaml side:

```ocaml
let int_sqrt x =  
  let count = ref 0 in  
  let sum = ref 1 in  
  while
    [@gospel {|
      invariant !count >= 0
      invariant x >= sqr !count
      invariant !sum = sqr (!count + 1)
      variant x - !count |
    }] !sum <= x do  
    count := !count + 1;  
    sum := !sum + (2 * !count + 1)  
  done;  
  !count
[@@gospel {|
  r = int_sqrt x
  requires x >= 0
  ensures int_sqrt_spec x r |
}]```

The OCaml side:

```ocaml
let int_sqrt x =  
  let count = ref 0 in  
  let sum = ref 1 in  
  while  
    [@gospel { | invariant !count >= 0  
      invariant x >= sqr !count  
      invariant !sum = sqr (!count + 1)  
      variant x - !count |}] !sum <= x do  
    count := !count + 1;  
    sum := !sum + (2 * !count + 1)  
  done;  
  !count  
[@@gospel { | r = int_sqrt x  
    requires x >= 0  
    ensures int_sqrt_spec x r |}]```

The WhyML side:

```whyml
let int_sqrt x  
  requires { x >= 0 }  
  ensures { int_sqrt_spec x r }  
= let ref count = 0 in  
  let ref sum = 1 in  
  while sum <= x do  
    invariant { count >= 0 }  
    invariant { x >= sqr count }  
    invariant { sum = sqr (count + 1) }  
    variant { x - !count }  
    count <- count + 1;  
    sum <- sum + (2 * count + 1)  
  done;  
  count```
Translation of Type Declarations and \texttt{assert false}

The OCaml side:

\begin{verbatim}

    type 'a non_empty_list = {
      self: 'a list
    } (*@ invariant self <> [] *)

    let hd (l: 'a non_empty_list) =
      match l with
      | [] -> assert false
      | x :: _ -> x
      (*@ r = hd l
       ensures match l with
       | [] -> false
       | x :: _ -> r = x *)

\end{verbatim}
Translation of Type Declarations and \texttt{assert false}

The \texttt{OCaml} side:

\begin{verbatim}
type 'a non_empty_list = {
  self: 'a list
} (@ invariant self <> [])

let hd (l: 'a non_empty_list) = 
  match l with
  | [] -> assert false
  | x :: _ -> x
(*@ r = hd l
  ensures match l with
    | [] -> false
    | x :: _ -> r = x *)
\end{verbatim}

The \texttt{WhyML} side:

\begin{verbatim}
type non_empty_list 'a = {
  self: list 'a
} invariant { self <> Nil }

let hd (l: non_empty_list 'a)
  returns { r ->
    match l with
      | Nil -> false
      | Cons x _ -> x = r end }

= match l with
  | [] -> absurd
  | Cons x _ -> x end

\texttt{assert false} is treated in a special way by
the \texttt{OCaml} type-checker.
\end{verbatim}
The OCaml side:

```ocaml
module type PARTIAL_ORD = sig
  type t
  val leq : t -> t -> bool
end

module Make (E: PARTIAL_ORD) = struct
  type elt = E.t
  type t = E | N of int * elt * t * t

  let _make_node x a b =
    if _rank a >= _rank b then N (_rank b + 1, x, a, b)
    else N (_rank a + 1, x, b, a)

  let rec merge t1 t2 = match t1, t2 with
    | t, E | E, t -> t
    | N (_, x, a1, b1), N (_, y, a2, b2) ->
      if E.leq x y then _make_node x a1 (merge b1 t2)
      else _make_node y a2 (merge t1 b2)
end
```
The **WhyML** side:

```whyml
scope Make
  scope E
    type t
      val leq (x: t) (y: t): bool
  end

type elt = E.t
type t = E | N int elt t t

let _make_node x a b =
  if _rank a >= _rank b
  then N (_rank b + 1) x a b
  else N (_rank a + 1) x b a

let rec merge t1 t2 = match t1, t2 with
  | t, E | E, t -> t
  | N _ x a1 b1, N _ y a2 b2 ->
    if E.leq x y
    then _make_node x a1 (merge b1 t2)
    else _make_node y a2 (merge t1 b2)
  end
end
```
A Demo is Worth a Thousand Words
Extensions to Core Cameleer
Q: Compute the height of a binary tree, without stack-overflow
A: CPS transformation

```ocaml
type 'a t = E | N of 'a t * 'a * 'a t

let rec height t k = match t with
  | E -> k 0
  | N (l, _, r) ->
    height l (fun hl ->
      height r (fun hr -> k (1 + max hl hr)))

let main t = height t (fun x -> x)
```
Defunctionalization

type 'a kont =
  | Kid
  | Kleft of 'a tree * 'a kont
  | Kright of 'a kont * int

let rec height t k = match t with
  | E -> apply k 0
  | N (l, _, r) -> height l (KLeft (r, k))
and apply k arg = match k with
  | Kid ->
    let x = arg in x
  | Kleft (r, k) ->
    let hl = arg in height r (Kright (k, hl))
  | Kright (k, hl) ->
    let hr = arg in apply (1 + max hl hr) k
R Q: can use defunctionalization as a proof technique?

How to specify higher-order functions [Régis-Gianas & Pottier, MPC’08]

\[
\begin{align*}
  & f : \tau_1 \to \tau_2 \\
  & \text{pre } f : \tau_1 \to \text{prop} \\
  & \text{post } f : \tau_1 \to \tau_2 \to \text{prop}
\end{align*}
\]

Our recipe:

• extend Gospel with pre and post predicates
• use Gospel to provide specification to higher-order programs
• translate the code and specification into first-order
  - implement defunctionalization on top of Cameleer
• let Cameleer do the rest

Gospel specification:

(*@ function H (t: 'a tree) : int = match t with
    / E -> 0
    / N (l, _, r) -> 1 + (max (H l) (H r)) *)

let rec height t k = match t with
    | E -> k 0
    | N (l, _, r) ->
        height l (fun hl ->
            height r (fun hr -> k (1 + max hl hr)))

(*@ r = height t k
    *)

let main t = height t (fun x -> x)
Gospel specification:

(*@ function H (t: 'a tree) : int = match t with
  | E -> 0
  | N (l, _, r) -> 1 + (max (H l) (H r)) *@)

let rec height t k = match t with
  | E -> k 0
  | N (l, _, r) ->
    height l (fun hl ->
      (*@ ensures post k result (1 + max hl (H r)) *)
      height r (fun hr -> k (1 + max hl hr)))
    (*@ ensures post k result (1 + max hl hr) *)
  (*@ r = height t k
    ensures post k r (H t) *)

let main t = height t (fun x -> x)
  (*@ r = main t ensures r = H t *)
Verified CPS-height of a Tree (2/2)

Automatically generated by Cameleer:

\[
(*@ \text{predicate } \text{post} \ (k: \ 'a \ \text{kont}) \ (\text{res: int}) \ (\text{arg: int}) = \ \\
\text{match } k \ \text{with} \\
\quad | \text{Kid} \rightarrow \text{res} = \text{arg} \\
\quad | \text{Kleft} \ (r, \ k) \rightarrow \text{post} \ k \ \text{res} \ (1 + \ \text{max} \ \text{arg} \ (H \ r)) \\
\quad | \text{Kright} \ (k, \ hl) \rightarrow \text{post} \ k \ \text{res} \ (1 + \ \text{max} \ hl \ \text{arg}) \ ) \ *
\]

let rec height t k = match t with
  ...
(*@ r = height t k
  ensures \text{post} k \ r \ (H \ t) \ ) *
and apply k arg = match k with
  ...
(*@ r = apply k arg
  ensures \text{post} k \ arg \ r \ )
What About Effects?

\[ \text{pre } f : \tau_1 \rightarrow \text{state} \rightarrow \text{prop} \]

\[ \text{post } f : \tau_1 \rightarrow \text{state} \rightarrow \text{state} \rightarrow \tau_2 \rightarrow \text{prop} \]

[Kanig, 2011]
let rec distinct_elts_loop (t: int tree) (k: unit -> unit) =
  match t with
  | Empty -> k ()
  | Node (l, x, r) ->
    h := S.add x !h;
    distinct_elts_loop l (fun () ->
      (@@ ensures post k () (set_of_tree r (old !h)) !h () *)
    distinct_elts_loop r (fun () ->
      (@@ ensures post k () (old !h) !h () *)
    k ())))
  (@@ r = distinct_elts_loop t k
   ensures post k () (set_of_tree t (old !h)) !h () *
let n_distinct_elts t =
    let h := S.empty () in
    let rec distinct_elts_loop t k ... in
    distinct_elts_loop t
    (fun x -> (*@ ensures !h = (old !h) *) x);
    S.cardinal !h
(*@ r = n_distinct_elts t
    ensures r = Set.cardinal (set_of_tree t Set.empty) *)

Automatically proved after defunctionalization.
The Good, the Bad, and the Ugly
<table>
<thead>
<tr>
<th>Case Study</th>
<th>Lines of Code</th>
<th>Lines of Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicative Queue</td>
<td>25</td>
<td>17</td>
</tr>
<tr>
<td>Binary Search</td>
<td>62</td>
<td>40</td>
</tr>
<tr>
<td>CNF Conversion</td>
<td>113</td>
<td>47</td>
</tr>
<tr>
<td>Ephemeral Queue</td>
<td>40</td>
<td>29</td>
</tr>
<tr>
<td>Fast Exponentiation</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>13</td>
<td>34</td>
</tr>
<tr>
<td>Leftist Heap</td>
<td>99</td>
<td>178</td>
</tr>
<tr>
<td>Mjrtty</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>OCaml List.fold_left</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>OCaml Stack</td>
<td>25</td>
<td>27</td>
</tr>
<tr>
<td>Pairing Heap</td>
<td>65</td>
<td>101</td>
</tr>
<tr>
<td>Same Fringe</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>Small-step Iterators</td>
<td>42</td>
<td>52</td>
</tr>
<tr>
<td>OCaml Set</td>
<td>122</td>
<td>117</td>
</tr>
<tr>
<td>Union Find</td>
<td>36</td>
<td>29</td>
</tr>
<tr>
<td>Arithmetic Compiler</td>
<td>235</td>
<td>44</td>
</tr>
</tbody>
</table>
type 'a cell =
    | Nil
    | Cons of { content: 'a; mutable next: 'a cell }

In Why3:

Error: This field has non-pure type, it cannot be used in a recursive type definition

In Viper:

field content: Int
field next: Ref

predicate Queue (this: Ref) {
    this != null ==> 
    acc(this.content, 1/2) && acc(this.next) && Queue(this.next) }
The problem of mixing languages:

```ocaml
type 'a t = {
    ... 
    mutable view : 'a list [@ghost];
}

let pop q =
    ... 
    q.view <- tail_list q.view
```
Gospel as a proof language:

```ocaml
type 'a t = {  
  ...
  (@ mutable model view : 'a seq *)
}

let pop q =  
  ...
  (@ q.view <- q.view[1 ..] *)
```
Proof of OCamlGraph modules:

```ocaml
module Check (G: sig
  type t
  module V : Sig.COMPARABLE
  val iter_succ : (V.t -> unit) -> t -> V.t -> unit
end) = struct

let check_path pc v1 v2 =
  ...
  let q = Queue.create () in
  ...
  G.iter_succ (fun v' -> Queue.add v' q) pc.graph v
```
Proof of OCamlGraph modules:

```ocaml
module Check (G: sig
    type t
    module V : Sig.COMPARABLE
    val succ : t - > V.t -> V.t list
end) = struct

let check_path pc v1 v2 =
...
    let q = Queue.create () in
    ...
    let sucs = G.succ pc.graph v in
    let rec iter_succ = function
        | [] -> ()
        | v' :: r -> Queue.add v' q; iter_succ r in
    iter_succ sucs
```

39
Attach Gospel specification to the higher-order iterator...

\[
G.\text{iter}\_\text{succ} (\text{fun } v' \rightarrow (\text{@ invariant } I \text{ }) \text{Queue.add } v' \text{ q}) \text{ pc.graph } v
\]

...and translate it to a first-order counterpart

\[
\text{for } v' \text{ in pc.graph, } v \text{ with Iter\_succ do}
\]

\[
\text{invariant } \{ I \}
\]

\[
\text{Queue.add } v' \text{ q}
\]

\[
\text{done}
\]

Question: are we doing an equivalence proof here?

- there is an equivalent clause in Gospel
Conclusion
Conclusion & Future Work

The road so far:

• a deductive verification tool for a subset of OCaml
• several (semi-)automatically verified case studies
• use of defunctionalization to verify higher-order programs

The road ahead:

• specify and verify more and larger case studies
• formalization of the defunctionalization approach
• heap-manipulating programs
• higher-order effectful programs: CFML [Chargueraud et al.]
• a more general analysis framework for OCaml:
  - complexity checking [Gueneau et al., ESOP’18]
  - information flow [Pottier & Simonet, TOPLAS’03]
  - model checking [Kobayashi et al., PLDI’11 & ESOP’17]
  - runtime assertion checking [Filliâtre & Pascutto, RV’21]
Cameleer
a Deductive Verification Tool for OCaml

Mário Pereira and António Ravara,
“Cameleer: a Deductive Verification Tool for OCaml”,

Mário Pereira
“Deductive Verification of OCaml Programs in Cameleer”,
International Conference on Functional Programming 2021. (tutorial)

https://github.com/ocaml-gospel/cameleer