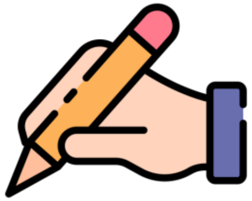
Implementing OCaml APIs in Coq

Joshua M. Cohen

Princeton University

CoqPL 2025

```
Fixpoint app {A: Type} (l1 l2: list A) : list A :=
    match l1 with
    | [] => []
    | x :: t => x :: (app t l2)
    end.
```



```
Fixpoint app {A: Type} (l1 l2: list A) : list A :=
    match l1 with
    | [] => []
    | x :: t => x :: (app t l2)
    end.
```

Extraction

```
(** val app : 'a1 list -> 'a1 list -> 'a1 list **)
let rec app l1 l2 = match l1 with
| [] -> []
| x :: t -> x :: (app t l2)
```

```
Fixpoint app {A: Type} (l1 l2: list A) : list A :=
    match l1 with
    | [] => []
    | x :: t => x :: (app t l2)
    end.
```

Extraction

```
(** val app : 'a1 list -> 'a1 list -> 'a1 list **)
let rec app l1 l2 = match l1 with
| [] -> []
| x :: t -> x :: (app t l2)
```

```
...
List.iter print_int (app l1 l2)
...
```

```
Fixpoint app {A: Type} (l1 l2: list A) : list A :=
    match l1 with
    | [] => []
    | x :: t => x :: (app t l2)
    end.
```

Extraction

```
(** val app : 'a1 list -> 'a1 list -> 'a1 list **)
let rec app l1 l2 = match l1 with
| [] -> []
| x :: t -> x :: (app t l2)
```

```
...
List.iter print_int (app l1 l2)
...
```

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
   let rec append l1 l2 = ...
```

list.ml

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

Stdlib

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
let rec append l1 l2 = ...
```

list.ml

Core

```
val empty :
('a, 'cmp) comparator ->
('a, 'b, 'cmp) t
```

map.mli

```
let empty c = ...
```

map.ml

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

Stdlib

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
   let rec append l1 l2 = ...
```

list.ml

```
val empty :
('a, 'cmp) comparator ->
('a, 'b, 'cmp) t
```

map.mli

```
   let empty c = ...
```

map.ml

Core

```
val spanningtree :
  G.t -> G.E.t list
```

kruskal.mli

```
   let spanningtree g = ...
```

kruskal.ml

ocamlgraph

Stdlib

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
let rec append l1 l2 = ...
```

list.ml

```
val empty :
('a, 'cmp) comparator ->
('a, 'b, 'cmp) t
```

map.mli

```
let empty c = ...
```

map.ml

Core

```
val spanningtree :
  G.t -> G.E.t list
```

kruskal.mli

```
let spanningtree g = ...
```

kruskal.ml

ocamlgraph

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

Software Analyzers

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let m = Core.Map.empty (module
String)
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

Stdlib

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
  let rec append l1 l2 = ...
```

list.ml

```
val empty :
('a, 'cmp) comparator ->
('a, 'b, 'cmp) t
```

map.mli

```
  let empty c = ...
```

map.ml

Core

```
val spanningtree :
  G.t -> G.E.t list
```

kruskal.mli

```
  let spanningtree g = ...
```

kruskal.ml

ocamlgraph

```
...
let l3 = Stdlib.List.append l1 l2
...
...
...
...
let
Str
...
...
...
...
module P1 = Kruskal.Make(G)(W)
let x = P1.spanningtree g
...
```

foo.ml

Enables *incremental* verification

But, must match *exact* interface

Stdlib

```
val append :
  'a list -> 'a list -> 'a list
```

list.mli

```
  let rec append l1 l2 = ...
```

list.ml

```
val empty :
('a, 'cmp) comparator ->
('a, 'b, 'cmp) t
```

map.mli

Core

```
  let empty c = ...
```

map.ml

```
val spanningtree :
  G.t -> G.E.t list
```

kruskal.mli

```
  let spanningtree g = ...
```

kruskal.ml

ocamlgraph

Software Analyzers

Problem: Coq (Gallina) != OCaml

Problem: Coq (Gallina) != OCaml

```
val create_param_decl : lsymbol -> decl

let create_param_decl ls =
  if ls.ls_constr <> 0 || ls.ls_proj then
    raise (UnexpectedProjOrConstr ls);
  let news = Sid.singleton ls.ls_name in
  mk_decl (Dparam ls) news
```

Problem: Coq (Gallina) != OCaml

```
val create_param_decl : lsymbol -> decl

let create_param_decl ls =
  if ls.ls_constr <> 0 || ls.ls_proj then
    raise (UnexpectedProjOrConstr ls);
  let news = Sid.singleton ls.ls_name in
  mk_decl (Dparam ls) news
```

Throws an exception

Problem: Coq (Gallina) != OCaml

```
val create_param_decl : lsymbol -> decl

let create_param_decl ls =
  if ls.ls_constr <> 0 || ls.ls_proj then
    raise (UnexpectedProjOrConstr ls);
  let news = Sid.singleton ls.ls_name in
  mk_decl (Dparam ls) news
```

Throws an exception

Stateful (hash-consing)

Problem: Coq (Gallina) != OCaml

```
val create_param_decl : lsymbol -> decl

let create_param_decl ls =
  if ls.ls_constr <> 0 || ls.ls_proj then
    raise (UnexpectedProjOrConstr ls);
  let news = Sid.singleton ls.ls_name in
  mk_decl (Dparam ls) news
```

Not reflected in type!

Throws an exception

Stateful (hash-consing)

OCaml Features Not (Idiomatically) Representable in Coq

- Exceptions
- Mutable State
- Opaque Types
- Reference Equality
- Machine-length integers(*)
- Mixed record-inductive types
- Etc

We want *computable* code in Coq and OCaml => No Axioms!

Our Solution

- A lightweight, pragmatic solution: represent features *differently in Coq and OCaml* by (carefully) modifying extraction

- We propose a *design principle* using this idea and provide a small library to enable programming with this pattern

Example: Exceptions

- Idea: implement error handling in Coq with error monad

Running example: hd

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Example: Exceptions

- Idea: implement error handling in Coq with error monad

Step 1: Model exceptions in Coq

```
Record errtype : Type := {errname : string; errargs : Type; errdata : errargs}.

Definition mk_errtype name {A} (x: A) :=
    {| errname := name; errargs := A; errdata := x |}.

Definition Failure (msg: string) : errtype := mk_errtype "Failure" msg.
```

Example: Exceptions

- Idea: implement error handling in Coq with error monad

Step 2: Define monadic interface (with coq-ext-lib)

```
Definition errorM A : Type := Datatypes.sum errtype A.

Definition err_ret {A} (x: A) : errorM A := ret x.

Definition err_bnd {A B} (f: A -> errorM B) (x: errorM A) : errorM B := bind x f.

Definition throw {A} (e: errtype) : errorM A := raise e.
```

Example: Exceptions

- Idea: implement error handling in Coq with error monad

Step 3: Erase monadic interface when extracting

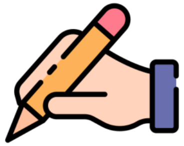
```
Extract Constant errorM "'a" => "'a".

Extract Inductive errtype => exn [""].

Extract Inlined Constant err_ret => "(fun x -> x)".

Extract Inlined Constant err_bnd => "(@@)".

Extract Inlined Constant Failure => "Failure".
```

Example: Exceptions

- Idea: implement error handling in Coq with error monad

Step 4: Implement API in Coq using interface

```
Definition hd {A: Type} (l: list A) : errorM A :=
    match l with
    | [] => throw (Failure "hd")
    | x :: _ => err_ret x
    end.
```



```
let hd = function
| [] -> raise (Failure "hd")
| x :: _ -> x
```

Recipe

Given OCaml feature not representable in Coq:

- Identify abstract interface and model in Coq
- Implement primitives in Coq and map to OCaml
- Coq clients *only* use primitives to interact with feature

Result:

1. Coq code is computable, provable, and axiom-free
2. OCaml code is computable and has interface client expects (no monads)

Mutable State

- Implement state in Coq with state monad, in OCaml with mutable reference
  - `st A B` extracted to B
  - `bind` and `ret` same as error
  - Provide type for mutable references, extract to `'a ref`
  - `get` looks up in reference (!)
  - `set` sets value of reference (:=)
- Provide generic State module for creating state/reference of any type

Mutable State Soundness

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Unsound!

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Unsound!

Solution: fix initial value!

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Unsound!

Solution: fix initial value!

```
Module Type State (T: ModTy).
Parameter create : unit -> st T.t unit.
Parameter get : unit -> st T.t T.t.
Parameter set : T.t -> st T.t unit.
Parameter runState : forall {A: Type}, st T.t A -> A.
End State.
```

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Unsound!

Solution: fix initial value!

Module storing type and default element

```
Module Type State (T: ModTy).
Parameter create : unit -> st T.t unit.
Parameter get : unit -> st T.t T.t.
Parameter set : T.t -> st T.t unit.
Parameter runState : forall {A: Type}, st T.t A -> A.
End State.
```

Mutable State Soundness

```
Definition st A B := A -> A * B.
Definition runState (s: st A B) (x: A) : B := snd (s x).
```

```
val runState : 'b -> 'a -> 'b
let runState x y = x
```

Unsound!

Solution: fix initial value!

Module storing type and default element

```
Module Type State (T: ModTy).
Parameter create : unit -> st T.t unit.
Parameter get : unit -> st T.t T.t.
Parameter set : T.t -> st T.t unit.
Parameter runState : forall {A: Type}, st T.t A -> A.
End State.
```

Run state starting from fixed value, OCaml resets mutable reference

Example: Integer Term API

```
type var
type tm_bound
type tm = private ... | Tvar of var | Tlet of tm * tm_bound

val create_var : string -> var
val t_open_bound: tm_bound -> (var * tm)
val t_close_bound: var -> tm -> tm_bound
val sub_t: var -> tm -> tm -> tm
```

Proving Things about State

- Variant of Hoare State Monad [TPHOLs '09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub_t v t tm1)
    (fun _ t2 _ => forall vv,
      tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

Proving Things about State

- Variant of Hoare State Monad [TPHOLs '09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

If input well-formed (variables <= current state)

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub_t v t tm1)
    (fun _ t2 _ => forall vv,
      tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

Proving Things about State

- Variant of Hoare State Monad [TPHOLs '09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub_t v t tm1)
    (fun _ t2 _ => forall vv,
      tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

Proving Things about State

- Variant of Hoare State Monad [TPHOLs ‘09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub_t v t tm1)
    (fun _ t2 _ => forall vv,
      tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

After running stateful function sub_t

Proving Things about State

- Variant of Hoare State Monad [TPHOLs '09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub_t v t tm1)
    (fun _ t2 _ => forall vv,
      tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

Proving Things about State

- Variant of Hoare State Monad [TPHOLs ‘09] (shallow embedding)

```
Definition st_spec {A B: Type} (Pre: A -> Prop) (s: st A B)
  (Post: A -> B -> A -> Prop) : Prop :=
  forall i, Pre i -> Post i (fst (runState s i)) (snd (runState s i)).
```

- Use this to prove substitution correct

```
Theorem sub_t_rep (tm1: tm):
  st_spec (fun i => tm_st_wf i t /\ tm_st_wf i tm1 /\ var_st_wf i v)
    (sub t v t tm1)
    (fun _ t2 _ => forall vv,
     tm_rep vv t2 = tm_rep (add_val v (tm_rep vv t) vv) tm1).
```

The result coincides with semantic substitution

Limitations

- Difficult to scale, need e.g. new definition/extraction directive for each exception
- Need dune preprocessing, not difficult but repetitive
- Need to redefine monadic operations for each monad to avoid Object.magic in OCaml code
  - Could be solved with a hypothetical `Extraction cbv foo` command
- Enlarge TCB with extraction directives
- Big limitation: cannot enforce that Coq client only uses interface!

Related + Future Work

- Ynot [ICFP ’08] - framework for imperative programming, axiomatizes stateful operations
- ITrees [POPL ‘20] – model impure code with coinductive monads, cannot compute in Coq
- Proof-certificate-producing stateful CakeML from monadic HOL [IJCAR ‘18]
    - Could be used to certify correctness (with an OCaml program logic)
- FVDP [Boulmé Thesis ‘21] – use may-return monad for untrusted OCaml oracle
    - Could allow generated OCaml APIs to depend on other OCaml code
- VeriFFI [POPL ‘25] – verified FFI between Coq and C
    - Solves opacity issue by axiomatizing foreign functions, giving rewrite rules

Conclusion

- Our approach aims to be *lightweight* and *practical*
- Resulting programs:
  - Executable in both OCaml and Coq
  - Usable by clients in both languages
  - Can be reasoned about in Coq using ordinary Coq logic
- Code available at https://github.com/joscoh/coq-ocaml-api
- Includes examples: List API, mutable counter, term API
- Thanks for listening!

References

- Abrahamsson, O., Ho, S., Kanabar, H. et al. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. J Autom Reasoning 64, 1287–1306 (2020). https://doi.org/10.1007/s10817-020-09559-8
- Sylvain Boulmé. Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles). Software Engineering [cs.SE]. Université Grenoble-Alpes, 2021.
- Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. Proc. ACM Program. Lang. 9, POPL, Article 24 (January 2025), 31 pages. https://doi.org/10.1145/3704860
- Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP '08). Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1411204.1411237
- Swierstra, W. (2009). A Hoare Logic for the State Monad. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds) Theorem Proving in Higher Order Logics. TPHOLs 2009. Lecture Notes in Computer Science, vol 5674. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03359-9_30
- Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. Proc. ACM Program. Lang. 4, POPL, Article 51 (January 2020), 32 pages. https://doi.org/10.1145/3371119