Osiris: an Iris-based program logic for OCaml.

Arnaud Daby-Seesaram (ENS Paris-Saclay, France)
François Pottier (Inria, Paris, France)
Armaël Guéneau (Inria, Laboratoire Méthodes Formelles, France)

8 September 2023

General Context.

Context

- Some verification tools are based on:
  - automatic solvers,
  - (manual) deductive reasoning about programs.
- Coq is a proof assistant ;
- Iris is a Coq framework for separation logic and program verification.

General Context.

Context

- Some verification tools are based on:
  - automatic solvers,
  - (manual) deductive reasoning about programs.
- Coq is a proof assistant ;
- Iris is a Coq framework for separation logic and program verification.

Why choose Iris ?

Builtin proof techniques to help program verification. Iris handles:

- divergent programs,
- programs manipulating a heap,
- programs with higher order functions,
- ...

Osiris allows users to use most Iris features.

Program Verification

Program specification.

- Pre-condition: condition under which the program is proven safe ;
- Post-condition: provides information on the result of a computation.

Specification of `length`:

$$\{v \ \texttt{represents the list l}\}$$
$$\texttt{call length } v$$
$$\{\lambda res.\ \ulcorner res = \texttt{length of the list l}\urcorner\}$$

Program Verification

Program specification.

- Pre-condition: condition under which the program is proven safe ;
- Post-condition: provides information on the result of a computation.

Specification of `length`:

$$\{v \texttt{ represents the list l}\}$$
$$\texttt{call length } v$$
$$\{\lambda res.\ulcorner res = \texttt{length of the list l}\urcorner\}$$

To verify a program should ensure:

- its safety $\Rightarrow$ no crash,
- its progress $\Rightarrow$ it is not stuck,
- the respect of its post-condition ϕ.

Previous Work and contributions.

Previous Work

- CFML2 allows interactive proofs of OCaml programs in Coq.
- Iris has been instantiated with small ML-like languages,
- Other projects have used Iris to reason about specific aspects of OCaml:

Project	Aspect of the language
Cosmo	Multicore OCaml and weak-memory
iris-time-proofs	Time complexity in presence of `lazy`
Hazel	Effect Handlers
Space-Lambda	Garbage Collection

Our contributions.

- a proof methodology to prove OCaml programs,
- an original semantics for OCaml,
- a program logic using Iris.

In this talk

1. Proof methodology: how to verify an OCaml program?
2. Structure of Osiris:
   - an original semantics for OCaml,
   - a program logic built on Iris $\rightarrow$ Coq tactics.

Osiris is still a prototype at the moment.

Proof Methodology

Methodology:

- translate OCaml files into Coq files,
- write specifications of the files (seen as modules) and their functions,
- prove these specifications.

Translation tool.

Translation process:

1. retrieve the Typed-Tree of the OCaml file to translate (using `compilerlibs`),

```
(* Content of [file.ml] *)
let cst = 10
```

Translation tool.

Translation process:

1. retrieve the Typed-Tree of the OCaml file to translate (using `compilerlibs`),

```
(* Content of [file.ml] *)
let cst = 10
```

2. translate the Typed-Tree into an Osiris AST,

```
MkStruct [ ILet (Binding1 (PVar "cst") (EInt 10)) ]
```

Translation tool.

Translation process:

1. retrieve the Typed-Tree of the OCaml file to translate (using `compilerlibs`),

```
(* Content of [file.ml] *)
let cst = 10
```

2. translate the Typed-Tree into an Osiris AST,

```
MkStruct [ ILet (Binding1 (PVar "cst") (EInt 10)) ]
```

3. print the module-expression into a Coq file.

```
Definition _File : mexpr :=
  MkStruct [ ILet (Binding1 (PVar "cst") (EInt 10)) ].
```

Example: a toy module. (I)

```
module Toy = struct
  let rec length l =
    match l with
    | []  → 0
    | _ :: l → 1 + length l

  let lily = [1; 2; 3; 4]

  let len = length lily
end
```

Example: a toy module. (II)

```
module Toy = struct
  let rec length l =
    match l with
    | []  → 0
    | _ :: l → 1 + length l

  let lily = [1; 2; 3; 4]

  let len = length lily
end
```

Specification of the module:

- it contains a function `length`;
- the function `length` satisfies the aforementioned specification.

Example: a toy module. (II)

```
module Toy = struct
  let rec length l =
    match l with
    | []  → 0
    | _ :: l → 1 + length l

  let lily = [1; 2; 3; 4]

  let len = length lily
end
```

Specification of the module:

- it contains a function `length`;
- the function `length` satisfies the aforementioned specification.

Verification of a module.

- evaluate the module-expression,

  ↪ The evaluation contains breakpoints, *e.g.* at:

  - function calls,
  - let-bindings.

- use tactics to make progress if need be.

  ↪ *e.g.* heap manipulations, non-deterministic constructs of the semantics.

Example: Proof script.

```
module Toy = struct
  let rec length l =
    match l with
    | []  → 0
    | _ ::  l → 1 + length l

  let lily = [1; 2; 3; 4]
  let len = length l
end
```

```
wp. (* ← starts the evaluation of [Toy]. *)

(* The evaluation stops after the body of [length]. *)
oSpecify "length"   (* I want to prove that [length] *)
        spec_length (* satisfies [spec_length]. *)
        "#Hlen"!.   (* Please remember this fact as "Hlen". *)
{ (* Omitted. *) }

(* The evaluation starts again...
   and stops after the evaluation of [1; 2; 3; 4]. *)
wp_continue. (* Nothing to do here. *)

(* The evaluation starts once more...
   and stops on the function call [length lily] *)
wp_use "Hlen". (* Use "Hlen". *)
(* Omitted : introduction of the result. *)

(* [len] is about to be added to the environment
   ⇒ this is a breakpoint for the evaluation. *)
wp_continue. (* Nothing to do here. *)

(* Osiris has all the ingredients and can finish the proof. *)
oModuleDone.
```

Description of the tool.

Goal

Prove programs using Coq tactics.

Steps

1. Give meaning to the syntax,
   $\hookrightarrow$ define an operational semantics for OCaml.
2. Define reasoning rules to reason about this semantics,
   $\hookrightarrow$ these rules are proven once and for all.
3. Define Coq tactics to exploit these rules.
   $\hookrightarrow$ the tactics rely on aforementioned rules $\Rightarrow$ they are correct by construction.

Motivation for an ample-step semantics.

Most Iris projects use a small-step semantics.

Small-step semantics $\longrightarrow$ Iris-provided program logic

This is appealing... but OCaml is a large language.

Motivation for an ample-step semantics.

Most Iris projects use a small-step semantics.

Small-step semantics $\longrightarrow$ Iris-provided program logic

This is appealing... but OCaml is a large language.

A small-step semantics for OCaml semantics is large.

Number of transitions due to the many constructions of the language.
$\hookrightarrow$ *e.g.* pattern-matching, ADTs, records, modules.

Non-Determinism the order of evaluation of expressions is not defined, and some expressions can be erased ;
$\hookrightarrow$ *e.g.* function calls, tuples, dynamic checks.

Solution.

A semantics in two steps, each tackling one of these issues.

Ample-step semantics.

Definition: Ample-step semantics

1. Evaluate OCaml expressions in a smaller language `micro A` ;

```
Fixpoint eval : env → expr → micro val.
Definition call : val → val → micro val.
```

   `micro A` describes generic computations of type `A`.

2. Provide a small-step semantics to `micro A`.

```
Inductive step : store * micro A → store * micro A → Prop.
```

Definition of `micro A`.

```
Inductive micro A :=
| Ret (a : A)
| Crash
| Next
| Par {A1 A2} (m1 : micro A1) (m2 : micro A2)
    (k : A1 * A2 → micro A)
    (ko : unit → micro A)
| Stop {X Y} (c : code X Y) (x : X)
    (k : Y → micro A)
    (ko : unit → micro A).
```

(a) Computations of type `A`.

```
Inductive code : Type → Type → Type :=
(* code X Y : Type of a system call.
   X : type of the parameter of the syst. call,
   Y : type of the returned value. *)
(* Provides:
   - Non-deterministic binary choice ;
   - heap manipulation ;
   - potential divergence. *)
```

(b) System calls, implementing OCaml features.

Figure: Definition of `micro A`.

Par is used to model non-determinism, *not* parallelism.

Example

```
(* Evaluation of a function call. *)
eval η (EApp e1 e2) =
  Par (eval η e1)
      (eval η e2)
      (λ '(v1, v2), call v1 v2)
      (λ _, Next)
```

Proofs of programs.

To prove an expression e

is to prove

$$\texttt{after}\ (\texttt{eval}\,\eta\,e)\ \{\phi\}$$

- $\texttt{eval}\,\eta\,e$: $\texttt{micro val}$,
- `after` ensures safety, *etc.*

Proofs of programs.

To prove an expression e

is to prove

$$\texttt{after}\ (\texttt{eval}\ \eta\ e)\ \{\phi\}$$

- $\texttt{eval}\ \eta\ e\ : \texttt{micro val}$,
- `after` ensures safety, *etc.*

A Selection of reasoning rules

$$\textsc{Ret}\ \frac{\phi\,(a)}{\texttt{after}\ (\texttt{Ret}\,(a))\ \{\phi\}} \qquad \textsc{Par}\ \frac{\begin{array}{c}\texttt{after}\ (m_1)\ \{\phi_1\} \qquad \texttt{after}\ (m_2)\ \{\phi_2\} \\ \forall v_1\ v_2.\phi_1\,(v_1) -\!\!* \phi_2\,(v_1) -\!\!* \texttt{after}\ (k\,(v_1,\ v_2))\ \{\phi\}\end{array}}{\texttt{after}\ (\texttt{Par}\,(m_1,\ m_2,\ k,\ ko))\ \{\phi\}}$$

$$\textsc{alloc}\ \frac{\triangleright(\forall \ell.\ell \mapsto v -\!\!* \texttt{after}\ (k\,(\ell))\ \{\phi\})}{\texttt{after}\ (\texttt{Stop}\,(\texttt{CAlloc},\ v,\ k,\ ko))\ \{\phi\}}$$

An alternative Program Logic for pure programs.

Définition : simp

$$\texttt{simp}\ m_1\ m_2 \triangleq \text{«The computation } m_1 \text{ can be simplified into } m_2.\text{»}$$

after and simp

$$\textsc{Simp}\ \frac{\texttt{simp}\ m_1\ m_2 \qquad \texttt{after}\ (m_2)\ \{\phi\}}{\texttt{after}\ (m_1)\ \{\phi\}}$$

Two uses of simp:

- Program specification: Let f be an OCaml function represented by the Gallina function f and a be represented by a.
$$\texttt{simp (call f a) (Ret } (f\ a)\texttt{)}$$
- Program simplification: simp (eval η $\underbrace{1+2+3+4+5}_{\texttt{8 function calls}}$) (Ret 15).

Short- and long-term goals for Osiris.

Short-term goal

To add support for more OCaml constructs and features.

(Very) long-term goal

Osiris might some day incorporate previous work:
Hazel, *Cosmo*, *iris-time-proofs* or *Space-Lambda*.

We are far from this!
There is still a lot of work to be done before we can even begin to think about it.

Conclusion

Osiris currently supports:

- modules and sub-modules,
- immutable records,
- function calls,
- recursive functions,
- `for`-loops,
- manipulation of references,
- ADTs and pattern-matching.

↪ Note: we need more tests about these constructs.

Future work

We have yet to understand how:

- pure modules and functions should be specified and used;
- to specify modules;

  ↪ we have used two styles of specifications, but neither is fully satisfying yet.
- to describe dependencies;
- ...

↪ There is still work to do to make the tool more ergonomic, and some uncertainties *wrt.* some semantic choices.

▸ Main menu

Separation Logic and Iris.

- ▸ Separation Logic
- ▸ Iris

▸ Main menu

A few words on Separation Logic.

In Separation Logic. . .

- Notion of resources, describing various logical information.
- Propositions are called «*assertions*».
- An assertion holds *iff* resources at hand satisfy it. *e.g.*

$$W^i \triangleq \text{«ownership of } i \text{ tons of wood.»}$$

Two additional operators:

- Separating conjunction ($*$) :

$$W^{40} \vdash W^{30} * W^{10}$$

- Magic Wand ($-\!*$) :

$$W^{27} \vdash W^{3} -\!* W^{30}$$

A few words on Iris.

Iris is a framework for Separation Logic. It is written, proven and usable in Coq.

Iris' logic is modal and step-indexed

- Persistence modality $\square P$: $\square P \vdash \square P * P$.
- *later* modality $\triangleright P$: P will hold at the next logical step.
- Fancy-Update modality ${}_{\mathcal{E}_1}{\Rrightarrow}_{\mathcal{E}_2} P$: P and invariants whose name appear in $\mathcal{E}_2$ hold, under the assumption that all invariants whose name occurs in $\mathcal{E}_1$ hold.
- Basic-Update modality $\dot{\Rrightarrow} P$: allows to update the ghost state before proving P.

Proof techniques provided by Iris

resources: Users can define their own resources ;

invariants: $\boxed{P}^{\mathcal{N}}$ is a logical black box containing P. The name $\mathcal{N}$ is associated with the box ;

induction de Löb: $(\square (\triangleright P \twoheadrightarrow P)) \twoheadrightarrow P$.

Weakest Precondition.

- ▸ Highly simplified, simplified and exact definition of `after`
- ▸ Adequacy theorem

▸ Main menu

Definition of $\texttt{after}$.

Very simplified version: no heap, no invariant.

Weakest Precondition

- If $\exists v. m = \texttt{Ret}\,(v)$, then
$$\texttt{after}\;(m)\;\{\Phi\} \triangleq \Phi\,(v)$$
- Otherwise
$$\begin{aligned}\texttt{after}\;(m)\;\{\Phi\} \triangleq\ & \\ & \ulcorner \exists m'.\ m \rightsquigarrow m' \urcorner * \\ & \quad \forall m'.\ \ulcorner m \rightsquigarrow m' \urcorner \mathrel{-\!\!*} \\ & \qquad \triangleright\, \texttt{after}\;\left(m'\right)\;\{\Phi\}\end{aligned}$$

▸ Return

▸ Main menu

Definition of `after`.

Simplified version: there is a heap, but still no invariants.

Logical Heap

For any physical heap σ, $\mathcal{S}(\sigma)$ is an assertion describing the heap. It is provided by Iris.

Weakest Precondition

- If $\exists v. m = \texttt{Ret}(v)$, then

$$\texttt{after}\ (m)\ \{\Phi\} \triangleq \forall \sigma.\ \mathcal{S}(\sigma) \twoheadrightarrow \mathcal{S}(\sigma) * \Phi(v)$$

- Otherwise

$$\begin{aligned}\texttt{after}\ (m)\ \{\Phi\} \triangleq\ & \forall \sigma.\ \mathcal{S}(\sigma) \twoheadrightarrow \\ & \ulcorner \exists \sigma', m'.\ (\sigma, m) \leadsto (\sigma', m') \urcorner * \\ & \forall \sigma', m'.\ \ulcorner (\sigma, m) \leadsto (\sigma', m') \urcorner \twoheadrightarrow \\ & \triangleright \mathcal{S}(\sigma') * \texttt{after}\ (m')\ \{\Phi\}\end{aligned}$$

Definition of `after`.

Real definition of `after`.

Logical Heap

For any physical heap σ, $\mathcal{S}(\sigma)$ is an assertion describing the heap. It is provided by Iris.

Weakest Precondition

- If $\exists v. m = \mathtt{Ret}(v)$, then

$$\mathtt{after}_{\mathcal{E}}\ (m)\ \{\Phi\} \triangleq \forall \sigma.\ \mathcal{S}(\sigma) \twoheadrightarrow {}_{\mathcal{E}}{\Rrightarrow}_{\emptyset}\ {}_{\emptyset}{\Rrightarrow}_{\mathcal{E}} \mathcal{S}(\sigma) * \Phi(v)$$

- Otherwise

$$\begin{aligned}\mathtt{after}_{\mathcal{E}}\ (m)\ \{\Phi\} \triangleq\ & \forall \sigma.\ \mathcal{S}(\sigma) \twoheadrightarrow \\ & {}_{\mathcal{E}}{\Rrightarrow}_{\emptyset} \ulcorner \exists \sigma', m'.\ (\sigma, m) \rightsquigarrow (\sigma', m') \urcorner * \\ & \forall \sigma', m'.\ \ulcorner (\sigma, m) \rightsquigarrow (\sigma', m') \urcorner \twoheadrightarrow \\ & {}_{\emptyset}{\Rrightarrow}_{\emptyset} \triangleright {}_{\emptyset}{\Rrightarrow}_{\emptyset}\ {}_{\emptyset}{\Rrightarrow}_{\mathcal{E}} \mathcal{S}(\sigma') * \mathtt{after}_{\mathcal{E}}\ (m')\ \{\Phi\}\end{aligned}$$

Return

Main menu

Adequacy theorem for `after`.

Adequacy theorem

Let `A` be a type, m_1 and m_n terms of type `micro A`, σ_n a heap, n a natural integer, and ψ a pure proposition.
If the configuration $(\emptyset,\ m_1)$ reduces in n steps to $(\sigma_n,\ m_n)$, and if the following assertion holds:

$$\vdash {}_{\top}{\Rrightarrow}_{\top} \exists (\Phi : \mathtt{A} \to \mathit{iProp}\ \Sigma)\,.\mathtt{after}_{\top}\ (m_1)\ \{\Phi\} * \Big(\mathtt{after}_{\top}\ (\mathcal{S}\,(\sigma_{\top}) * m_{\top})\ \{\phi\} \mathrel{-\!\!*} {}_{\top}{\Rrightarrow}_{\emptyset} \ulcorner \psi \urcorner\Big)$$

then ψ is true.

Corollary : Progress and respect of the post-condition.

Let `A` be a type, m_1 and m_n terms of type `micro A`, σ_n a heap, n a natural integer and ψ a pure post-condition (*i.e.* of type $\mathtt{A} \to \mathtt{Prop}$).
If $(\emptyset,\ m_1)$ reduces to $(\sigma_n,\ m_n)$ in n steps, and that the following assertion holds:

$$\vdash \forall (\text{ hypothesis granted access to resources})\,.\mathtt{after}_{\top}\ (m_1)\ \{\lambda v.\ \ulcorner \psi\,(v) \urcorner\}$$

then the configuration $(\sigma_n,\ m_n)$ is not stuck, *i.e.* either m_n is a value, or $(\sigma_n,\ m_n)$ can step. Moreover, if m_n is a value v, then $\psi\,(v)$ holds.

Examples: programs verifies with Orisis.

- Counter
- Records

Main menu

Monotone counters.

- Code
- Specifications
- Proof
- Use-Case

Return

Main menu

Counters : code

```
module Counter = struct
  let make () = ref 0
  let incr c = c := !c + 1
  let set c v = assert (!c <= v) ;
                c := v
  let get c = !c
end
```

Return
Main menu

Counters (uc) : code

```
open Counters
let do2 (f : 'a → 'b) (a : 'a) : 'b * 'b = (f a, f a)
let count_for n =
  let c, c' = do2 Counter.make () in (* !c = !c' = 0 *)
  Counter.set c' n ;
  for i = 1 to n do
  Counter.incr c;
  Counter.set c' (n + i) (* [c] stores i and [c'] stores (n + i). *)
  done;

  (* As [c] stores [n] and [c'] stores [n+n] after the for-loop, the difference
  is [n]. *)
  assert (Counter.get c' − Counter.get c = n) ;

  (* Return [n] *)
  Counter.get c

let count_rec n =
let c = Counter.make () in
  let rec aux i =
    let () = assert (0 <= i) in
    match i with
    | 0 → Counter.get c
    | _ → Counter.incr c; aux (i − 1)
  in aux n

let () = assert (2 = count_for 2)
let () = assert (2 = count_rec 2)
```

▸ Return

▸ Main menu

Counters : Specification. I

```
Definition is_counter (n : nat) (v : val) : iProp Σ:=
  ∃ (ℓ : loc), ⌜v = #ℓ⌝∗ℓ ↦#n.

Definition make_spec (vmake : val) : iProp Σ:=
  □WP call vmake #() {{ λres, is_counter 0 res }}.

Definition get_spec (vget : val) : iProp Σ:=
  □∀ (v : val) (n : nat),
  is_counter n v −∗ WP call vget v {{ λres, ⌜res = #n⌝ ∗is_counter n v }}.

Definition incr_spec (vincr : val) : iProp Σ:=
  □∀ (v : val) (n : nat),
  is_counter n v −∗
  WP call vincr v {{ λres, ⌜res = VUnit⌝ ∗is_counter (S n) v }}.

Definition set_spec (vset : val) : iProp Σ:=
  □∀ (v : val),
  WP call vset v {{
        λ res,
          ∀ (n m : nat),
          ⌜(n <= m)%nat⌝ →
          ⌜representable n⌝ →
          ⌜representable m⌝ →
          is_counter n v −∗
          WP call res #m {{ λres, ⌜res = VUnit⌝ ∗is_counter m v }} }}.
```

Counters : Specification. II

```
Definition Counter_specs : spec val :=
  SpecModule
    Auto
    [
      ("make", SpecImpure NoAuto make_spec) ;
      ("get", SpecImpure NoAuto get_spec) ;
      ("incr", SpecImpure NoAuto incr_spec) ;
      ("set", SpecImpure NoAuto set_spec)
    ]
    emp%I.

Definition Counter_spec : val →iProp Σ:=
  λ v, (□ satisfies_spec Counter_specs v)%I.

Definition File_spec (v : val) : iProp Σ:=
  □satisfies_spec
    (SpecModule Auto [("Counter", SpecImpure NoAuto Counter_spec)] emp%I) v.
```

Return
Main menu

Counters : proof

```
Lemma File_correct :
  ⊢ WP eval_mexpr η_Counters {{ File_spec }}.
Proof using Hη osirisGS0 Ση.
  oSpecify "make" make_spec vmake "#Hmake" !.
  { iIntros "!>".
    @oCall unfold; wp_bind; wp_continue.
    wp_alloc ℓ "[Hℓ _]".
    iExists ℓ.
    iSplit; first equality.
    by cbn. }

  oSpecify "incr" incr_spec vincr "#Hincr" !.
  { iIntros "!>" (? n) "(%ℓ&→ &Hℓ)".
    call. wp_load "Hℓ". wp_store "Hℓ".
    replace (VInt (repr (n + 1))) with (#(S n)); last first.
    { simpl. do 2 f_equal; lia. }
    prove_counter. }

  oSpecify "set" set_spec vset "#Hset" !.
  { (* ... *) }

  oSpecify "get" get_spec vget "#Hget" !.
  { iIntros "!>"(? nc) "(%ℓ&→ &Hℓ)".
    call. wp_load "Hℓ". prove_counter. }

  oSpecify "Counter" Counter_spec vCounter "#?" !.
  { iModIntro. wp_prove_spec. }

  iModIntro; wp_prove_spec.
Qed.
```

Records

- Code
- Specifications
- Proof

Records : code

```
type r = {
  i: int;
  b: bool;
}

let r_elt: r = {
  i = 10;
  b = true;
}

let flip r = { r with b = not r.b }

let lily = [ r_elt; flip r_elt ]

let r_val r =
  match r.b with
  | true -> r.i * 2 - 1
  | false -> r.i

let sum r1 r2 =
  r_val r1 + r_val r2
```

```
let rec is_odd_naive n =
  assert (n >= 0);
  if n > 1 then
    is_odd_naive (n-2)
  else begin
    if n = 0
      then false
      else true
    end

let is_odd n = n mod 2 = 0

type nat =
| O
| S of nat

let rec is_odd' = function
| O -> true
| S n -> not (is_odd' n)
```

Return

Main menu

Records : specifications I

```
(* (2) Definition of some values; useful to write the specs below. *)
Definition enc_r_elt : val := #{| b := true; i := 10 |}.
Definition enc_r_elt' : val := #{|b := false; i := 10|}.
Definition enc_lily : val := #[enc_r_elt; enc_r_elt'].

(* (3) Definition of specifications. *)
Definition is_equal (v res: val) : iProp Σ:= □⌜ res = v ⌝.

(* [flip] negates [b] in records of type [{ b: bool; i: int}]. *)
Definition flip_spec (v : val) : iProp Σ:=
  □∀ (b: bool) (i: Z), WP call v #{| b := b; i := i |} {{ λr, is_equal r #{| b := negb b; i := i |} }}.

(* [r_val_spec] performs a different arithmetic computation depending on the
   fiels [b] of a record. *)
Definition r_val_pure (r: R) : Z := (* ... *)
Definition r_val_spec (r_val: val): iProp Σ:=
  □∀ (r: R), WP call r_val #r {{ λresult, is_equal result #(r_val_pure r) }}.

Definition sum_pure (r1 r2: R) : Z := r_val_pure r1 + r_val_pure r2.
Definition sum_spec (vsum: val) : iProp Σ:=
  □∀ (r1 r2 : R),
  WP call vsum #r1 {{
        λ vpart,
        WP call vpart #r2 {{
              λ res,
              is_equal res #(sum_pure r1 r2) }} }}.
```

Return

Main menu

Records : specifications II

```
Fixpoint is_odd_pure (n: nat) : bool := (* ... *)
Definition is_odd_spec (vis_odd: val) : iProp Σ:=
  □∀ (n : nat), WP call vis_odd #n {{ is_equal #(is_odd_pure n) }}.

(* Specification of the module. *)
Definition Λ :=
[
  ("sum", sum_spec) ;
  ("r_val", r_val_spec) ;
  ("lily", is_equal enc_lily) ;
  ("flip", flip_spec) ;
  ("r_elt", is_equal enc_r_elt) ;
  ("is_odd'", is_odd_spec)
].
```

Return

Main menu

records : Proof. I

```
Lemma Records_spec :
  let η := EnvCons "Stdlib" Stdlib $
           EnvNil in
  ⊢ WP eval_mexpr η_Records {{ module_spec Λ}}.
Proof.
  intros η. wp.
  simpl. wp.

  (* [r_elt] is a known value. *)
  wp_bind. wp_continue. wp_bind.

  (* [flip] has the expected spec. *)
  oSpecify "flip" flip_spec vflip "#Hflip".
  { iIntros "!>" (b i); wp.
    wp_continue.
    simpl.
    wp. equality. }
  wp_bind.

  (* [flip] is applied to [r_elt]. *)
  wp.
  replace
    (VRecord (EnvCons "b" VTrue (EnvCons "i" (VInt (int.repr 10)) EnvNil)))
    with #{| b := true; i := 10 |}; last reflexivity.
    wp_use "Hflip". iIntros (? ← ). wp_bind.
```

▸ Return

▸ Main menu

records : Proof. II

```
  (* [lily] has the expected value. *)
  wp_continue. wp_bind.

  (* [r_val] has the expected value. *)
  oSpecify "r_val" r_val_spec vr_val "#Hr_val".
  { iIntros "!>" ([[|] i]); wp; wp_bind; wp_continue; wp_bind; wp_continue; iPureIntro; equality. }
  wp_bind.

  (* [sum] is given the trivial spec for now. *)
  oSpecify "sum" sum_spec vsum "#Hsum".
  { iIntros "!>" ([b1 i1] [b2 i2]).
    wp.
    do 2 wp_continue.
    wp_par; (* ... *). }
  wp_continue. wp_bind.

  (* [is_odd] is given the trivial spec for now. *)
  oSpecify "is_odd" trivial_spec vis_odd "#?"; first done. wp_bind.

  oSpecify "is_odd'" is_odd_spec vis_odd' "#His_odd'".
  { (* ... *) }

  (* Every spec has been proven: [wp_module_spec] can finish the proof. *)
  wp_module_spec.
Time Qed.
```

Extra slides

- ▸ Separation Logic and Iris
- ▸ Weakest Precondition `WP`
- ▸ Examples