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
## 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?

Built-in 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:

\[
\{ \nu \text{ represents the list } l \} \\
\text{call length } \nu \\
\{ \lambda res. \left[ res = \text{length of the list } l \right] \}
\]
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:

\[
\{ v \text{ represents the list } l \} \\
\text{call length } v \\
\{ \lambda res. \text{「}res = \text{length of the list } l\text{」} \}
\]

To verify a program should ensure:

- its safety \(\Rightarrow\) no crash,
- its progress \(\Rightarrow\) it is not stuck,
- the respect of its post-condition \(\phi\).
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:

<table>
<thead>
<tr>
<th>Project</th>
<th>Aspect of the language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cosmo</td>
<td>Multicore OCaml and weak-memory</td>
</tr>
<tr>
<td>iris-time-proofs</td>
<td>Time complexity in presence of lazy</td>
</tr>
<tr>
<td>Hazel</td>
<td>Effect Handlers</td>
</tr>
<tr>
<td>Space-Lambda</td>
<td>Garbage Collection</td>
</tr>
</tbody>
</table>

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 → 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 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 [ ILit (Bindi 1 (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)

```ocaml
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)

```ocaml
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)

```ocaml
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.

```ocaml
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.
```
Goal
Prove programs using Coq tactics.

Steps
1. Give meaning to the syntax, 
   \( \rightarrow \) define an operational semantics for OCaml.
2. Define reasoning rules to reason about this semantics, 
   \( \rightarrow \) these rules are proven once and for all.
3. Define Coq tactics to exploit these rules. 
   \( \rightarrow \) 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 $\rightarrow$ 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 $\rightarrow$ 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.

- e.g. pattern-matching, ADTs, records, modules.

Non-Determinism the order of evaluation of expressions is not defined, and some expressions can be erased;

- e.g. function calls, tuples, dynamic checks.

Solution.

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

- $\text{Ret} (a : A)$
- $\text{Crash}$
- $\text{Next}$
- $\text{Par} \{A1 \ A2\} (m1 : \text{micro } A1)(m2 : \text{micro } A2)$
  - $(k : A1 \ast A2 \rightarrow \text{micro } A)$
  - $(ko : \text{unit } \rightarrow \text{micro } A)$
- $\text{Stop} \{X \ Y\} (c : \text{code } X \ Y)(x : X)$
  - $(k : Y \rightarrow \text{micro } A)$
  - $(ko : \text{unit } \rightarrow \text{micro } A)$.

**Inductive code** : $\text{Type } \rightarrow \text{Type } \rightarrow \text{Type} :=$

- $(\ast \text{ code } X \ Y : \text{Type of a system call.} \newline X : \text{type of the parameter of the syst. call,} \newline Y : \text{type of the returned value. } \ast)$
- $(\ast \text{ Provides:} \newline - \text{Non-deterministic binary choice;} \newline - \text{heap manipulation;} \newline - \text{potential divergence. } \ast)$

(a) Computations of type $A$.  (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. *)

\[
\text{eval } \eta (\text{EApp } e_1 e_2) = \\
\text{Par} (\text{eval } \eta e_1) \\
(\text{eval } \eta e_2) \\
(\lambda '(v_1, v_2), \text{call } v_1 v_2) \\
(\lambda _, \text{Next})
\]
Proofs of programs.

To prove an expression \( e \) is to prove after \((\text{eval } \eta \ e)\) \{\( \phi \}\)

- \(\text{eval } \eta \ e\) : micro val,
- after ensures safety, etc.
Proofs of programs.

To prove an expression \( e \)

is to prove

\[
\text{after (eval } \eta \text{ } e) \{ \phi \}
\]

- \( \text{eval } \eta \text{ } e : \text{micro } \text{val} \),
- \( \text{after } \text{ensures } \text{safety, etc.} \)

A Selection of reasoning rules

\[
\begin{align*}
\text{RET} & \quad \phi(a) \\
\text{PAR} & \quad \forall v_1 v_2. \phi_1(v_1) \rightarrow \phi_2(v_1) \rightarrow \text{after } (k(v_1, v_2)) \{ \phi \} \\
\text{ALLOC} & \quad \triangleright (\forall \ell. \ell \mapsto v \rightarrow \text{after } (k(\ell)) \{ \phi \})
\end{align*}
\]

\[
\text{after } (\text{Stop(CAlloc, } v, k, ko)) \{ \phi \}
\]
An alternative Program Logic for pure programs.

Définition : simp

\[ \text{simp} \; m_1 \; m_2 \triangleq \text{« The computation } m_1 \text{ can be simplified into } m_2. \text{ »} \]

after and simp

\[
\text{SIMP} \quad \frac{\text{simp} \; m_1 \; m_2 \quad \text{after} \; (m_2) \; \{ \phi \} } { \text{after} \; (m_1) \; \{ \phi \} } \]

Two uses of simp:

- Program specification: Let \( f \) be an OCaml function represented by the Gallina function \( f \) and \( a \) be represented by \( a \).
  \[
  \text{simp} \; (\text{call} \; f \; a) \; (\text{Ret} \; (f \; a))
  \]

- Program simplification: simp (eval \( \eta \) \[\begin{array}{c} 1 + 2 + 3 + 4 + 5 \end{array}\]) (Ret 15).
  
  8 function calls
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.
Separation Logic and Iris.
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 \((\star)\) :

\[ W^{40} \vdash W^{30} \star W^{10} \]

- Magic Wand \((\neg\star)\) :

\[ W^{27} \vdash W^{3} \rightarrow\star W^{30} \]
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 $\square P$: $\square P \vdash \square P \ast P$.

- later modality $\triangleright P$: $P$ will hold at the next logical step.

- Fancy-Update modality $\varepsilon_1 \Rightarrow \varepsilon_2 P$: $P$ and invariants whose name appear in $\varepsilon_2$ hold, under the assumption that all invariants whose name occurs in $\varepsilon_1$ hold.

- Basic-Update modality $\overset{.}.| P$: allows to update the ghost state before proving $P$.

Proof techniques provided by Iris

- resources Users can define their own resources;

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

- induction de Löb $\left(\square (\triangleright P \rightarrow P)\right) \rightarrow P$. 
Weakest Precondition.

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

Main menu
Definition of \textit{after}.

Very simplified version: no heap, no invariant.

\textbf{Weakest Precondition}

- If $\exists v. m = \text{Ret}(v)$, then
  \[ \text{after} (m) \{ \Phi \} \triangleq \Phi (v) \]

- Otherwise
  \[ \text{after} (m) \{ \Phi \} \triangleq \exists m'. m \leadsto m' \land * \]
  \[ \forall m'. \neg (m \leadsto m' \land *) \]
  \[ \triangleright \text{after} (m') \{ \Phi \} \]
Definition of after.
Simplified version: there is a heap, but still no invariants.

Logical Heap
For any physical heap $\sigma$, $S(\sigma)$ is an assertion describing the heap. It is provided by Iris.

Weakest Precondition
- If $\exists v. m = \text{Ret}(v)$, then
  $$\text{after}(m)\{\Phi\} \triangleq \forall \sigma. S(\sigma) \not\rightarrow S(\sigma) \ast \Phi(v)$$
- Otherwise
  $$\text{after}(m)\{\Phi\} \triangleq \forall \sigma. S(\sigma) \not\rightarrow \exists \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \not\rightarrow$$
  $$\forall \sigma', m'. \neg(\sigma, m) \leadsto (\sigma', m') \not\rightarrow$$
  $$\triangleright S(\sigma') \ast \text{after}(m')\{\Phi\}$$
Definition of after.

Real definition of after.

Logical Heap

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

Weakest Precondition

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

  \[
  \text{after}_\varepsilon(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \Rightarrow \varepsilon \models_0 \models_\varepsilon S(\sigma) \Rightarrow \Phi(v)
  \]

- Otherwise

  \[
  \text{after}_\varepsilon(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \Rightarrow
  \[
  \varepsilon \models_0 \models_\varepsilon \exists \sigma', m'. (\sigma, m) \rightsquigarrow (\sigma', m') \Rightarrow
  \[
  \forall \sigma', m'. \models (\sigma, m) \rightsquigarrow (\sigma', m') \Rightarrow
  \[
  \models_0 \models_0 \models_\varepsilon S(\sigma') \Rightarrow \text{after}_\varepsilon(m') \{\Phi\}
  \]
Adequacy theorem for after.

**Adequacy theorem**

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

$$\vdash \exists (\Phi : A \to i\text{Prop } \Sigma).\text{after}_T (m_1) \{ \Phi \} \ast (\text{after}_T (S(\sigma_T) \ast m_T) \{ \phi \}) \rightarrow_T \emptyset \nmid \psi$$

then $\psi$ is true.

**Corollary : Progress and respect of the post-condition.**

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

$$\vdash \forall (\text{hypothesis granted access to resources}).\text{after}_T (m_1) \{ \lambda v. \nmid \psi(v) \}$$

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.
Monotone counters.
Counters : code

```ocaml
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
```
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
  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)
Counters : Specification. 1

**Definition** is_counter \((n : \text{nat}) (v : \text{val}) : \text{iProp} \Sigma := \exists (\ell : \text{loc}), \left[ v = \#\ell\downarrow\ell \mapsto \#n. \right] .

**Definition** make_spec \((vmake : \text{val}) : \text{iProp} \Sigma := \Box \text{WP call vmake } #() \{ \lambda res, \text{is_counter } 0 \text{ res } \} .

**Definition** get_spec \((vget : \text{val}) : \text{iProp} \Sigma := \Box \forall (v : \text{val}) (n : \text{nat}), \text{is_counter } n \text{ v } \dashv \text{WP call vget } v \{ \lambda res, \left[ res = \#n\downarrow n * \text{is_counter } n \text{ v } \right] \} .

**Definition** incr_spec \((vincr : \text{val}) : \text{iProp} \Sigma := \Box \forall (v : \text{val}) (n : \text{nat}), \text{is_counter } n \text{ v } \dashv \text{WP call vincr } v \{ \lambda res, \left[ res = \mathsf{VUnit}\mathsf{\uparrow} * \text{is_counter } (S n) \text{ v } \right] \} .

**Definition** set_spec \((vset : \text{val}) : \text{iProp} \Sigma := \Box \forall (v : \text{val}), \text{WP call vset } v \{ \lambda res, \forall (n m : \text{nat}), \left( n \leq m \% \text{nat} \right) \rightarrow \left( \text{representable } n \right) \rightarrow \left( \text{representable } m \right) \rightarrow \text{is_counter } n \text{ v } \dashv \text{WP call res } #m \{ \lambda res, \left[ res = \mathsf{VUnit}\mathsf{\uparrow} * \text{is_counter } m \text{ v } \right] \} \} .
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.
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
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 =
| 0
| S of nat

let rec is_odd' = function
| 0 → true
| S n → not (is_odd' n)
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 fields [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 {
    λvpart,
    WP call vpart #r2 {
      λres,
      is_equal res #(sum_pure r1 r2) } } }].
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) ].
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.
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