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)

18 September 2023
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.
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:

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

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),

```ocaml
(* 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)) ].
Specifications

Goals

A predicate over values that:

- describes the behaviour of a (module-)expression;
- can be recognized (e.g. to skip some breakpoints).

\[ \leftrightarrow \] to reduce the path \( M.N.f \) if \( M \) is a module containing a sub-module \( N \) containing a function \( f \).
Specifications

Goals

A predicate over values that:

- describes the behaviour of a (module-)expression;
- can be recognized (e.g. to skip some breakpoints).

→ to reduce the path \( M.N.f \) if \( M \) is a module containing a sub-module \( N \) containing a function \( f \).

A type to rule them all.

\[
\text{Inductive } \text{spec} : \text{Type} \rightarrow \text{Type} := \\
\hspace{1em} \text{SpecPure } \{A\} : \text{spec_usage} \rightarrow (A \rightarrow \text{Prop}) \rightarrow \text{spec A} \\
\hspace{1em} \text{SpecImpure } \{A\} : \text{spec_usage} \rightarrow (A \rightarrow \text{iProp } \Sigma) \rightarrow \text{spec A} \\
\hspace{1em} \text{SpecEquality } \{A\} : \text{spec_usage} \rightarrow A \rightarrow \text{spec A} \\
\hspace{1em} \text{SpecModule} : \text{spec_usage} \rightarrow \text{list } (\text{string } \ast \text{ spec val}) \rightarrow \text{iProp } \Sigma \rightarrow \text{spec val}.
\]
Example: a toy module. (l)

```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.
```
Description of the tool.

Goal
Prove programs using Coq tactics.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Give meaning to the syntax,</td>
</tr>
<tr>
<td></td>
<td>↩ define an operational semantics for OCaml.</td>
</tr>
<tr>
<td>2</td>
<td>Define reasoning rules to reason about this semantics,</td>
</tr>
<tr>
<td></td>
<td>↩ these rules are proven once and for all.</td>
</tr>
<tr>
<td>3</td>
<td>Define Coq tactics to exploit these rules.</td>
</tr>
<tr>
<td></td>
<td>↩ the tactics rely on aforementioned rules ⇒ they are correct by construction.</td>
</tr>
</tbody>
</table>
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.  
$\leftarrow$ e.g. pattern-matching, ADTs, records, modules.

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

   \[
   \text{Fixpoint } \text{eval} : \text{env} \rightarrow \text{expr} \rightarrow \text{micro val}.
   \]

   \[
   \text{Definition } \text{call} : \text{val} \rightarrow \text{val} \rightarrow \text{micro val}.
   \]

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

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

   \[
   \text{Inductive } \text{step} : \text{store} \ast \text{micro } A \rightarrow \text{store} \ast \text{micro } A \rightarrow \text{Prop}.
   \]
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). |

**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: |
  - potential divergence; *)
| CEval : code (env * env * expr) val |
| CLoop : code (env * var * int * int * expr) val |
(* - non-deterministic binary choices; *)
| CFlip : code unit bool |
(* - heap manipulation. *)
| CAlloc : code val loc |
| CLoad : code loc val |
| CStore : code (loc * val) unit. |

(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. *)

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

Spall-step semantics of store * micro A

**Inductive** step \{A\}: config A → config A → Prop :=
- **StepAlloc** :
  \( \forall \sigma v l k ko, \sigma !! l = \text{None} \rightarrow \)
  \( \text{step} \ (\sigma, \text{Stop CAlloc v k ko}) \)
  \( \langle l := v \rangle \sigma, k l \)
- **StepParLeft** :
  \( \forall \{A1 A2\} \sigma' (m1 m'1 : \text{micro A1}) (m2 : \text{micro A2}) k ko, \)
  \( \text{step} \ (\sigma, m1) (\sigma', m'1) \rightarrow \)
  \( \text{step} \ (\sigma, \text{Par m1 m2 k ko}) \)
  \( (\sigma', \text{Par m'1 m2 k ko}) \)
- **StepParRight** :
  \( \forall \{A1 A2\} \sigma' (m1 : \text{micro A1}) \{m2 m'2 : \text{micro A2}\} k ko, \)
  \( \text{step} \ (\sigma, m2) (\sigma', m'2) \rightarrow \)
  \( \text{step} \ (\sigma, \text{Par m1 m2 k ko}) \)
  \( (\sigma', \text{Par m1 m'2 k ko}). \)

**Figure**: Fragment of the definition of step.
So far, we have seen...

- how to use Osiris,
  - evaluate module-expressions (representing files);
  - use Coq tactics to move forward in the programs (or proofs).
- the semantics of OCaml.
So far, we have seen...  

- how to use Osiris,  
  - evaluate module-expressions (representing files);  
  - use Coq tactics to move forward in the programs (or proofs).  
- the semantics of OCaml.

Next: how to reason about our semantics.

Our goal is:  

- to allow users to use Iris features;  
- to provide an ergonomic tool.
Proofs of programs.

To prove an expression $e$ is to prove

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

- $\text{eval } \eta e : \text{micro val}$,
- $\text{after ensures ...}$
  - safety of the computations,
  - progress,
  - respect of post-conditions.
Proofs of programs.

To prove an expression $e$ is to prove

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

- $\text{eval } \eta e : \text{micro val}$,
- $\text{after}$ ensures ...
  - safety of the computations,
  - progress,
  - respect of post-conditions.

A Selection of reasoning rules

**RET**

<table>
<thead>
<tr>
<th>$\phi(a)$</th>
<th>$\text{after } (\text{Ret } (a)) { \phi }$</th>
</tr>
</thead>
</table>

**PAR**

<table>
<thead>
<tr>
<th>$\forall v_1 v_2. \phi_1 (v_1) \Rightarrow \phi_2 (v_1) \Rightarrow \text{after } (k(v_1, v_2)) { \phi }$</th>
<th>$\text{after } (\text{Par } (m_1, m_2, k, ko)) { \phi }$</th>
</tr>
</thead>
</table>

**ALLOC**

<table>
<thead>
<tr>
<th>$\triangleright (\forall \ell. \ell \mapsto v \Rightarrow \text{after } (k(\ell)) { \phi })$</th>
<th>$\text{after } (\text{Stop } (\text{CAlloc } v, k, ko)) { \phi }$</th>
</tr>
</thead>
</table>
An alternative Program Logic for pure programs.

**Definition:** simp

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

**after and simp**

\[
\begin{align*}
\text{SIMP} & \quad \text{simp } m_1 \text{ } m_2 \quad \text{after } (m_2) \{ \phi \} \\
& \quad \text{after } (m_1) \{ \phi \}
\end{align*}
\]

**Two uses of simp:**

- **Program specification:**
  
  \[ \text{e.g. simp (call length \#1) (Ret (List.length l))} \]

- **Program simplification:** \[ \text{simp (eval } \eta \text{1 + 2 + 3 + 4 + 5) (Ret 15).} \]
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

  $\text{after}(m) \{ \Phi \} \triangleq \exists m'. m \leadsto m' \land \forall m'. \exists m''. m \leadsto m'' \land \neg \ast \triangleright \text{after}(m') \{ \Phi \}$
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
- gives meaning to "$\ell \mapsto v$";
- is provided by Iris.

Weakest Precondition
- If $\exists v. m = \text{Ret}(v)$, then
  $$\text{after}(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \rightarrow* S(\sigma) \star \Phi(v)$$
- Otherwise
  $$\text{after}(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \rightarrow*$$
  $$\forall \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \rightarrow*$$
  $$\forall \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \rightarrow*$$
  $$\triangleright S(\sigma') \star \text{after}(m') \{\Phi\}$$
Definition of after.

Real definition of after.

Logical Heap

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

- gives meaning to “$\ell \mapsto v$”;
- is provided by Iris.

Weakest Precondition

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

$$\text{after}_\varepsilon(m) \{ \Phi \} \triangleq \forall \sigma . S(\sigma) \rightarrow* \varepsilon \models_0 0 \models_\varepsilon S(\sigma) \rightarrow* \Phi(v)$$

- Otherwise

$$\text{after}_\varepsilon(m) \{ \Phi \} \triangleq \forall \sigma . S(\sigma) \rightarrow*$$

$$\varepsilon \models_0 \exists \sigma', m' . (\sigma, m) \rightarrow (\sigma', m') \rightarrow*$$

$$\forall \sigma', m' . \models (\sigma, m) \rightarrow (\sigma', m') \rightarrow*$$

$$0 \models_0 0 \models_0 0 \models_\varepsilon S(\sigma') \rightarrow* \text{after}_\varepsilon(m') \{ \Phi \}$$
after-related rules

\[
\begin{align*}
\text{RET} & \quad \frac{\phi(a)}{\text{after}_E \, \text{Ret}(a) \{ \phi \}} \\
\text{FLIP} & \quad \frac{\phi}{\text{after} \,(\text{Stop}(\text{CFlip}, (), k, ko)) \{ \phi \}} \\
\text{BIND-2} & \quad \frac{\text{after} \,(m_1) \{ \psi \} \quad \forall v.\psi(v) \rightarrow \ast \text{after} \,(m_2(v)) \{ \phi \}}{\text{after} \,(\text{bind}(m_1, m_2)) \{ \phi \}} \\
\text{ALLOC} & \quad \frac{\phi}{\text{after} \,(\text{Stop}(\text{CAlloc}, v, k, ko)) \{ \phi \}} \\
\text{STORE} & \quad \frac{\ell \mapsto v \ast \phi}{\text{after} \,(\text{Stop}(\text{CStore}, (\ell, v'), k, ko)) \{ \phi \}} \\
\text{LOAD} & \quad \frac{\ell \mapsto_q v \ast \phi}{\text{after} \,(\text{Stop}(\text{CLoad}, \ell, k, ko)) \{ \phi \}}
\end{align*}
\]
Inductive simp {A : Type} : micro A → micro A → Prop :=
  | SimpFlip :
    ∀ x k ko m,
    simp (k false) m →
    simp (k true) m →
    simp
    (Stop CFlip x k ko)
    m
  | SimpParRetLeft:
    ∀ {A1 A2} (a1 : A1) (m2 : micro A2) k ko,
    simp
    (Par (Ret a1) m2 k ko)
    (try m2 (λ v2, k (a1, v2)) ko)
  | SimpPar:
    ∀ {A1 A2} m1 m'1 m2 m'2 (k : A1 * A2 → micro A) ko,
    simp m1 m'1 →
    simp m2 m'2 →
    simp (Par m1 m2 k ko) (Par m'1 m'2 k ko)
  | SimpReflexive:
    ∀ m,
    simp m m
  | SimpTransitive:
    ∀ m1 m2 m3,
    simp m1 m2 →
    simp m2 m3 →
    simp m1 m3
Another example.
### 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 \textit{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 (\(\ast\)):

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

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

\[ W^{27} \vdash W^3 \ast \neg W^{30} \]
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 $\Box P$: $\Box P \vdash \Box P \ast P$.
- *later* modality $\triangleright P$: $P$ will hold at the next logical step.
- Fancy-Update modality $\mathcal{E}_1 \mathrel{\leadsto} \mathcal{E}_2 P$: $P$ 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 $\mathcal{E} P$: allows to update the ghost state before proving $P$.

Proof techniques provided by Iris

- **resources** Users can define their own resources;
- **invariants** $\mathcal{P}^N$ is a logical black box containing $P$. The name $N$ is associated with the box;
- **induction de L"ob** $(\Box (\triangleright P \ast P)) \ast P$. 
Weakest Precondition.
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
  \[ \text{after}(m) \{ \Phi \} \triangleq \exists m'. m \xrightarrow{\neg} m' \star \]
  \[ \forall m'. \neg m \xrightarrow{\neg} m' \star \]
  \[ \triangleright \text{after}(m') \{ \Phi \} \]
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
  
  $\text{after}(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \rightarrow S(\sigma) \ast \Phi(v)$

- Otherwise
  
  $\text{after}(m) \{\Phi\} \triangleq \forall \sigma. S(\sigma) \rightarrow$

  $\exists \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \upharpoonright \ast$

  $\forall \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \upharpoonright \rightarrow$

  $\triangleright S(\sigma') \ast \text{after}(m') \{\Phi\}$
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

\[
\text{after}_E (m) \{ \Phi \} \triangleq \forall \sigma. S(\sigma) \rightarrow_E \exists \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \downarrow^* \\
\forall \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \downarrow^* \Rightarrow \exists \sigma', m'. (\sigma, m) \leadsto (\sigma', m') \downarrow^* \\
\Rightarrow S(\sigma') \rightarrow_E \text{after}_E (m') \{ \Phi \}
\]

- Otherwise

\[
\text{after}_E (m) \{ \Phi \} \triangleq \forall \sigma. S(\sigma) \rightarrow^* \\
\text{after}_E (m) \{ \Phi \} \triangleq \forall \sigma. S(\sigma) \rightarrow^*
\]
Adequacy theorem for \texttt{after}.

**Adequacy theorem**

Let $A$ be a type, $m_1$ and $m_n$ terms of type 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:

$$
\vdash \top \Rightarrow \exists (\Phi : A \rightarrow iProp \Sigma). \text{after}_T (m_1) \{ \Phi \}^* (\text{after}_T (S(\sigma_T) \ast m_T) \{ \phi \} \rightarrow T \Rightarrow_\emptyset \neg \psi)
$$

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 micro $A$, $\sigma_n$ a heap, $n$ a natural integer and $\psi$ a pure post-condition (i.e. of type $A \rightarrow \text{Prop}$).

If $(\emptyset, m_1)$ reduces to $(\sigma_n, m_n)$ in $n$ steps, and that the following assertion holds:

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

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

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
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)
Counters : Specification. I

**Definition** is_counter (n : nat) (v : val) : iProp Σ :=

\( \exists (\ell : loc), [v = \#\ell\#\ell \mapsto \#n]. \)

**Definition** make_spec (vmake : val) : iProp Σ :=

\( \Box WP \; call \; vmake \; \#() \{\{ \lambda res, is-counter \; 0 \; res \}\}. \)

**Definition** get_spec (vget : val) : iProp Σ :=

\( \Box \forall (v : val) (n : nat), \)

is_counter n v \( \mapsto \) WP call vget v \( \{\{ \lambda res, [res = \#n\#n \mapsto \lambda is-counter \; n \; v] \}\}. \)

**Definition** incr_spec (vincr : val) : iProp Σ :=

\( \Box \forall (v : val) (n : nat), \)

is_counter n v \( \mapsto \)

WP call vincr v \( \{\{ \lambda res, [res = VUnit\#\#n \mapsto \lambda is-counter \; (S \; n) \; v] \}\}. \)

**Definition** set_spec (vset : val) : iProp Σ :=

\( \Box \forall (v : val), \)

WP call vset v \( \{\{ \lambda res, \forall (n m : nat), \)

\( [n \leq m]%nat\#\#n \mapsto \)

representable n \#\rightarrow \)

representable m \#\rightarrow \)

is_counter n v \( \mapsto \)

WP call res \#m \( \{\{ \lambda res, [res = VUnit\#\#n \mapsto \lambda is-counter \; m \; v] \}\}. \)
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 \( \eta \)_Counters \( \{ \{ \text{File_spec} \} \} \).

Proof using \( \text{H\( \eta \) osirisGS0} \) \( \Sigma \eta \).

  oSpecify "make" make_spec vmake "#Hmake" !.
  { iIntros "!>".
    @oCall unfold; wp_bind; wp_continue.
    wp_alloc \( \ell \) "[H\( \ell \_\)]".
    iExists \( \ell \).
    iSplit; first equality.
    by cbn. }

  oSpecify "incr" incr_spec vincr "#Hincr" !.
  { iIntros "!>" (? n) "(\%\(\ell\&\to\&H\ell\))".
    call. wp_load "H\( \ell \)". wp_store "H\( \ell \)".
    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) "(\%\(\ell\&\to\&H\ell\))".
    call. wp_load "H\( \ell \)". prove_counter. }

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

iModIntro; wp_prove_spec.
Qed.
Records

• Code
• Specifications
• Proof
```ocaml
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
```

```ocaml
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
```

```ocaml
let rec is_odd n =
  | 0 -> true
  | S of nat
    -> 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
   field [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 \{\n      λvpart,
      WP call vpart #r2 \{\n         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