Osiris: Towards Formal Semantics and Reasoning for OCaml

Remy Seassau, Irene Yoon, Jean-Marie Madiot, François Pottier

February 4, 2025

Inria Paris

What is OCaml?

```
let rec sum l =
  match l with
  | [] -> 0
  | h :: t -> h + sum t
```

"An industrial-strength functional programming language with an emphasis on expressiveness and safety."

What is OCaml?

```
let rec sum l =
  match l with
  | [] -> 0
  | h :: t -> h + sum t
```

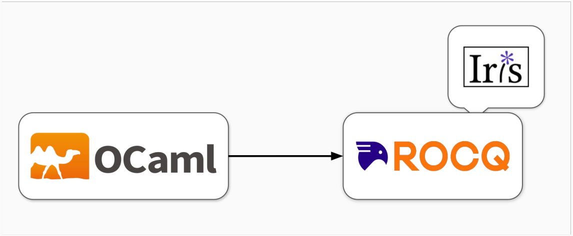
"An industrial-strength functional programming language with an emphasis on expressiveness and safety."

```
let sum l =
  let res = ref 0 in
  List.iter (fun x -> res := !res + x) l;
  !res
```

OCaml Has No Formal Semantics

"This document is intended as a reference manual for the OCaml language. It lists the language constructs, and gives their precise syntax and informal semantics. [...] No attempt has been made at mathematical rigor: words are employed with their intuitive meaning, without further definition."

What is Osiris?



Inside the Rocq proof assistant, the Osiris project aims to build:

- A representation of the syntax of OCaml 5;
- A formal *semantics*; — the focus of this talk
- A program verification environment, which includes:
  - A Hoare Logic for pure expressions;
  - An Iris-based Separation Logic for arbitrary expressions.

Which Semantic Style?

Our semantics is *untyped*.

Its architecture is in two layers:

- a *monadic interpreter*;
- a *small-step semantics* for monadic computations.

The interpreter has type:

$$\mathit{eval} : \mathit{env} \to \mathit{expr} \to \mathit{micro}\ \mathit{val}\ \mathit{exn}$$

It can also be viewed as a translation of OCaml into simpler *"microcode"*.

A Monadic Definitional Interpreter

The monad encapsulates all of the computational effects that we need:

- Exceptions
- Divergence
- State
- Nondeterminism / Parallelism
- Delimited Control

Outline of this Talk

A Monadic Definitional Interpreter for OCaml

The Monad's Public API

The Monad's Internal Syntax

The Monad's Small-Step Semantics

Program Logics

A Monadic Definitional Interpreter for OCaml

Crashing and Exceptions

e_1 && e_2

```
Fixpoint eval η e :=
  match e with
  | EBoolConj e1 e2 =>
      b <- as_bool (eval η e1) ;
      if b then eval η e2 else ret VFalse
  | ERaise e =>
      exn <- eval η e ;
      throw exn
  | ...
```

Crashing and Exceptions

e_1 && e_2

```
Fixpoint eval η e :=
  match e with
  | EBoolConj e1 e2 =>
      b <- as_bool (eval η e1) ;
      if b then eval η e2 else ret VFalse
  | ERaise e =>
      exn <- eval η e ;
      throw exn
  | ...
```

```
Definition as_bool (v : val) :=
  match v with
  | VFalse => ret false
  | VTrue => ret true
  | _ => crash
  end.
```

Crashing and Exceptions

```
e₁ && e₂
raise e
```

```
Fixpoint eval η e :=
  match e with
  | EBoolConj e1 e2 =>
      b <- as_bool (eval η e1) ;
      if b then eval η e2 else ret VFalse
  | ERaise e =>
      exn <- eval η e ;
      throw exn
  | ...
```

```
Definition as_bool (v : val) :=
  match v with
  | VFalse => ret false
  | VTrue => ret true
  | _ => crash
  end.
```

Divergence

```
Fixpoint eval η e :=
  match e with
  | ...
  | EWhile e₁ e₂ =>
      b <- as_bool (eval η e₁) ;
      if b then
        _ <- eval η e₂ ;
        eval η (EWhile e₁ e₂)
      else
        ret VUnit
  | ...
```

```
while e₁ do
  e₂
done
```

Divergence

```
Fixpoint eval η e :=
  match e with
  | ...
  | EWhile e₁ e₂ =>
      b <- as_bool (eval η e₁) ;
      if b then
        _ <- eval η e₂ ;
        eval η (EWhile e₁ e₂)
      else
        ret VUnit
  | ...
```

```
while e₁ do
  e₂
done
```

Divergence

```
Fixpoint eval η e :=
  match e with
  | ...
  | EWhile e₁ e₂ =>
      b <- as_bool (eval η e₁) ;
      if b then
        _ <- eval η e₂ ;
        please_eval η (EWhile e₁ e₂)
      else
        ret VUnit
  | ...
```

```
while e₁ do
  e₂
done
```

State

```
| ...
| ERef e =>
    v <- eval η e ;
    l <- alloc v ;
    ret (VLoc l)
| ELoad e =>
    l <- as_loc (eval η e) ;
    load l
| EStore e₁ e₂ =>
    v <- eval η e₂ ;
    l <- as_loc (eval η e₁) ;
    _ <- store l v
| ...
```

```
let x = ref 0 in
let y = !x in
x := y + 1
```

Nondeterminism / Parallelism

In OCaml, evaluation order is unspecified.

```
...
| EApp e₁ e₂ =>
    (v₁, v₂) <- par (eval η e₁) (eval η e₂) ;
    match v₁ with
    | VClo η (AnonFun x e) =>
        please_eval ((x, v₂) :: η) e
    | _ =>
        crash
    end
| ...
```

Delimited Control

```
effect Get : int
effect Set : int -> unit

let run (init : int) (main : unit -> 'a) : 'a =
  let var = ref init in
  match main () with
  | res -> res
  | effect Get, k -> continue k (!var)
  | effect (Set y), k -> var := y; continue k ()

let (i : int) =
  run 0 @@ fun () ->
    perform (Set 1); perform Get
```

Delimited Control

```
match e with
| p₁ -> e₁
| exception p₂ -> e₂
| effect p₃, k -> e₃
```

```
Fixpoint eval η e :=
  ...
  | EMatch e bs =>
      handle (eval η e) (fun o => deep_match η o bs)
  | EPerform e =>
      eff <- eval η e ;
      perform eff
  | EContinue e1 e2 =>
      k <- as_cont (eval η e1) ;
      v <- eval η e2 ;
      resume k (O2Ret v)
  | EDiscontinue e1 e2 =>
      k <- as_cont (eval η e1) ;
      v <- eval η e2 ;
      resume k (O2Throw v)
  | ...
```

The Monad's Public API

Types

Inductive $\mathit{outcome}_2\ A\ E :=$

$\quad \mathit{O2Ret}\ (a : A) \mid \mathit{O2Throw}\ (e : E)$

Inductive $\mathit{outcome}_3\ A\ E :=$

$\quad \mathit{O3Ret}\ (a : A) \mid \mathit{O3Throw}\ (e : E) \mid \mathit{O3Perform}\ (v : \mathit{val})\ (\ell : \mathit{loc})$

$\mathit{micro}\ A\ E : \quad \mathit{Type}$

In the next slides, for brevity, the parameters of these types are *hidden*.

Final Results; Sequencing

$$
\begin{aligned}
\mathit{ret} &: & A \to \mathit{micro} \\
\mathit{throw} &: & E \to \mathit{micro} \\
\mathit{crash} &: & \mathit{micro} \\
\mathit{try}_2 &: & \mathit{micro} \to (\mathit{outcome}_2 \to \mathit{micro}) \to \mathit{micro} \\
\mathit{bind} &: & \mathit{micro} \to (A \to \mathit{micro}) \to \mathit{micro}
\end{aligned}
$$

Ad Hoc Combinators

$$
\begin{aligned}
\mathit{please\_eval} &: \mathit{env} \to \mathit{expr} \to \mathit{micro} \\
\mathit{alloc} &: \mathit{val} \to \mathit{micro} \\
\mathit{load} &: \mathit{loc} \to \mathit{micro} \\
\mathit{store} &: \mathit{loc} \to \mathit{val} \to \mathit{micro} \\
\mathit{par} &: \mathit{micro} \to \mathit{micro} \to \mathit{micro} \\
\mathit{choose} &: \mathit{micro} \to \mathit{micro} \to \mathit{micro} \\
\mathit{handle} &: \mathit{micro} \to (\mathit{outcome}_3 \to \mathit{micro}) \to \mathit{micro} \\
\mathit{perform} &: \mathit{val} \to \mathit{micro} \\
\mathit{resume} &: \mathit{loc} \to \mathit{outcome}_2 \to \mathit{micro} \\
\mathit{install} &: \mathit{bool} \to \mathit{loc} \to \mathit{env} \to \mathit{handler} \to \mathit{micro}
\end{aligned}
$$

The Monad's Internal Syntax

A Syntax for Computations

A monadic computation is a piece of *syntax*. It is a *tree*, where:

- *Ret*, *Throw*, *Crash* are leaves;
- A *Stop* node represents a *"system call"* and carries one child for each possible result;
- A *Par* node allows parallel computation;
- A *Handle* node serves as a delimiter of control effects and carries an effect handler.

Ret and *Stop* alone form the *freer monad* (Kiselyov & Ishii, 2015).

A Syntax for Computations

Inductive $\mathit{micro}\ A\ E :=$

$$
\begin{array}{lll}
\mid \mathit{Ret} & : & A \to \mathit{micro}\ A\ E \\
\mid \mathit{Throw} & : & E \to \mathit{micro}\ A\ E \\
\mid \mathit{Crash} & : & \mathit{micro}\ A\ E \\
\mid \mathit{Stop}\ (!) & : & \mathit{code}\ X\ Y\ E' \to X \to \\
 & & (\mathit{outcome}_2\ Y\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E \\
\mid \mathit{Par} & : & \mathit{micro}\ A_1\ E' \to \mathit{micro}\ A_2\ E' \to \\
 & & (\mathit{outcome}_2\ (A_1 \times A_2)\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E \\
\mid \mathit{Handle} & : & \mathit{micro}\ \mathit{val}\ \mathit{exn} \to \\
 & & (\mathit{outcome}_3\ \mathit{val}\ \mathit{exn} \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E
\end{array}
$$

System Calls

These system calls suffice for our purposes:

$$
\begin{array}{lcl}
\text{Inductive}\ \mathit{code} & : & \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} := \\
\quad \mid \mathit{CEval} & : & \mathit{code}\ (\mathit{env} \times \mathit{expr})\ \mathit{val}\ \mathit{exn} \\
\quad \mid \mathit{CFlip} & : & \mathit{code}\ \mathit{unit}\ \mathit{bool}\ \mathit{exn} \\
\quad \mid \mathit{CAlloc} & : & \mathit{code}\ \mathit{val}\ \mathit{loc}\ \mathit{exn} \\
\quad \mid \mathit{CLoad} & : & \mathit{code}\ \mathit{loc}\ \mathit{val}\ \mathit{exn} \\
\quad \mid \mathit{CStore} & : & \mathit{code}\ (\mathit{loc} \times \mathit{val})\ \mathit{unit}\ \mathit{exn} \\
\quad \mid \mathit{CPerf} & : & \mathit{code}\ \mathit{val}\ \mathit{val}\ \mathit{exn} \\
\quad \mid \mathit{CResume} & : & \mathit{code}\ (\mathit{loc} \times \mathit{outcome}_2\ \mathit{val}\ \mathit{exn})\ \mathit{val}\ \mathit{exn} \\
\quad \mid \mathit{CInstall} & : & \mathit{code}\ (\mathit{bool} \times \mathit{loc} \times \mathit{env} \times \mathit{handler})\ \mathit{loc}\ \mathit{exn}
\end{array}
$$

The Monad's Small-Step Semantics

A Small-Step Reduction Semantics

The meaning, or *behavior*, of a computation is given by a *small-step semantics*.

$$m \;/\; \sigma \longrightarrow m' \;/\; \sigma'$$

A *heap* σ maps memory locations to values (v) or continuations (k or $\lightning$).

Divergence

The system call *CEval* reduces to a recursive call to *eval*.

$$! \, \mathit{CEval} \, (\eta, e) \, k \, / \, \sigma \quad \longrightarrow \quad \mathit{try}_2 \, (\mathit{eval} \; \eta \; e) \, k \, / \, \sigma$$

As a special case, *please_eval* η *e* reduces to *eval* η *e*.

This technique is inspired by McBride (2015).

Non-determinism

The system call *CFlip* produces an arbitrary Boolean result b.

$$! \, \mathit{CFlip} \, () \, k \, / \, \sigma \quad \longrightarrow \quad \mathit{continue} \; k \; b \, / \, \sigma$$

$\mathit{continue}\ k\ v$ stands for $k\ (\mathit{O2Ret}\ v)$.

$\mathit{discontinue}\ k\ v$ stands for $k\ (\mathit{O2Throw}\ v)$.

State

The system calls *CAlloc*, *CLoad*, *CStore* deal with ML-style references.

$$
\begin{array}{rcl}
!\,\mathit{CAlloc}\;v\;k\,/\,\sigma & \longrightarrow & \mathit{continue}\;k\;\ell\,/\,[\ell := v]\sigma \\
 & & \text{if } \ell \notin \mathit{dom}(\sigma) \\
!\,\mathit{CLoad}\;\ell\;k\,/\,\sigma & \longrightarrow & \mathit{continue}\;k\;v\,/\,\sigma \\
 & & \text{if } \sigma(\ell) = v \\
!\,\mathit{CLoad}\;\ell\;k\,/\,\sigma & \longrightarrow & \mathit{Crash}\,/\,\sigma \\
 & & \text{otherwise} \\
!\,\mathit{CStore}\,(\ell, v')\;k\,/\,\sigma & \longrightarrow & \mathit{continue}\;k\;()\,/\,[\ell := v']\sigma \\
 & & \text{if } \sigma(\ell) = v \\
!\,\mathit{CStore}\,(\ell, v')\;k\,/\,\sigma & \longrightarrow & \mathit{Crash}\,/\,\sigma \\
 & & \text{otherwise}
\end{array}
$$

Nondeterminism / Parallelism

Par offers fork/join parallelism (with nondeterministic interleaving).

$$
\begin{array}{rcr}
\mathit{Par}\ m_1\ m_2\ /\ \sigma & \longrightarrow & \mathit{Par}\ m_1'\ m_2\ /\ \sigma' \\
 & & \text{if } m_1\ /\ \sigma \longrightarrow m_1'\ /\ \sigma' \\
\mathit{Par}\,(\mathit{Ret}\ v_1)\,(\mathit{Ret}\ v_2)\ k\ /\ \sigma & \longrightarrow & \mathit{continue}\ k\ (v_1, v_2)\ /\ \sigma \\
\mathit{Par}\ \mathit{Crash}\ m_2\ k\ /\ \sigma & \longrightarrow & \mathit{Crash}\ /\ \sigma \\
\mathit{Par}\,(\mathit{Throw}\ v)\ m_2\ k\ /\ \sigma & \longrightarrow & \mathit{discontinue}\ k\ v\ /\ \sigma \\
\mathit{Par}\,(!\ \mathit{CPerf}\ v\ k)\ m_2\ k'\ /\ \sigma & \longrightarrow & !\ \mathit{CPerf}\ v\,(\lambda o.\ \mathit{Par}\,(k\ o)\ m_2\ k')\ /\ \sigma
\end{array}
$$

Delimited Control

Handle observes a computation's $outcome_3$ and invokes a handler.

$$
\begin{array}{rcl}
Handle\ (Ret\ v)\ h\ /\ \sigma & \longrightarrow & h\ (O3Ret\ v)\ /\ \sigma \\
Handle\ (Throw\ v)\ h\ /\ \sigma & \longrightarrow & h\ (O3Throw\ v)\ /\ \sigma \\
Handle\ (!\ CPerf\ v\ k)\ h\ /\ \sigma & \longrightarrow & h\ (O3Perform\ v\ \ell)\ /\ [\ell := k]\sigma \\
 & & \text{if } \ell \notin dom(\sigma) \\
Handle\ Crash\ h\ /\ \sigma & \longrightarrow & Crash\ /\ \sigma \\
Handle\ m\ h\ /\ \sigma & \longrightarrow & Handle\ m'\ h\ /\ \sigma' \\
 & & \text{if } m\ /\ \sigma \longrightarrow m'\ /\ \sigma'
\end{array}
$$

Delimited Control

The system call *CResume* fetches and resumes a stored continuation.

$$\begin{array}{rcl} !\,\mathit{CResume}\,(\ell, o)\, k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(k'\ o)\, k \,/\, [\ell := \lightning]\sigma \\ & & \text{if } \sigma(\ell) = k' \\ !\,\mathit{CResume}\,(\ell, o)\, k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma \\ & & \text{otherwise} \end{array}$$

Delimited Control

The system call *CInstall* wraps a stored continuation in an effect handler, yielding a new stored continuation.

$$! \, \mathit{CInstall}\, (\mathit{deep}, \eta, \ell, \mathit{bs})\, k \,/\, \sigma \quad \longrightarrow \quad \mathit{continue}\ k\ \ell' \,/\, [\ell' := k']\sigma \qquad \text{if } \ell' \notin \mathit{dom}(\sigma)$$

where $k' = \lambda o.\, \mathit{Handle}\ (\mathit{resume}\ \ell\ o)\ (\lambda o.\, \mathit{eval\_match}\ \mathit{deep}\ \eta\ o\ \mathit{bs})$

Program Logics

Hoare-Style Reasoning About Pure Programs

We isolate a "pure" subrelation $m \longrightarrow_{pure} m'$ (omitted).

Based on it, we define a (total) (dual-postcondition) Hoare Logic:

$$\frac{\varphi(v)}{pure\ (ret\ v)\ \varphi\ \psi} \qquad \frac{\psi(e)}{pure\ (throw\ e)\ \varphi\ \psi}$$

$$\frac{\begin{array}{c}\exists m' \quad m \longrightarrow_{pure} m' \\ \forall m' \quad m \longrightarrow_{pure} m' \Rightarrow pure\ m'\ \varphi\ \psi\end{array}}{pure\ m\ \varphi\ \psi}$$

pure $m\ \varphi\ \psi$ means m terminates and obeys the postconditions φ and ψ.

Hoare-Style Reasoning About Pure Programs

We prove a number of reasoning rules,
first at the level of the *monadic syntax* (omitted),
then at the level of OCaml's *surface syntax*.

$$\frac{\mathit{pure}\ (\mathit{eval}\ \eta\ e_1)\ \varphi_1\ \psi \qquad \mathit{pure}\ (\mathit{eval}\ \eta\ e_2)\ \varphi_2\ \psi \qquad (\forall x_1\ x_2.\ \varphi_1\ x_1\ \rightarrow \varphi_2\ x_2 \rightarrow \varphi\ (x_1 + x_2))}{\mathit{pure}\ (\mathit{eval}\ \eta\ (e_1 +\ e_2))\ \varphi\ \psi}$$

Iris-Style Reasoning About Impure Programs

We define a (partial) (dual-postcondition) Iris-based Separation Logic.

Its judgement is parameterized with a *protocol* (de Vilhena and P., 2021).

We establish a connection between the two logics:

$$\frac{\textit{pure}\ m\ \varphi\ \psi}{\mathsf{ewp}\langle\bot\rangle\ m\ \lceil\varphi\rceil\ \lceil\psi\rceil}$$

Conclusion (So Far)

We have built

- a formal semantics for a large sequential subset of OCaml 5;
- a Hore Logic for pure expressions;
- an Iris-based Separation Logic for arbitrary expressions.

Ongoing and future work:

- Make our program logics more comfortable for end users.
- Support a larger subset of OCaml (e.g., modules; concurrency).