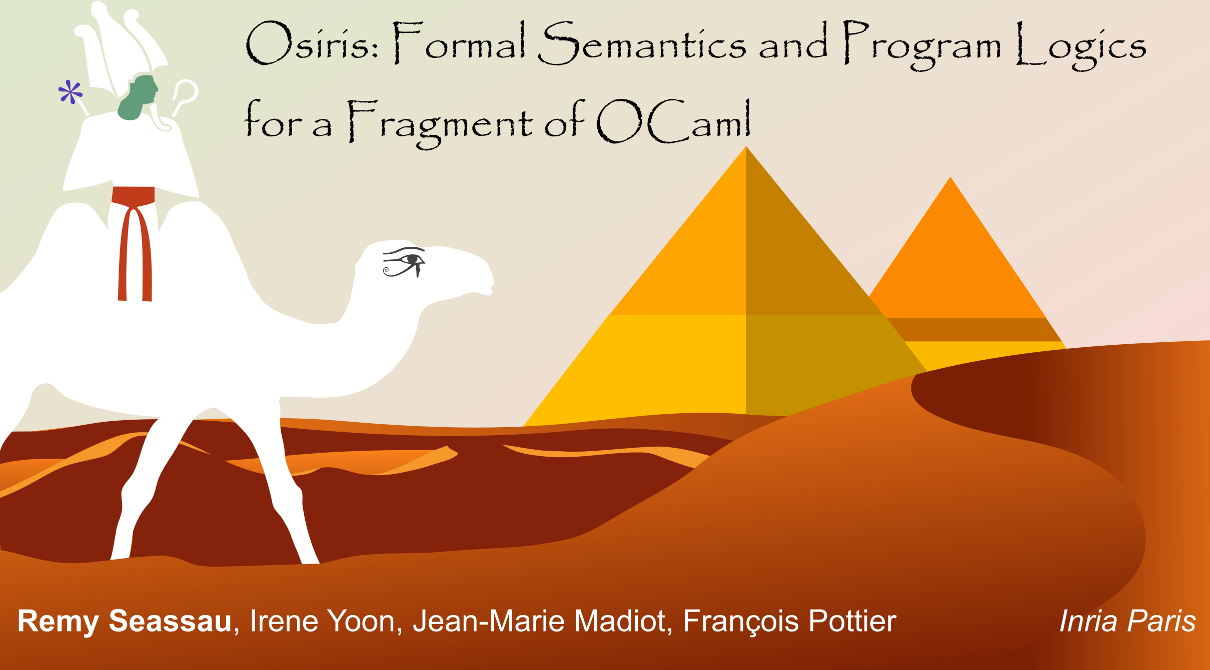
Osiris: Formal Semantics and Program Logics for a Fragment of OCaml



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

Inria Paris

The Osiris Project:

- No object system
- No formalisation of the type system

The Osiris Project:

- No object system
- No formalisation of the type system

Everything else is fair game

The Osiris Project:

- No object system
- No formalisation of the type system

Everything else is fair game

This talk:

- Sequential OCaml 5
- Effect Handlers

What is Osiris?

ROCQ

- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

Automatically generate

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

What is Osiris?

- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

Automatically generate

ROCQ

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

Prove properties about

Iris

- proofs
  - alpha_renaming.v
  - beta_red.v
  - environment.v
  - inductive.v
  - instructionHandler.v
  - substitution.v
  - typing.v

OCaml Has No Formal Semantics

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

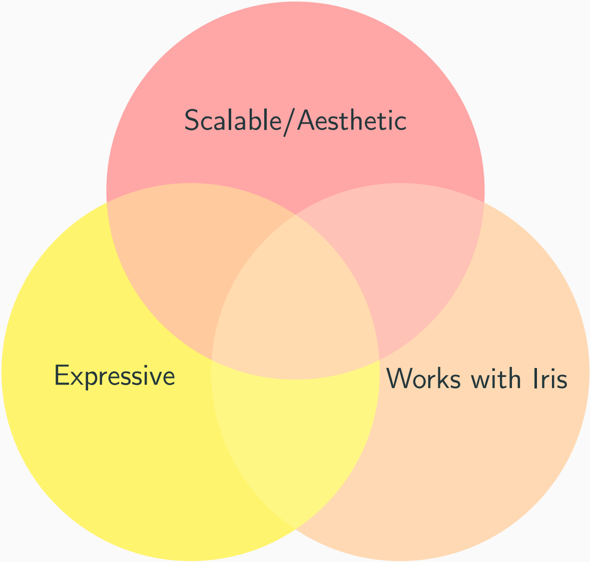
OCaml Has No Formal Semantics (yet!)

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

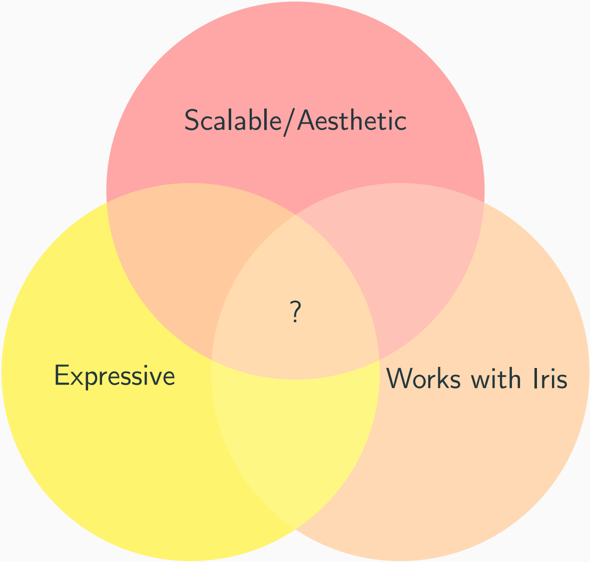
Goals:

- Design and implement formal, mechanized semantics for OCaml
- Build a framework to reason about these semantics

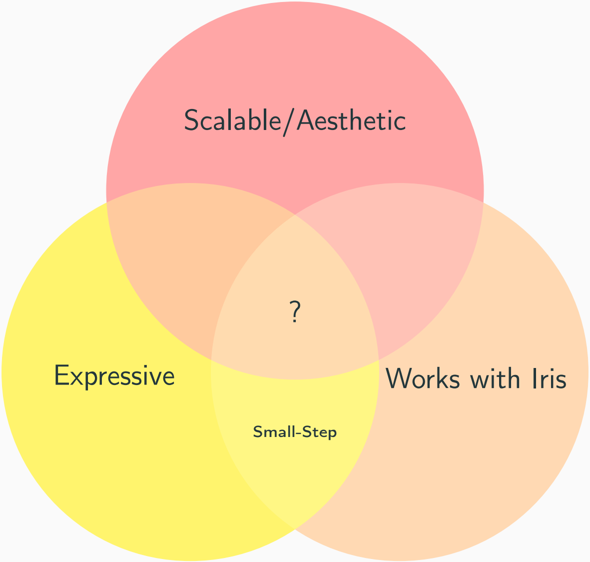
Design Constraints



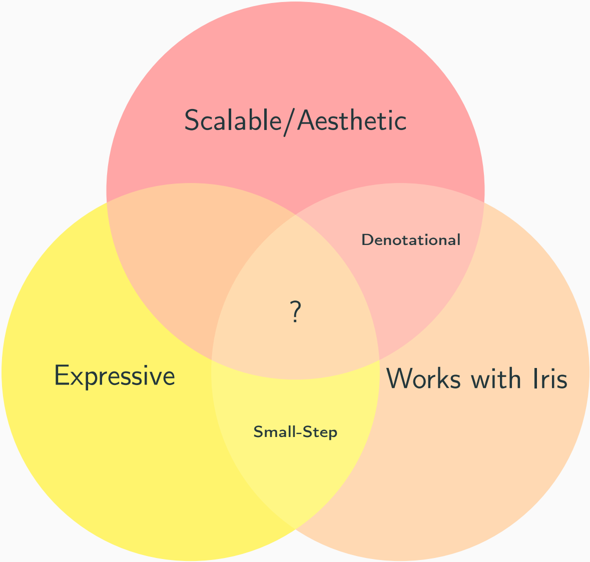
Design Constraints



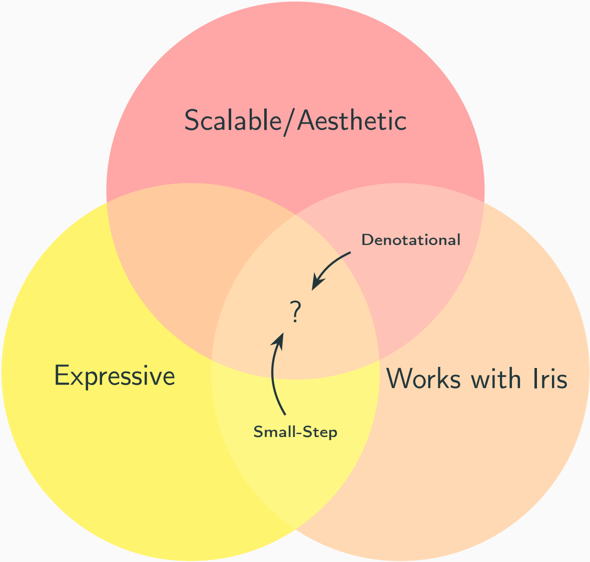
Design Constraints



Design Constraints



Design Constraints



ROCQ

Automatically generate

- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

Prove properties about

Iris

- proofs
  - alpha_renaming.v
  - beta_red.v
  - environment.v
  - inductive.v
  - instructionHandler.v
  - substitution.v
  - typing.v

ROCQ

Automatically generate

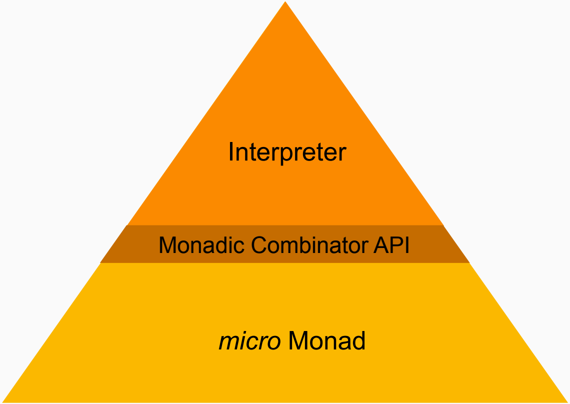
- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

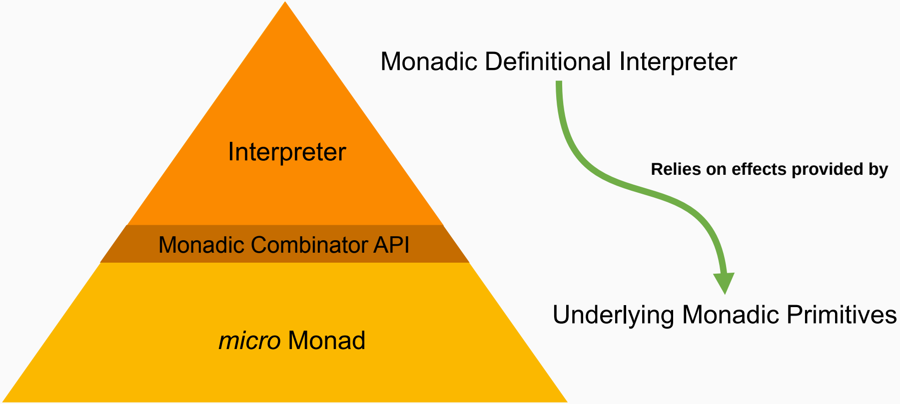
Iris

- proofs
  - alpha_renaming.v
  - beta_red.v
  - environment.v
  - inductive.v
  - instructionHandler.v
  - substitution.v
  - typing.v

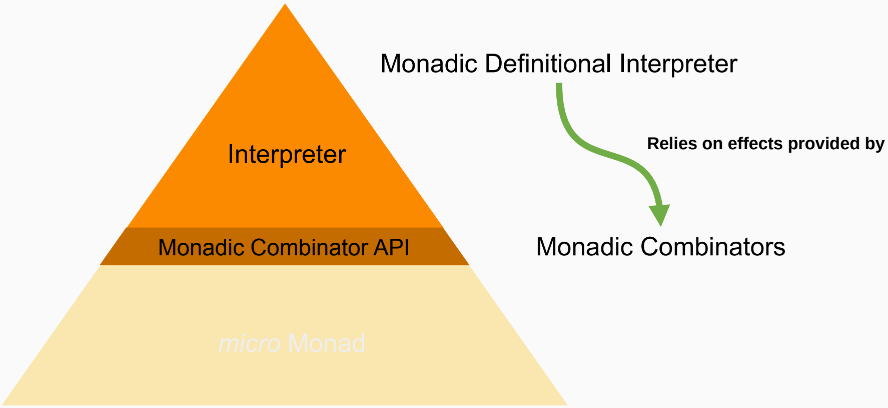
A Layered Monadic Definitional Interpreter



A Layered Monadic Definitional Interpreter



A Layered Monadic Definitional Interpreter



```
Fixpoint eval η e : micro val exn :=
  match e with
  | ...
```

```
Fixpoint eval η e : micro val exn :=
  match e with
  | ...
```

The top-level interpreter has type:

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

```
Fixpoint eval η e : micro val exn :=
  match e with
  | ...
```

The top-level interpreter has type:

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

```
Inductive val :=
  | VInt (i : int) | VString (s: string)
  | VTuple (vs : list val)
  | VData (c : data) (v : list val)
  | VClo (η : env) (x : var) (e : expr)
  | VCont (l : loc)
  | ...
```

```
Fixpoint eval η e : micro val exn :=
  match e with
  | ...
  | EApp e1 e2 ⇒
      '((η, x, e), v2) ← par (as_clo (eval η e1)) (eval η e2) ;
      eval ((x, v2) :: η) e
  | ...
```

```
Fixpoint eval η e : micro val exn :=
  match e with
  | ...
  | EApp e1 e2 ⇒
      '((η, x, e), v2) ← par (as_clo (eval η e1)) (eval η e2) ;
      please_eval ((x, v2) :: η) e
  | ...
```

```
perform e

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

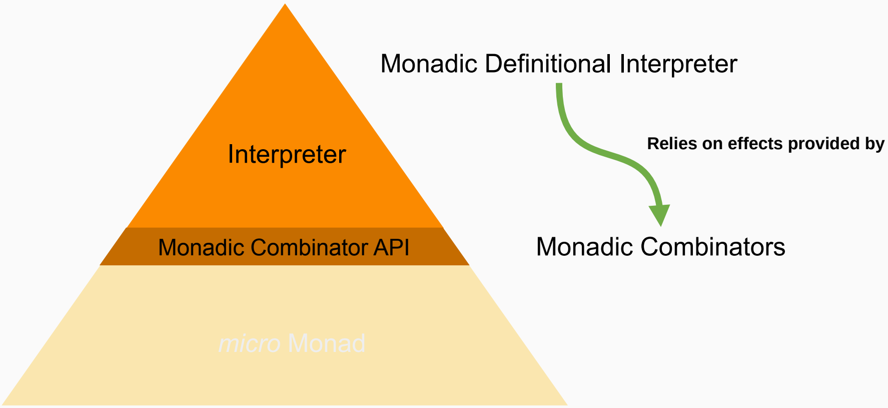
```
perform e

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

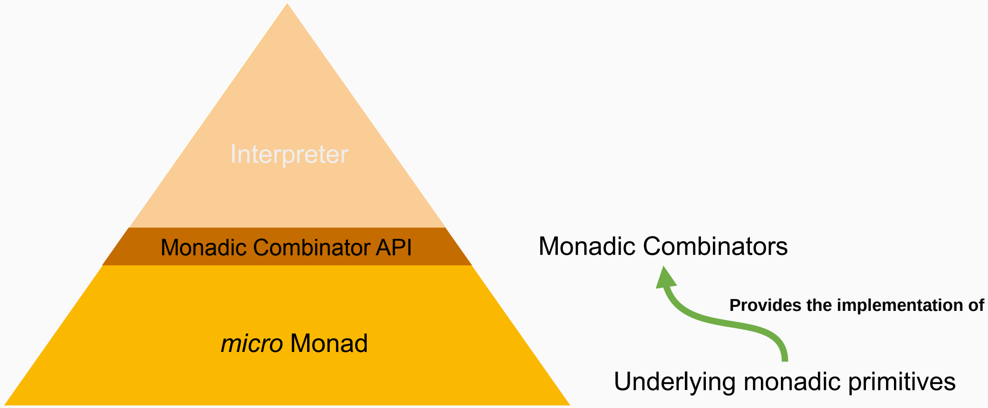
```
Fixpoint eval η e :=
  | ...
  | EPerform e =>
      eff <- eval η e ;
      perform eff
  | EMatch e bs =>
      handle (eval η e) (fun o => eval_branches η o bs)
```

$\boxed{\text{O3Ret v} \mid \text{O3Throw exn} \mid \text{O3Perform k v}}$

```
perform e

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

continue k e

discontinue k e
```

```
Fixpoint eval η e :=
  | ...
  | EPerform e =>
    eff <- eval η e ;
    perform eff
  | EMatch e bs =>
    handle (eval η e) (fun o => eval_branches η o bs)
  | 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)
```


Interpreter
Monadic Combinator API
micro Monad
Monadic Definitional Interpreter
Relies on effects provided by
Monadic Combinators


Interpreter
Monadic Combinator API
micro Monad
Monadic Combinators
Provides the implementation of
Underlying monadic primitives

A Syntax for Computations

$$\begin{aligned} please\_eval\ \eta\ e &:= \\ perform\ v &:= \\ resume\ k\ v &:= \end{aligned}$$

A Syntax for Computations

$$\begin{aligned} please\_eval\ \eta\ e &:= \textcolor{orange}{Stop}\ CEval\,(\eta, e)\ id_2 \\ perform\ v &:= \textcolor{orange}{Stop}\ CPerf\ v\ id_2 \\ resume\ k\ v &:= \textcolor{orange}{Stop}\ CResume\,(k, v)\ id_2 \end{aligned}$$

A Syntax for Computations

$$
\begin{aligned}
\mathit{please\_eval}\ \eta\ e &:= \mathit{Stop}\ \mathit{CEval}\,(\eta, e)\ \mathit{id}_2 \\
\mathit{perform}\ v &:= \mathit{Stop}\ \mathit{CPerf}\ v\ \mathit{id}_2 \\
\mathit{resume}\ k\ v &:= \mathit{Stop}\ \mathit{CResume}\,(k, v)\ \mathit{id}_2
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&\mathit{id}_2 : \mathit{outcome}_2 \to \mathit{micro}\ \mathit{val}\ \mathit{exn} \\
&\mathit{id}_2\ (\mathit{O2Ret}\ v) = \mathit{Ret}\ v \\
&\mathit{id}_2\ (\mathit{O2Throw}\ \mathit{ex}) = \mathit{Throw}\ \mathit{ex}
\end{aligned}
}
$$

A Syntax for Computations

$$
\begin{aligned}
\mathit{please\_eval}\ \eta\ e &:= \mathit{Stop}\ \mathit{CEval}\ (\eta, e)\ \mathit{id}_2 \\
\mathit{perform}\ v &:= \mathit{Stop}\ \mathit{CPerf}\ v\ \mathit{id}_2 \\
\mathit{resume}\ k\ v &:= \mathit{Stop}\ \mathit{CResume}\ (k, v)\ \mathit{id}_2
\end{aligned}
$$

- *Ret* is a leaf;
- A *Stop* node represents a "system call" and carries one child for each possible result;

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

A Syntax for Computations

$$\begin{aligned} please\_eval\ \eta\ e &:= Stop\ CEval\,(\eta, e)\ id_2 \\ perform\ v &:= Stop\ CPerf\ v\ id_2 \\ resume\ k\ v &:= Stop\ CResume\,(k, v)\ id_2 \end{aligned}$$

$$\begin{aligned} par\ m_1\ m_2 &:= Par\ m_1\ m_2\ id_2 \\ handle\ m\ h &:= Handle\ m\ h \end{aligned}$$

- *Ret* is a leaf;
- A *Stop* node represents a “system call” and carries one child for each possible result;

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

A Syntax for Computations

$$\begin{aligned} \mathit{please\_eval}\ \eta\ e &:= \mathit{Stop}\ \mathit{CEval}\ (\eta, e)\ \mathit{id}_2 \\ \mathit{perform}\ v &:= \mathit{Stop}\ \mathit{CPerf}\ v\ \mathit{id}_2 \\ \mathit{resume}\ k\ v &:= \mathit{Stop}\ \mathit{CResume}\ (k, v)\ \mathit{id}_2 \end{aligned}$$

$$\begin{aligned} \mathit{par}\ m_1\ m_2 &:= \mathit{Par}\ m_1\ m_2\ \mathit{id}_2 \\ \mathit{handle}\ m\ h &:= \mathit{Handle}\ m\ h \end{aligned}$$

- *Ret* is a leaf;
- 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

$$please\_eval\ \eta\ e := Stop\ CEval\ (\eta, e)\ id_2$$

$$perform\ v := Stop\ CPerf\ v\ id_2$$

$$resume\ k\ v := Stop\ CResume\ (k, v)\ id_2$$

$$par\ m_1\ m_2 := Par\ m_1\ m_2\ id_2$$

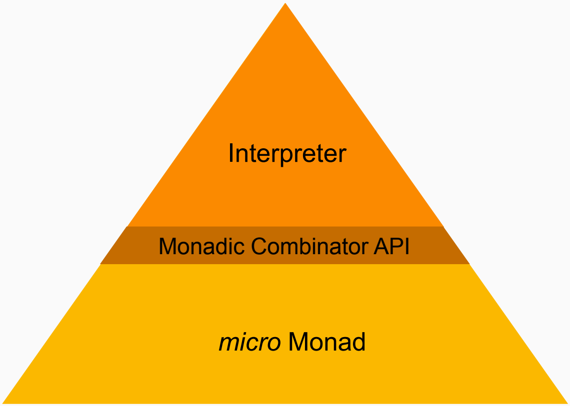
$$handle\ m\ h := Handle\ m\ h$$

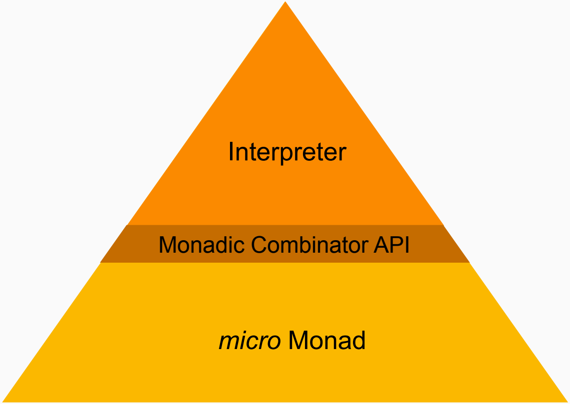
- *Ret*, *Throw*, *Crash* are leaves;
- A *Stop* node represents a "system call" and carries one child for each possible result;
- A *Par* node allows parallel computation;
- A *Handle* node serves as a delimiter of control effects and carries an effect handler.

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

A Syntax for Computations

$$
\begin{aligned}
&\textsf{Inductive}\ \mathit{micro}\ A\ E :=\\
&\quad \mid \textbf{Ret} &&: A \to \mathit{micro}\ A\ E\\
&\quad \mid \textbf{Throw} &&: E \to \mathit{micro}\ A\ E\\
&\quad \mid \textbf{Crash} &&: \mathit{micro}\ A\ E\\
&\quad \mid \textbf{Stop} &&: \mathit{code}\ X\ Y\ E' \to X \to (\mathit{outcome}_2\ Y\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E\\
&\quad \mid \textbf{Par} &&: \mathit{micro}\ A_1\ E' \to \mathit{micro}\ A_2\ E' \to\\
& && \qquad (\mathit{outcome}_2\ (A_1 \times A_2)\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E\\
&\quad \mid \textbf{Handle} &&: \mathit{micro}\ \mathit{val}\ \mathit{exn} \to (\mathit{outcome}_3\ \mathit{val}\ \mathit{exn} \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E
\end{aligned}
$$


Interpreter
Monadic Combinator API
micro Monad



+

$\longrightarrow$

A Small-Step Reduction Semantics

The meaning, or *behaviour*, of an effect is given by a *small-step semantics*.

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

A Small-Step Reduction Semantics

The meaning, or *behaviour*, of an effect is given by a *small-step semantics*.

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

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

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

This technique is inspired by *Turing Completeness Completely Free*, McBride (2015).

A Small-Step Reduction Semantics

The meaning, or *behaviour*, of an effect is given by a *small-step semantics*.

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

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

$$please\_eval \ \eta \ e \ / \ \sigma \longrightarrow eval \ \eta \ e \ / \ \sigma$$

This technique is inspired by *Turing Completeness Completely Free*, McBride (2015).

Delimited Control

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

$$\frac{m \;/\; \sigma \longrightarrow m' \;/\; \sigma'}{handle\; m\; h \;/\; \sigma \longrightarrow handle\; m'\; h \;/\; \sigma'}$$

Delimited Control

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

$$\frac{m \;/\; \sigma \longrightarrow m' \;/\; \sigma'}{handle\ m\ h \;/\; \sigma \longrightarrow handle\ m'\ h \;/\; \sigma'}$$

$$\begin{array}{rcl}
handle\,(Ret\ \ v)\ h\,/\,\sigma & \longrightarrow & h\ (O3Ret\ \ v)\,/\,\sigma \\
handle\,(Throw\ \ ex)\ h\,/\,\sigma & \longrightarrow & h\ (O3Throw\ \ ex)\,/\,\sigma \\
handle\,(Stop\ CPerf\ \ v\ k)\ h\,/\,\sigma & \longrightarrow & h\ (Perform_3\ \ v\ \ell)\,/\,\sigma[\ell := k] \text{ with } \ell \text{ fresh}
\end{array}$$

Delimited Control

Handle observes a computation's *outcome*$_3$ and invokes a handler.

$$\frac{m \;/\; \sigma \longrightarrow m' \;/\; \sigma'}{\mathit{handle}\ m\ h \;/\; \sigma \longrightarrow \mathit{handle}\ m'\ h \;/\; \sigma'}$$

$$\begin{array}{rcl}
\mathit{handle}\,(\mathit{Ret}\ v)\ h \,/\, \sigma & \longrightarrow & h\,(\mathit{O3Ret}\ v) \,/\, \sigma \\
\mathit{handle}\,(\mathit{Throw}\ \mathit{ex})\ h \,/\, \sigma & \longrightarrow & h\,(\mathit{O3Throw}\ \mathit{ex}) \,/\, \sigma \\
\mathit{handle}\,(\mathit{Stop}\ \mathit{CPerf}\ v\ k)\ h \,/\, \sigma & \longrightarrow & h\,(\mathit{Perform}_3\ v\ \ell) \,/\, \sigma[\ell := k] \text{ with } \ell \text{ fresh}
\end{array}$$

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

Delimited Control

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

$$\frac{\sigma(\ell) = k}{\mathit{resume}\ \ell\ o\ /\ \sigma \longrightarrow k\ o\ /\ \sigma[\ell := \lightning]}$$

Delimited Control

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

$$\frac{\sigma(\ell) = k}{\mathit{resume}\ \ell\ o \ /\ \sigma \longrightarrow k\ o \ /\ \sigma[\ell := \lightning]}$$

$$\frac{\sigma(\ell) = \lightning}{\mathit{resume}\ \ell\ o \ /\ \sigma \longrightarrow \mathit{Crash} \ /\ \sigma}$$

ROCQ

Automatically generate

- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

Iris

- proofs
  - alpha_renaming.v
  - beta_red.v
  - environment.v
  - inductive.v
  - instructionHandler.v
  - substitution.v
  - typing.v

\+ ⟶

ROCQ

Automatically generate

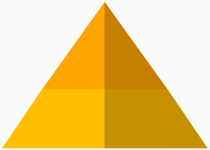
- src
  - alpha_renaming.ml
  - base_tactics.ml
  - beta_red.ml
  - environment.ml
  - inductive.ml
  - instructionHandler.ml
  - substitution.ml
  - typing.ml
  - dune

- code
  - og_alpha_renaming.v
  - og_beta_red.v
  - og_environment.v
  - og_inductive.v
  - og_instructionHandler.v
  - og_substitution.v
  - og_typing.v

+ $\longrightarrow$

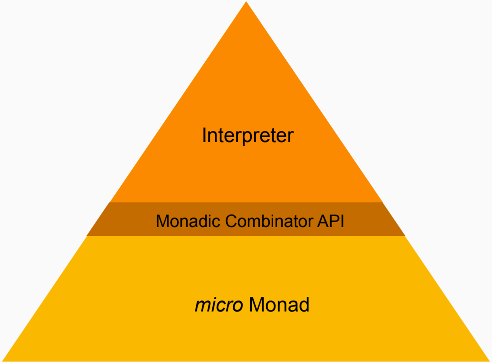
Iris

- proofs
  - alpha_renaming.v
  - beta_red.v
  - environment.v
  - inductive.v
  - instructionHandler.v
  - substitution.v
  - typing.v



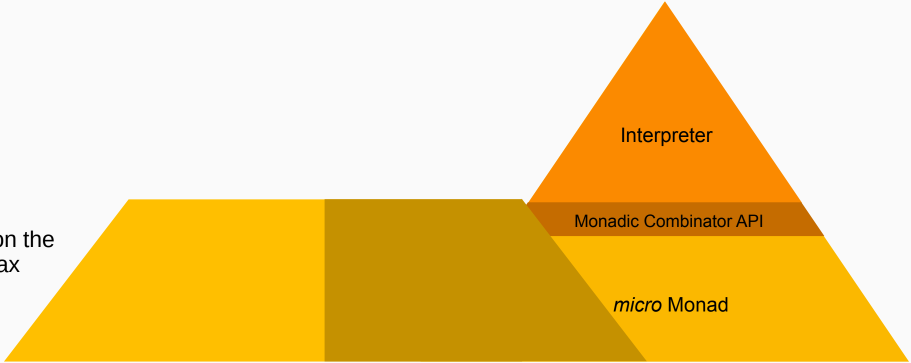
Building our Program Logics

$\longrightarrow$ **+**

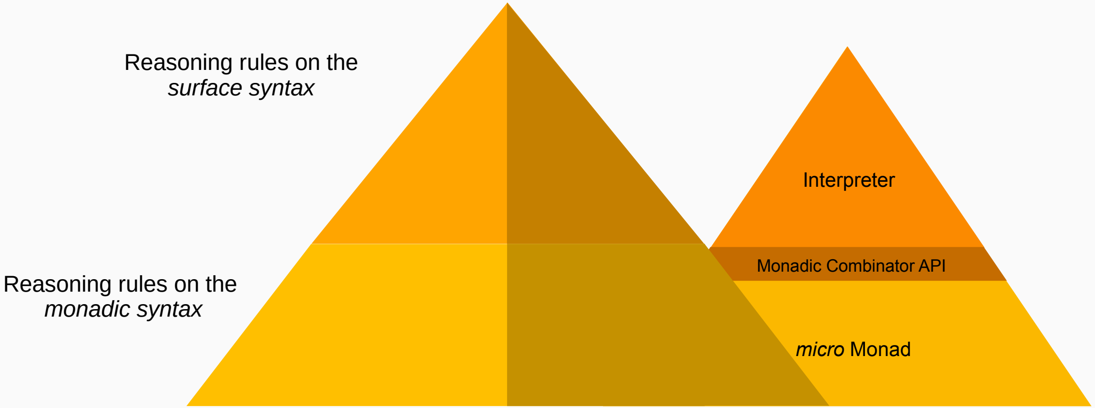


Building our Program Logics

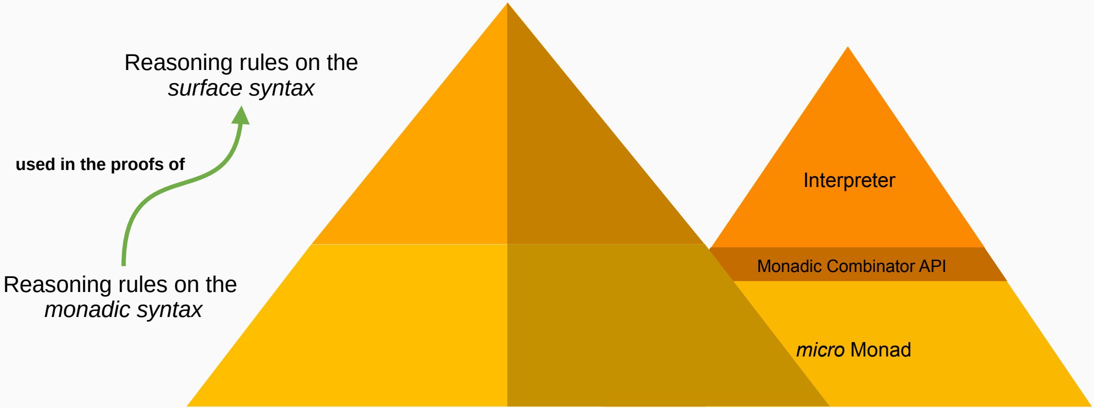
Reasoning rules on the
monadic syntax



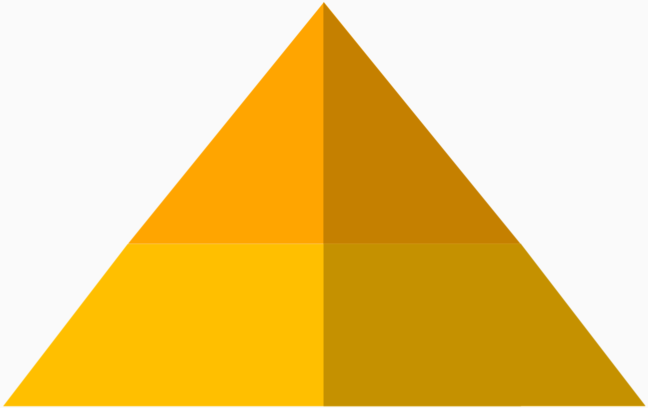
Building our Program Logics



Building our Program Logics



Horus: Hoare-Style Reasoning About Pure Programs



Horus: Hoare-Style Reasoning About Pure Programs

pure m φ ζ

Horus: Hoare-Style Reasoning About Pure Programs

$$\boxed{\begin{array}{l} \varphi : \mathsf{val} \to \mathsf{Prop} \\ \zeta : \mathsf{exn} \to \mathsf{Prop} \end{array}}$$

$$\mathit{pure}\ m\ \varphi\ \zeta$$

Horus: Hoare-Style Reasoning About Pure Programs

$$\boxed{\begin{aligned}\varphi &: \mathsf{val} \to \mathsf{Prop}\\ \zeta &: \mathsf{exn} \to \mathsf{Prop}\end{aligned}}$$

$$\mathit{pure}\ m\ \varphi\ \zeta$$

$$\mathit{pure}\ (\mathit{please\_eval}\ \eta\ e)\ \varphi\ \zeta$$

Horus: Hoare-Style Reasoning About Pure Programs

$$\boxed{\begin{array}{l}\varphi \ : \ \mathsf{val} \to \mathsf{Prop} \\ \zeta \ : \ \mathsf{exn} \to \mathsf{Prop}\end{array}}$$

$$\mathit{pure}\ m\ \varphi\ \zeta$$

$$\mathit{pure}\ (\mathit{eval}\ \eta\ e)\ \varphi\ \zeta$$

$$\mathit{pure}\ (\mathit{eval\_branches}\ \eta\ o\ \mathit{bs})\ \varphi\ \zeta$$

$$\mathit{pure}\ (\mathit{please\_eval}\ \eta\ e)\ \varphi\ \zeta$$

Horus: Hoare-Style Reasoning About Pure Programs

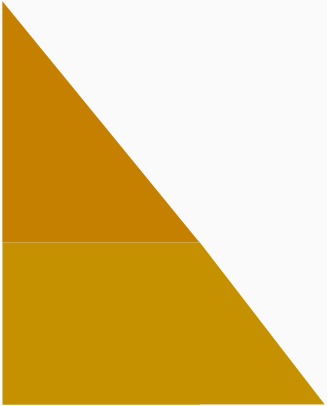
$$\mathit{expr}\ \eta\ e\ \varphi\ \zeta$$

$$\mathit{branches}\ \eta\ o\ \mathit{bs}\ \varphi\ \zeta$$

$$\mathit{pure}\ m\ \varphi\ \zeta$$

$$\mathit{pure}\ (\mathit{please\_eval}\ \eta\ e)\ \varphi\ \zeta$$

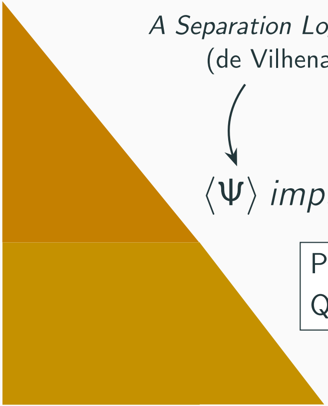
Iris-Style Reasoning About Arbitrary Programs



Iris-Style Reasoning About Arbitrary Programs

$\langle \Psi \rangle$ *impure m P Q*

Iris-Style Reasoning About Arbitrary Programs



A Separation Logic for Effect Handlers,
(de Vilhena & Pottier, 2021)

$$\langle \Psi \rangle \; \mathit{impure} \; m \; P \; Q$$

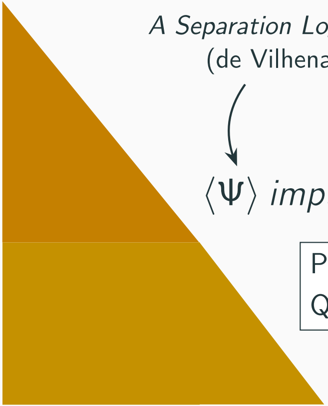
$$\boxed{\begin{array}{l} \mathsf{P} : \mathsf{val} \to \mathsf{iProp} \\ \mathsf{Q} : \mathsf{exn} \to \mathsf{iProp} \end{array}}$$

Iris-Style Reasoning About Arbitrary Programs

A Separation Logic for Effect Handlers, (de Vilhena & Pottier, 2021)

$\langle\Psi\rangle$ *impure* m P Q

$$\boxed{\begin{array}{l} \mathsf{P} : \mathsf{val} \to \mathsf{iProp} \\ \mathsf{Q} : \mathsf{exn} \to \mathsf{iProp} \end{array}}$$



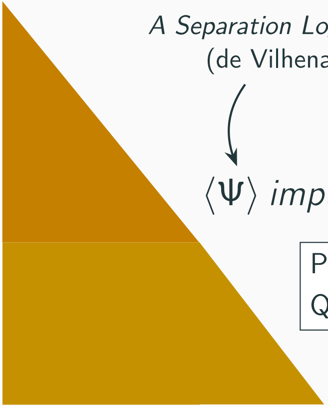
$\langle\Psi\rangle$ *impure* $(handle\ m\ h)$ P Q

Iris-Style Reasoning About Arbitrary Programs

$\langle \Psi \rangle$ *impure* $(\mathit{eval}\ \eta\ e)\ P\ Q$

deep-handler $\Psi\ P\ Q\ \eta\ o\ \mathit{bs}\ P'\ Q'$

$\langle \Psi \rangle$ *impure* $(\mathit{handle}\ m\ h)\ P\ Q$



A Separation Logic for Effect Handlers, (de Vilhena & Pottier, 2021)

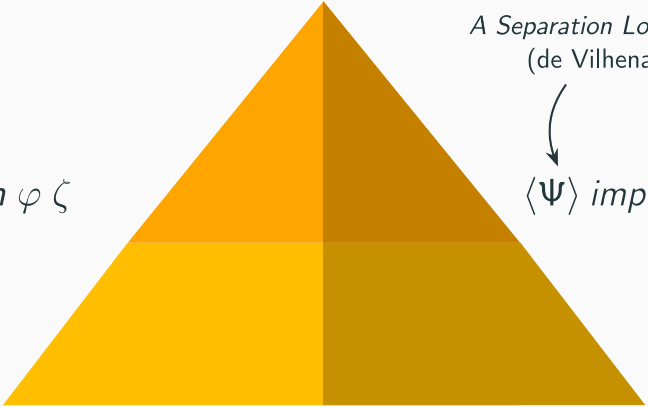
$\langle \Psi \rangle$ *impure* $m\ P\ Q$

$\boxed{\begin{array}{l} \mathsf{P} : \mathsf{val} \rightarrow \mathsf{iProp} \\ \mathsf{Q} : \mathsf{exn} \rightarrow \mathsf{iProp} \end{array}}$

A Separation Logic for Effect Handlers, (de Vilhena & Pottier, 2021)

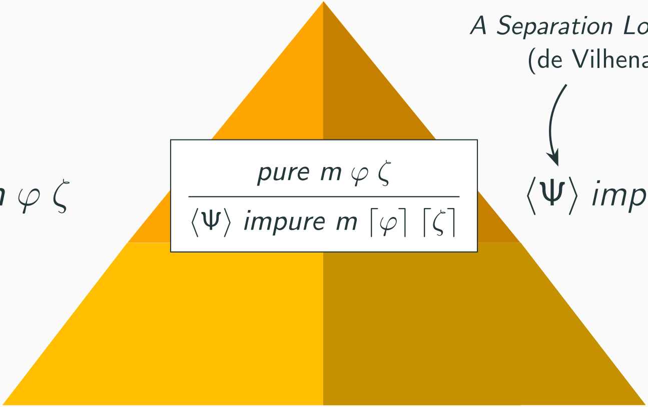
$\langle \Psi \rangle$ *impure* $m\ P\ Q$

pure $m\ \varphi\ \zeta$



Iris-Style Reasoning About Arbitrary Programs

$\mathit{pure}\ m\ \varphi\ \zeta$

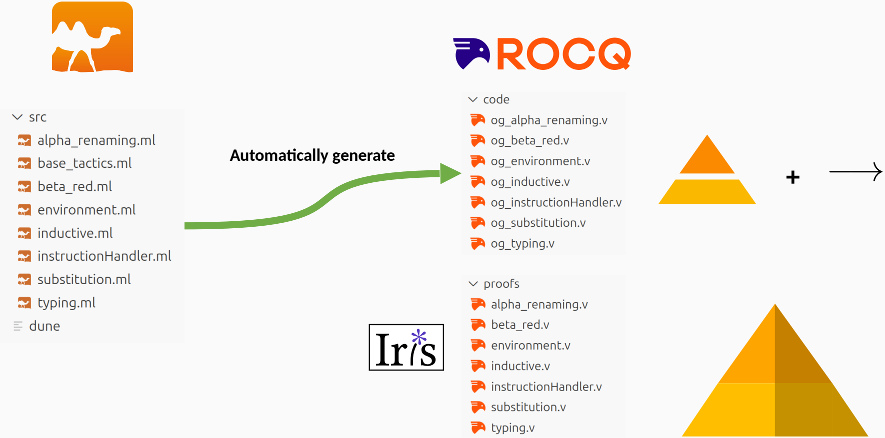


$$\frac{\mathit{pure}\ m\ \varphi\ \zeta}{\langle\Psi\rangle\ \mathit{impure}\ m\ \lceil\varphi\rceil\ \lceil\zeta\rceil}$$

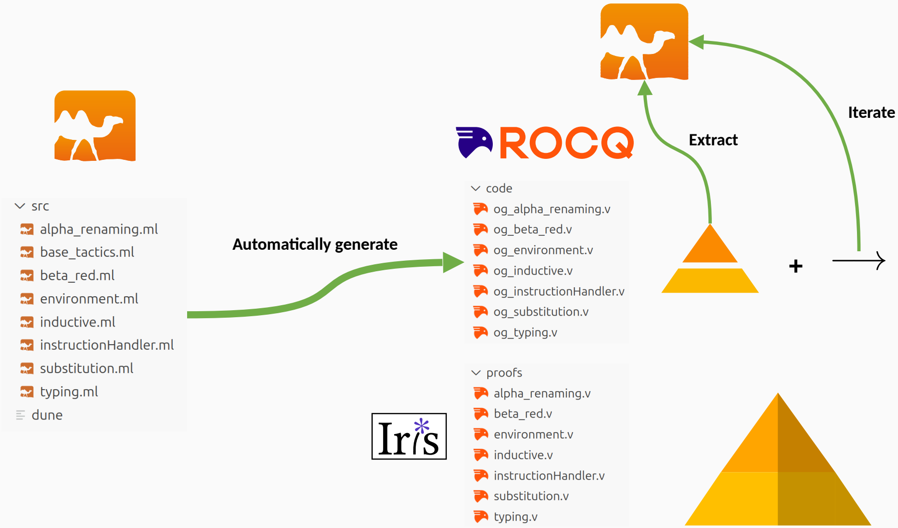
A Separation Logic for Effect Handlers,
(de Vilhena & Pottier, 2021)

$\langle\Psi\rangle\ \mathit{impure}\ m\ P\ Q$

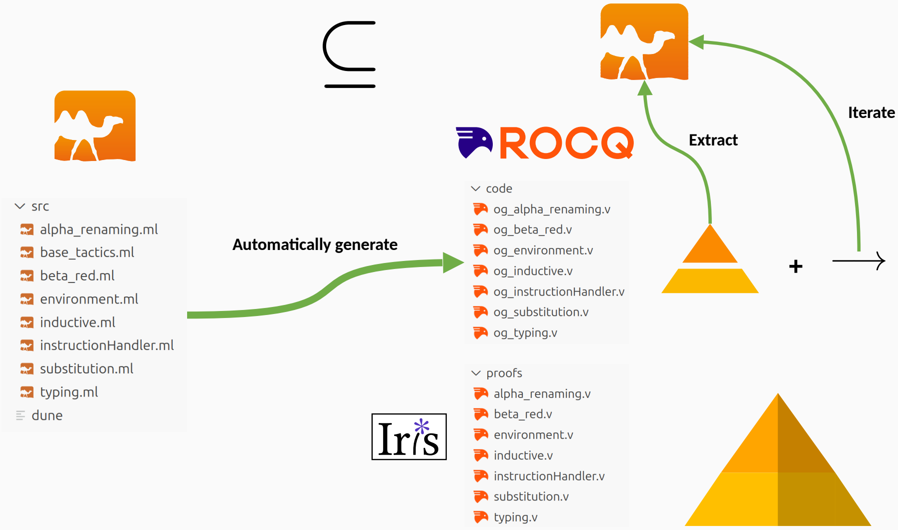
Validating the Semantics

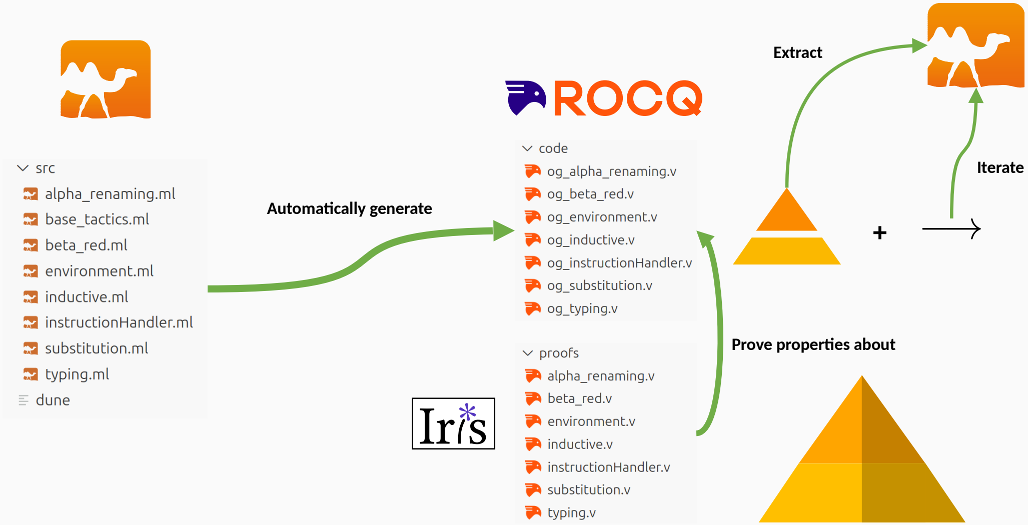


Validating the Semantics

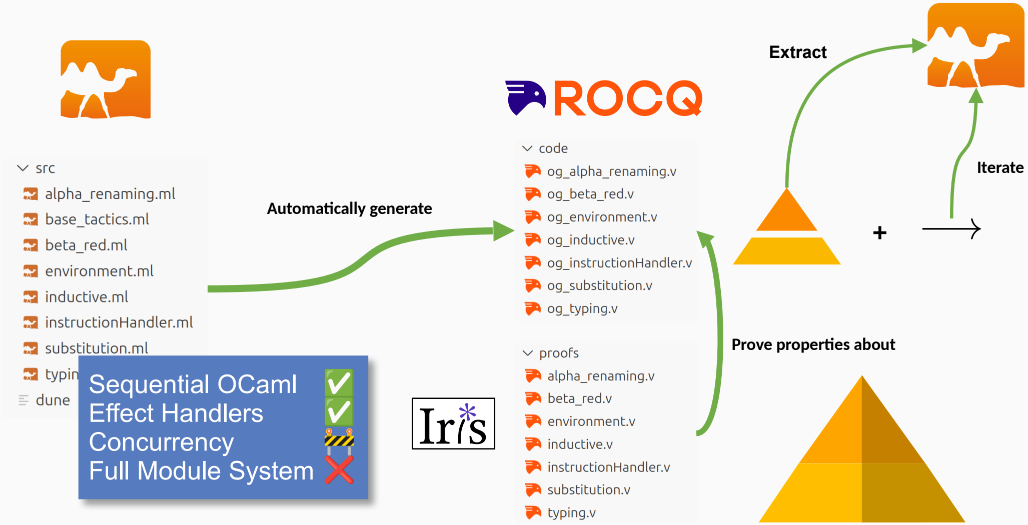


Validating the Semantics





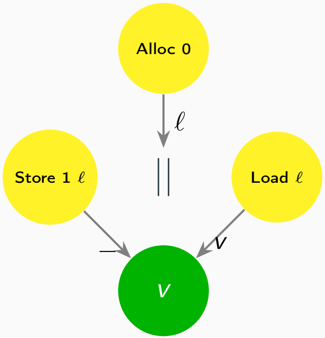
src
alpha_renaming.ml
base_tactics.ml
beta_red.ml
environment.ml
inductive.ml
instructionHandler.ml
substitution.ml
typing.ml
dune
Automatically generate
ROCQ
code
og_alpha_renaming.v
og_beta_red.v
og_environment.v
og_inductive.v
og_instructionHandler.v
og_substitution.v
og_typing.v
proofs
alpha_renaming.v
beta_red.v
environment.v
inductive.v
instructionHandler.v
substitution.v
typing.v
Iris
Extract
Iterate
+
Prove properties about


src
alpha_renaming.ml
base_tactics.ml
beta_red.ml
environment.ml
inductive.ml
instructionHandler.ml
substitution.ml
typin
dune
Sequential OCaml ✅
Effect Handlers ✅
Concurrency 🚧
Full Module System ❌
Automatically generate
Iris
ROCQ
code
og_alpha_renaming.v
og_beta_red.v
og_environment.v
og_inductive.v
og_instructionHandler.v
og_substitution.v
og_typing.v
proofs
alpha_renaming.v
beta_red.v
environment.v
inductive.v
instructionHandler.v
substitution.v
typing.v
Extract
Iterate
+
Prove properties about

Backup Slides

Nondeterminism / Parallelism

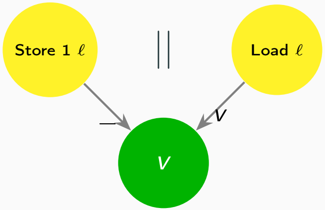
```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```



Nondeterminism / Parallelism

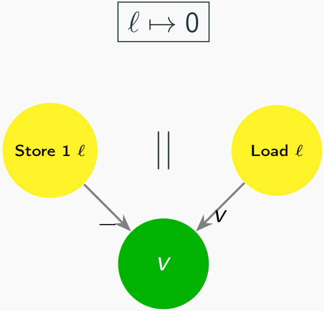
```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

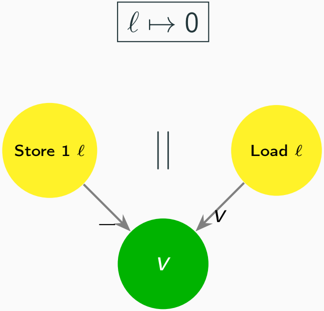
$\boxed{\ell \mapsto 0}$



Nondeterminism / Parallelism

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```



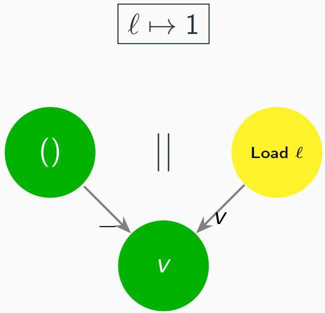


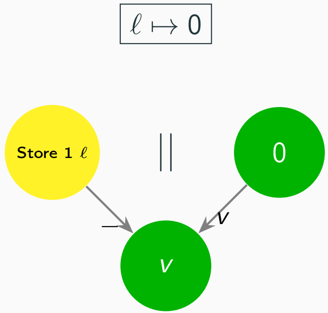
Nondeterminism / Parallelism

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

Nondeterminism / Parallelism

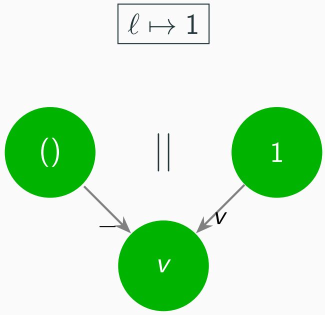
```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

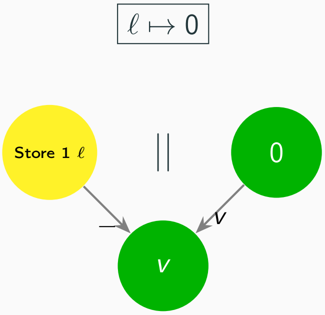




Nondeterminism / Parallelism

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```





Nondeterminism / Parallelism

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

Nondeterminism / Parallelism

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

$\boxed{\ell \mapsto 1}$ $\boxed{\ell \mapsto 1}$

1 0

```
let x = ref 0 in
let ((), v) = ((x := 1), !x) in
v
```

$\boxed{\ell \mapsto 1}$

1

```
let x = ref 0 in
let (_, v) = ((x := 1), !x) in
v
```

$\boxed{\ell \mapsto 1}$

0

Shallow Effect Handlers

Supporting shallow effect handlers only requires removing the instance of *wrap* in *eval_branches*.

```
Fixpoint eval η e :=
  ...
  | EShallowMatch e bs ⇒
      handle (eval η e) (λ o ⇒ shallow_eval_branches η o bs)
```

Shallow matches have a different set of proof rules, but they are built atop of the same *EWP*.

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

$$\begin{aligned} \mathit{Stop}\ \mathit{CWrap}\,(\eta, \ell, \mathit{bs})\ /\ \sigma \quad \longrightarrow \quad & \mathit{O2Ret}\ \ell'\ /\ \sigma[\ell' := k'] \\ & \text{if } \ell' \notin \mathit{dom}(\sigma) \end{aligned}$$

Delimited Control

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

$$\begin{array}{lll} \mathit{Stop}\ \mathit{CWrap}\,(\eta, \ell, \mathit{bs})\ /\ \sigma & \longrightarrow & \mathit{O2Ret}\ \ell'\ /\ \sigma[\ell' := k'] \\ & & \quad \text{if } \ell' \notin \mathit{dom}(\sigma) \end{array}$$

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