Melocoton: A Program Logic for Verified Interoperability Between OCaml and C

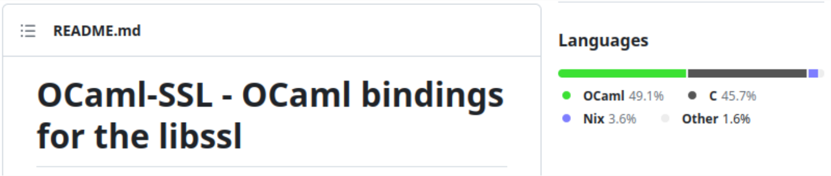
Armaël Guéneau   Johannes Hostert   Simon Spies   Michael Sammler
Lars Birkedal   Derek Dreyer

May 24, 2023

Many Real Programs Are Multi-Language

Consider the *ocaml-ssl* library:

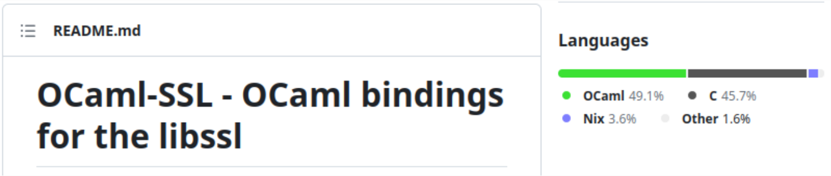
- Exposes `OpenSSL` (a C library) as an OCaml library
- To do so, it is implemented using a mix of *both* OCaml *and* C code:



Many Real Programs Are Multi-Language

Consider the *ocaml-ssl* library:

- Exposes `OpenSSL` (a C library) as an OCaml library
- To do so, it is implemented using a mix of *both* OCaml *and* C code:



How do we reason about such code (in Iris)?

Mind the gap!

OCaml **C**

Mind the gap!

OCaml

Structured values

$\boxed{\lambda_{\mathrm{ML}}}$ $V \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \mathit{Loc})$

$\mid \mathsf{true} \mid \mathsf{false}$

$\mid \langle\rangle \mid \langle V, V \rangle \cdots$

Garbage collection

$\boxed{\textbf{Iris}_{\mathrm{ML}}}$ $\ell \mapsto_{\mathrm{ML}} \vec{V}$

C

Integers and pointers

$\boxed{\lambda_{\mathrm{C}}}$ $w \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (a \in \mathit{Addr})$

Manual memory management

$\boxed{\textbf{Iris}_{\mathrm{C}}}$ $a \mapsto_{\mathrm{C}} w$

Mind the gap!

OCaml $\longleftarrow$ **OCaml FFI** $\longrightarrow$ **C**

Structured values

$\boxed{\lambda_{\text{ML}}}$ $V \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \mathit{Loc})$
$\mid \mathsf{true} \mid \mathsf{false}$
$\mid \langle\rangle \mid \langle V, V \rangle \cdots$

Integers and pointers

$\boxed{\lambda_{\text{C}}}$ $w \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (a \in \mathit{Addr})$

Garbage collection

$\boxed{\textbf{Iris}_{\text{ML}}}$ $\ell \mapsto_{\text{ML}} \vec{V}$

Manual memory management

$\boxed{\textbf{Iris}_{\text{C}}}$ $a \mapsto_{\text{C}} w$

Key Challenge

Can we build a program logic for reasoning about interoperability with an FFI, **while preserving language-local reasoning**?

λ_{ML} Semantics

Iris$_{\mathrm{ML}}$
Program Logic

given as black box

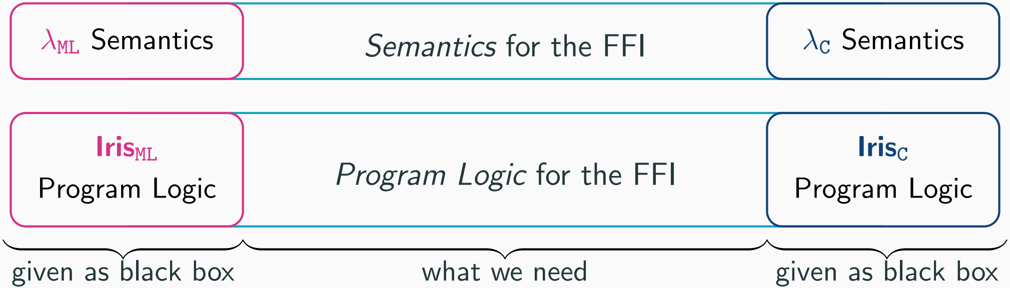
λ_{C} Semantics

Iris$_{\mathrm{C}}$
Program Logic

given as black box

Key Challenge

Can we build a program logic for reasoning about interoperability with an FFI, **while preserving language-local reasoning**?



Design choice: **reuse** most of existing semantics/program logics;
do not drop down to a lowest-common denominator (assembly)!

Contributions

Melocoton:

- Two instantiations of Iris for a ML-like and C-like language with *external calls*
- An *operational semantics* for the OCaml FFI, bridging between the two languages.
- A *separation logic* for the OCaml FFI, bridging between the two language logics.
- A number of interesting *case studies*

Contributions

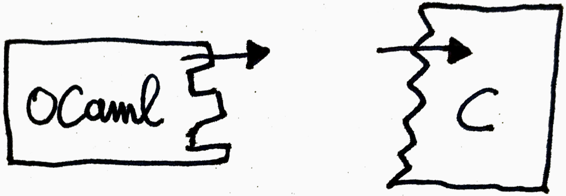
Melocoton:

- Two instantiations of Iris for a ML-like and C-like language with *external calls*
- An *operational semantics* for the OCaml FFI, bridging between the two languages.
- A *separation logic* for the OCaml FFI, bridging between the two language logics.
- A number of interesting *case studies*

Language-locality: Verification of mixed OCaml/C programs can be done *almost entirely* in logics for OCaml and C!

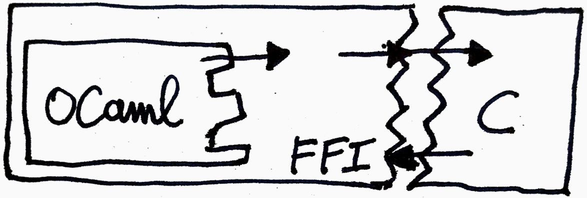
Outline

1. Language-local program logics with external calls



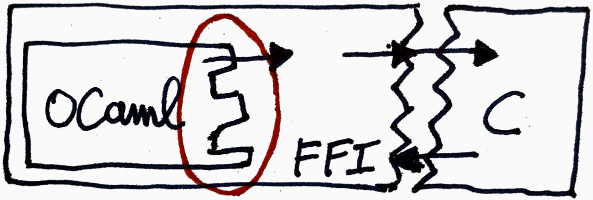
Outline

1. Language-local program logics with external calls
2. Program logic for FFI



Outline

1. Language-local program logics with external calls
2. Program logic for FFI
3. Focus: the language boundary



Example: updating an OCaml reference from C code

OCaml code:

```
let main () =
  let r = ref 0 in
  update_ref r; (* TODO call C code and use rand() *)
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Glue code:

```
value caml_update_ref(value r) {
  /* TODO */
  int y = rand(x);
  /* TODO */
}
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

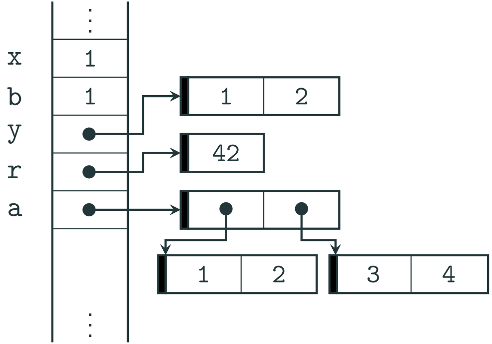
Glue code:

```
value caml_update_ref(value r) {
  /* TODO */
  int y = rand(x);
  /* TODO */
}
```

The runtime representation of OCaml values

At runtime, an OCaml value is either an integer or a pointer to a block:

```
let x = 1
let b = true
let y = (1, 2)
let r = ref 42
let a = [| (1, 2); (3, 4) |]
```



Glue code has access to this *low-level* representation of OCaml values.

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Glue code:

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

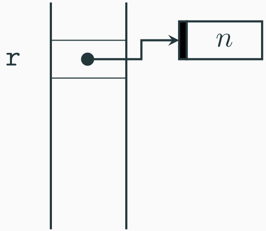
Glue code bridges between OCaml and C values by using powerful **FFI primitives**...

Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));     /* read the first field of the input block */
  int y = rand(x);                  /* get a random integer */
  Store_field(r, 0, Val_int(y));    /* store the value in the block */
  return Val_int(0);                /* return () */
}
```

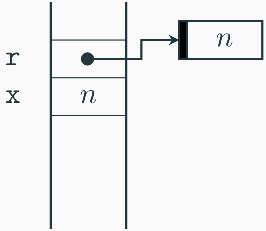
Writing glue code

```
value caml_update_ref(value r) {<--
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);                   /* get a random integer */
  Store_field(r, 0, Val_int(y));     /* store the value in the block */
  return Val_int(0);                 /* return () */
}
```



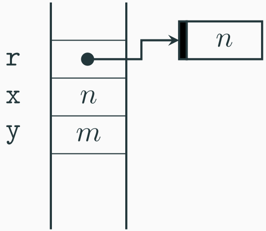
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0)); <-- /* read the first field of the input block */
  int y = rand(x);                  /* get a random integer */
  Store_field(r, 0, Val_int(y));    /* store the value in the block */
  return Val_int(0);                /* return () */
}
```



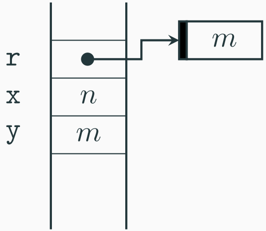
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);                <-- /* get a random integer */
  Store_field(r, 0, Val_int(y));     /* store the value in the block */
  return Val_int(0);                 /* return () */
}
```



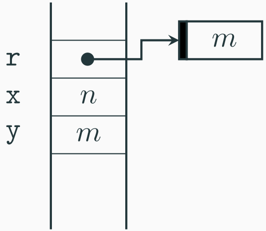
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));       /* read the first field of the input block */
  int y = rand(x);                    /* get a random integer */
  Store_field(r, 0, Val_int(y));<-- /* store the value in the block */
  return Val_int(0);                  /* return () */
}
```



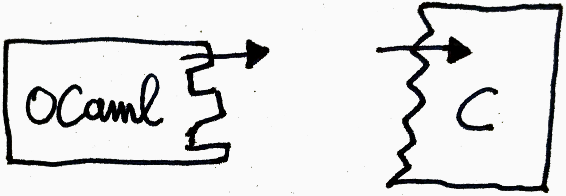
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);                   /* get a random integer */
  Store_field(r, 0, Val_int(y));     /* store the value in the block */
  return Val_int(0);             <-- /* return () */
}
```



Outline: Language-local reasoning

1. Language-local program logics with external calls



Language-local reasoning

We reuse:

$\lambda_{\texttt{ML}}$ Semantics

$\lambda_{\texttt{C}}$ Semantics

Iris$_{\texttt{ML}}$
Program Logic

Iris$_{\texttt{C}}$
Program Logic

The one change: a minimal extension allowing **external calls**.

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

We model external calls as a new syntactic construct (inlining the declaration):

$$e \in \mathit{Expr} ::= \cdots \mid \mathsf{call}\ \mathit{fn}\ \vec{e}$$

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

We model external calls as a new syntactic construct (inlining the declaration):

$$e \in \mathit{Expr} ::= \cdots \mid \textsf{call}\ \mathit{fn}\ \vec{e}$$

We assign **no semantics** to external calls: they are simply stuck!

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \textsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \textsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

To do so, we introduce **interfaces** Ψ, and weakest preconditions $\textsf{wp}\ e\ @\ \Psi\ \{v.\, Q\}$ that verify programs against them. For example, for `caml_update_ref`, we assume:

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n\rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\rangle \quad \sqsubseteq \Psi$$

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \textsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

To do so, we introduce **interfaces** Ψ, and weakest preconditions $\textsf{wp}\ e\ @\ \Psi\ \{v.\, Q\}$ that verify programs against them. For example, for `caml_update_ref`, we assume:

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n\rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\rangle \quad \sqsubseteq \Psi$$

⚠ **This is an assumption, not a (atomic) Hoare triple** ⚠

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

$$\Psi : \underbrace{FnName}_{\text{Name}} \rightarrow \underbrace{\vec{Val}}_{\text{Args}} \rightarrow \underbrace{(Val \rightarrow iProp)}_{\text{Postcondition}} \rightarrow \underbrace{iProp}_{\text{Precondition}}$$

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

$$\Psi : \underbrace{FnName}_{\text{Name}} \to \underbrace{\vec{Val}}_{\text{Args}} \to \underbrace{(Val \to \mathit{iProp})}_{\text{Postcondition}} \to \underbrace{\mathit{iProp}}_{\text{Precondition}}$$

We desugar

$$\forall \ell\, n.\ \langle \ell \mapsto_{\text{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]$$
$$\langle V'.\ \exists m.\ V' = \langle\rangle * \ell \mapsto_{\text{ML}} m \rangle$$

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

$$\Psi : \underbrace{\mathit{FnName}}_{\text{Name}} \rightarrow \underbrace{\vec{\mathit{Val}}}_{\text{Args}} \rightarrow \underbrace{(\mathit{Val} \rightarrow \mathit{iProp})}_{\text{Postcondition}} \rightarrow \underbrace{\mathit{iProp}}_{\text{Precondition}}$$

We desugar

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]$$
$$\langle V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \rangle$$

as follows:

$$\Psi_{\mathit{upd}}\ \mathit{fn}\ \vec{V}\ \Phi := \exists \ell n.\, \ell \mapsto_{\mathrm{ML}} n * \mathit{fn} = \texttt{caml\_update\_ref} * \vec{V} = [\ell]$$
$$* (\forall V' m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \twoheadrightarrow \Phi(V'))$$

Implementing Interface Triples

$$\Psi : \underbrace{FnName}_{\text{Name}} \to \underbrace{\vec{Val}}_{\text{Args}} \to \underbrace{(Val \to iProp)}_{\text{Postcondition}} \to \underbrace{iProp}_{\text{Precondition}}$$

Parameterize weakest pre by Ψ (inspired by de Vilhena and Pottier [2021]):

$$\text{wp } e \mathbin{@} \Psi \, \{\Phi\} := \begin{cases} \Phi(v) & e = v \\ \forall e', (e \to e') \Rightarrow \text{wp } e' \mathbin{@} \Psi \, \{\Phi\} & e \text{ reducible} \\ \Psi \; fn \; \vec{V} \; \underbrace{\left(\lambda V'. \, \text{wp } K[V'] \mathbin{@} \Psi \, \{\Phi\}\right)}_{\text{Postcondition}} & e = K[\text{call } fn \; \vec{V}] \end{cases}$$

Implementing Interface Triples

$$\Psi : \underbrace{FnName}_{\text{Name}} \to \underbrace{\vec{Val}}_{\text{Args}} \to \underbrace{(Val \to iProp)}_{\text{Postcondition}} \to \underbrace{iProp}_{\text{Precondition}}$$

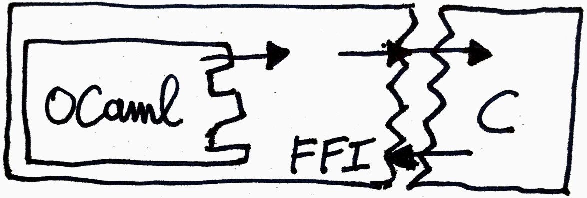
Parameterize weakest pre by Ψ (inspired by de Vilhena and Pottier [2021]):

$$\text{wp}\ e \mathbin{@} \Psi\ \{\Phi\} := \begin{cases} \Phi(v) & e = v \\ \forall e', (e \to e') \Rightarrow \text{wp}\ e' \mathbin{@} \Psi\ \{\Phi\} & e \text{ reducible} \\ \Psi\ fn\ \vec{V}\ \underbrace{\left(\lambda V'.\, \text{wp}\ K[V'] \mathbin{@} \Psi\ \{\Phi\}\right)}_{\text{Postcondition}} & e = K[\text{call}\ fn\ \vec{V}] \end{cases}$$

Note: In a OCaml-and-C program (after linking), adequacy holds for $\Psi\ fn\ \vec{V}\ \Phi := \bot$

Outline: The OCaml FFI

1. Language-local program logics with external calls
2. Glue code and program logic for FFI



External Calls in Glue Code

In glue code we treat operations of the OCaml FFI as **external functions**.

```
value caml_update_ref(value r) {
  int x = Int_val (Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

External Calls in Glue Code

In glue code we treat operations of the OCaml FFI as **external functions**.

```
value caml_update_ref(value r) {
  int x = Int_val (Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

$$\begin{array}{ll} \left\langle \mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \gamma \sim^{\theta}_{\mathrm{C}} w * v' \sim^{\theta}_{\mathrm{C}} w' \right\rangle & \\ \quad \mathsf{Store\_field}(w, i, w') & \sqsubseteq \Psi_{\mathrm{FFI}} \\ \left\langle \mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v}[i := v'] \right\rangle & \end{array}$$

External Calls in Glue Code

In glue code we treat operations of the OCaml FFI as **external functions**.

```
value caml_update_ref(value r) {
  int x = Int_val (Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

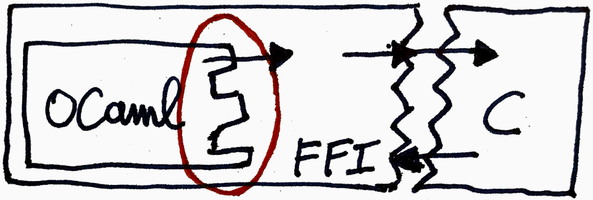
$$\{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \gamma \sim^{\theta}_{\mathrm{C}} w\}$$
$$\texttt{call caml\_update\_ref}\ [w] \ @\ \Psi_{\mathrm{FFI}}$$
$$\{w'.\, \exists m.\ \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m] * w' \sim^{\theta}_{\mathrm{C}} 0\}$$

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

$$\begin{array}{l}\langle \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \gamma \sim^{\theta}_{\mathrm{C}} w * v' \sim^{\theta}_{\mathrm{C}} w' \rangle \\ \quad \mathsf{Store\_field}(w, i, w') \qquad\qquad \sqsubseteq \Psi_{\mathrm{FFI}} \\ \langle \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v}[i := v'] \rangle\end{array}$$

Outline: The OCaml-FFI boundary

1. Language-local program logics with external calls
2. Glue code and program logic for FFI
3. Focus: the OCaml-FFI boundary



View Reconciliation

We assumed an interface for `caml_update_ref` that uses ML points-tos:

$$\forall \ell n.\ \langle \ell \mapsto_{\mathrm{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\ \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \rangle$$

Meanwhile, we proved the following specification for `caml_update_ref` using Ψ_{FFI}:

$$\begin{array}{l}\left\{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \gamma \sim^{\theta}_{\mathrm{C}} w\right\} \\ \quad \texttt{call caml\_update\_ref}\ [w]\ @\ \Psi_{\mathrm{FFI}} \\ \left\{w'.\ \exists m.\ \mathsf{GC}(\theta) * w' \sim^{\theta}_{\mathrm{C}} 0 * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m]\right\}\end{array}$$

These express two different views about the **same piece of state**!

View Reconciliation: Update Rules

Idea:

- make $\ell \mapsto_{\text{ML}} \vec{V}$ and $\gamma \mapsto_{\text{blk}[0|\textsf{mut}]} \vec{v}$ mutually exclusive (for related ℓ and γ)
- have *view reconciliation* rules to switch between the two representations

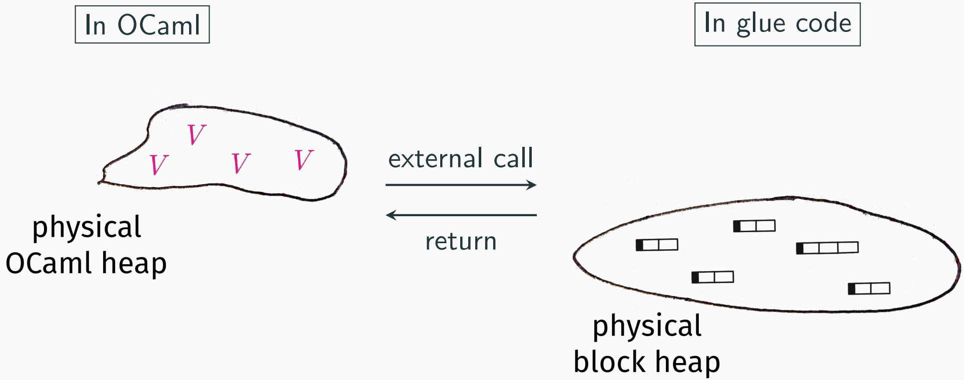
$$\mathsf{GC}(\theta) * \ell \mapsto_{\text{ML}} \vec{V} \equiv\!\!* \; \exists \vec{v}, \gamma.\, \mathsf{GC}(\theta) * \gamma \mapsto_{\text{blk}[0|\textsf{mut}]} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \quad \text{(ML-TO-FFI)}$$

$$\mathsf{GC}(\theta) * \gamma \mapsto_{\text{blk}[0|\textsf{mut}]} \vec{v} * \vec{V} \sim_{\text{ML}} \vec{v} \equiv\!\!* \; \exists \ell.\, \mathsf{GC}(\theta) * \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \quad \text{(FFI-TO-ML)}$$

View Reconciliation: Challenge

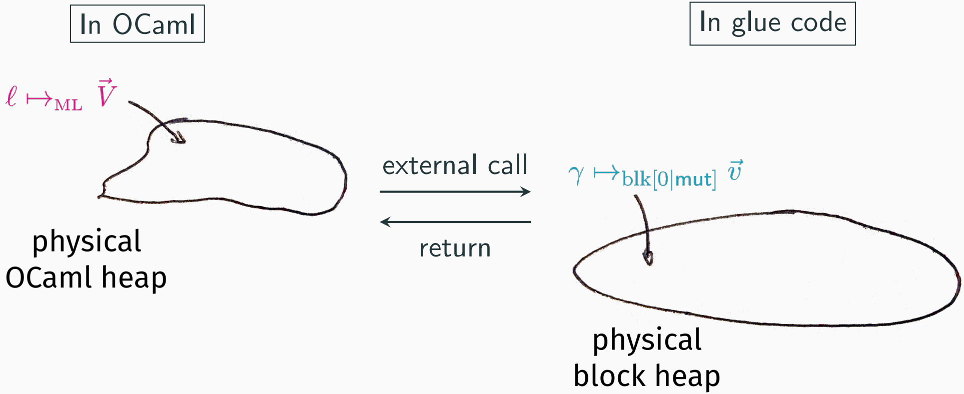
In **operational semantics**, there is *only one simultaneous view* of the OCaml state.

But resources do not reflect that!



View Reconciliation: Challenge (2) and Solution

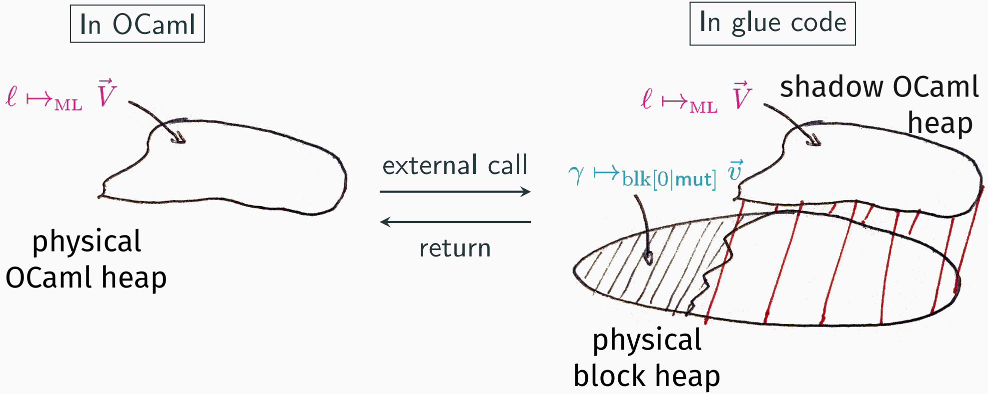
In **ghost state**: what happens to OCaml points-to?



View Reconciliation: Challenge (2) and Solution

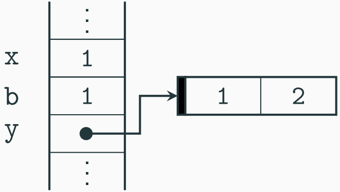
In **ghost state**: what happens to OCaml points-to?

Solution: track *both* views of the state in ghost state



Changing The Representation: Making Difficult Choices

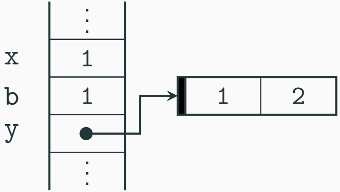
```
let x = ?
let b = ?
let y = (?, ?)
```



Quiz Time: What are the OCaml values of `x`, `b`, and `y`?

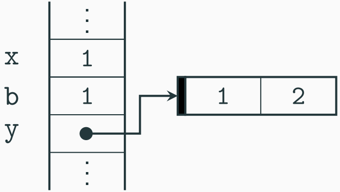
Changing The Representation: Making Difficult Choices

```
let x = 1
let b = true
let y = (1, 2)
```



Changing The Representation: Making Difficult Choices

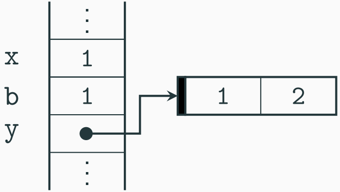
```
let x = 1
let b = true
let y = (1, 2)
```



High-level representation is **not unique**!

Changing The Representation: Making Difficult Choices

```
let x = 1
let b = true
let y = (1, 2)
```



High-level representation is **not unique**!

How does Operational Semantics choose the right value when switching to ML values?

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

$$\mathsf{wp}\ e\,\{\Phi\} :\hat{=} \cdots \vee \big(e\ \mathsf{reducible} * \forall e'.\, e \to e' \mathrel{-\!\!*} \mathsf{wp}\ e'\,\{\Phi\}\big) \qquad \text{usual Iris}$$

$$\mathsf{wp}\ e\,\{\Phi\} :\hat{=} \cdots \vee \big(\exists X.\, e \twoheadrightarrow X * \forall e'.\, e' \in X \mathrel{-\!\!*} \mathsf{wp}\ e'\,\{\Phi\}\big) \qquad \text{multi-relations}$$

Regular C and ML, not having angelic non-determinism, retain usual SOS

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

$$\mathsf{wp}\ e\ \{\Phi\} :\hat{=} \cdots \vee \big(e\ \mathsf{reducible} * \forall e'.\, e \to e' \twoheadrightarrow \mathsf{wp}\ e'\ \{\Phi\}\big) \qquad \text{usual Iris}$$

$$\mathsf{wp}\ e\ \{\Phi\} :\hat{=} \cdots \vee \big(\exists X.\, e \twoheadrightarrow X * \forall e'.\, e' \in X \mathrel{-\!\!*} \mathsf{wp}\ e'\ \{\Phi\}\big) \qquad \text{multi-relations}$$

Regular C and ML, not having angelic non-determinism, retain usual SOS

For adequacy, existential needs to be extracted ⇒ *transfinite Iris*

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

We give a **general recipe** for merging two languages:

1. Abstract over "the other side" using **interfaces and external calls**
2. Formalize the **semantics of the FFI** (memory model and primitives)
3. Bridge between memory models using **view reconciliation**

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

We give a **general recipe** for merging two languages:

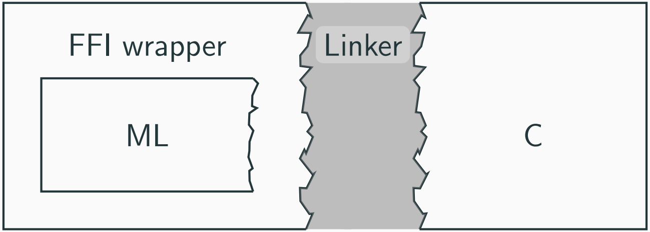
1. Abstract over "the other side" using **interfaces and external calls**
2. Formalize the **semantics of the FFI** (memory model and primitives)
3. Bridge between memory models using **view reconciliation**

More in the paper:

- **more detailed FFI**: callbacks, custom blocks, GC interaction
- **logical relation** for semantic typing of external functions

bonus slides

The semantics



The FFI wrapper

- Convert ML values to block-level
- Provide FFI: a C calling convention for ML

The Linker

- Link programs using the same calling convention
- Resolve external calls