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

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

Sept 11, 2023
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:
OCaml

C

Mind the gap!

Structured values
Integers and pointers
λ
ML
V
∈
Val::=(n∈Z)|(|ℓ∈Loc)|true|false|⟨⟩|⟨V,V⟩|λx.e

w∈Val::=(n∈Z)|(|a∈Addr)

Garbage collection
Manual memory management
Mind the gap!

OCaml

Structured values

\[ \lambda_{\text{ML}} \]

\[ V \in \text{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \text{Loc}) \]

<table>
<thead>
<tr>
<th>true</th>
<th>false</th>
</tr>
</thead>
</table>

| \langle \rangle | \langle V, V \rangle | \lambda x. e \cdot \cdot |

Garbage collection

C

Integers and pointers

\[ \lambda_{\text{C}} \]

\[ w \in \text{Val} ::= (n \in \mathbb{Z}) \mid (a \in \text{Addr}) \]

Manual memory management
Mind the gap!

OCaml FFI

Structured values
\[ \begin{align*}
\lambda_{\text{ML}} \quad V &\in \text{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \text{Loc}) \\
&\quad \mid \text{true} \mid \text{false} \\
&\quad \mid \langle \rangle \mid \langle V, V \rangle \mid \lambda x. e \quad \ldots
\end{align*} \]

Integers and pointers
\[ \begin{align*}
\lambda_{\text{C}} \quad w &\in \text{Val} ::= (n \in \mathbb{Z}) \mid (a \in \text{Addr})
\end{align*} \]

Garbage collection
Manual memory management
Write “glue code” using the OCaml FFI is **tricky and unsafe**. 

mistake $\Rightarrow$ *memory corruption* (often silent and hard to debug)

Which **rules** should I follow to safely use the OCaml FFI?
Key Challenge (as an expert in program logics)

We already have powerful *program logics* for OCaml and C

but those are for programs written in a *single* language

How do we **formally reason** about such multi-language code?
Can we build a program logic for reasoning about interoperability with an FFI, while preserving language-local reasoning?

\[ \lambda_{\text{ML}} \text{ Semantics} \]

\[ \lambda_{C} \text{ Semantics} \]

\[ \text{Iris}_{\text{ML}} \text{ Program Logic} \]

given as black box

\[ \text{Iris}_{C} \text{ Program Logic} \]

given as black box
Key challenge (in this work)

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

\[ \lambda_{\text{ML}} \] Semantics \hspace{2cm} \textit{Semantics for the FFI} \hspace{2cm} \lambda_{\text{C}} \] Semantics

\[ \text{Iris}_{\text{ML}} \] Program Logic \hspace{2cm} \textit{Program Logic for the FFI} \hspace{2cm} \text{Iris}_{\text{C}} \] Program Logic

given as black box \hspace{5cm} \text{what we need} \hspace{5cm} \text{given as black box}

Design choice: \textbf{reuse} most of existing semantics/program logics; \textbf{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!

**In Iris:** the logic is proved sound and all proofs are checked in Coq
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
The OCaml FFI deals with two core challenges:

- mediating between the different views of the OCaml memory
- interacting with the OCaml GC
Using the OCaml FFI: examples

The OCaml FFI deals with two core challenges:

- mediating between the different views of the OCaml memory
- interacting with the OCaml GC
Example: updating an OCaml reference from C code

**OCaml code:**

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

**C code:**

```c
int rand(int x) { ... }
```
Example: updating an OCaml reference from C code

OCaml code:

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

C code:

```c
int rand(int x) { ... }
```
Example: updating an OCaml reference from C code

OCaml code:

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

C code:

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

Glue code:

```c
value caml_update_ref(value r) {
  /* TODO */
  int y = rand(x);
  /* TODO */
}
```
Example: updating an OCaml reference from C code

OCaml code:

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

C code:

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

Glue code:

```c
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:

```ocaml
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 \textit{low-level} representation of OCaml values.
Example: updating an OCaml reference from C code

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

C code:
```c
int rand(int x) { ... }
```

Glue code:
```ocaml
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...
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 () */
}
```c
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 () */
}
```
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 () */
}
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 () */
}

```
  n
   |
   | r
   |   |
   |   |
   | x
   |   |
   | y
   |   |
```

m

```
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 () */
}
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 () */
}
The OCaml FFI deals with two core challenges:

- mediating between the different views of the OCaml memory
- interacting with the OCaml GC
Example: swapping an OCaml pair

OCaml code:

```ocaml
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

This implementation is unfortunately incorrect and will silently corrupt memory!

OCaml has a "tracing" garbage collector. Starts from roots; collects unreachable blocks; may also move blocks in memory.

```
let x, y = let l = [1; 2; 3] in (List.filter even l, List.tl l)
```
Example: swapping an OCaml pair

**OCaml code:**

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

**Glue code:**

(first attempt)

```
value caml_swap_pair(value p)
{
  value r = caml_alloc(0, 2); /* allocate a block for the result */
  value x = Field(p, 0);    /* read the input pair */
  value y = Field(p, 1);
  Store_field(r, 0, y);     /* initialize the output pair */
  Store_field(r, 1, x);
  return r;                /* return it */
}
```

This implementation is unfortunately incorrect and will silently corrupt memory!

OCaml has a "tracing" garbage collector. Starts from racines; collects unreachable blocks; may also move blocks in memory.

```
let x, y = let l = [1; 2; 3] in (List.filter even l, List.tl l)
...
```

1 2 3
2 x y

12
Example: swapping an OCaml pair

**OCaml code:**

```ocaml
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

**Glue code:**

(first attempt)

```c
value caml_swap_pair(value p)
{
    value r = caml_alloc(0, 2); /* allocate a block for the result */
    value x = Field(p, 0);      /* read the input pair */
    value y = Field(p, 1);
    Store_field(r, 0, y);       /* initialize the output pair */
    Store_field(r, 1, x);
    return r;                  /* return it */
}
```

This implementation is unfortunately **incorrect** and will silently corrupt memory!

*caml_alloc* may run the GC which does not know about C variables and arguments...
OCaml has a “tracing” garbage collector. Starts from *roots*; collects unreachable blocks; may also *move* blocks in memory.

```ocaml
let x, y = 
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...```

---

Example: swapping an OCaml pair

OCaml code:

```ocaml
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

Glue code:

(first attempt)

```ocaml
value caml_swap_pair(value p) {
  value r = caml_alloc(0, 2); /* allocate a block for the result */
  value x = Field(p, 0); /* read the input pair */
  value y = Field(p, 1);
  Store_field(r, 0, y); /* initialize the output pair */
  Store_field(r, 1, x);
  return r; /* return it */
}
```

This implementation is unfortunately incorrect and will silently corrupt memory!

`caml_alloc` may run the GC which does not know about C variables and arguments...

OCaml has a “tracing” garbage collector. Starts from *roots*; collects unreachable blocks; may also *move* blocks in memory.
Swapping pairs in the presence of a garbage collector

This implementation is unfortunately **incorrect!**

```plaintext
value caml_swap_pair(value p) {
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
This implementation is unfortunately **incorrect!**

```caml
value caml_swap_pair(value p)
{
    value r = caml_alloc(0, 2); <--
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
Registering roots

CAMLparam1(p) registers &p as a GC root.

The GC will avoid collecting the block, and will update p if the block moves.

value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
Registering roots

CAMLparam1(p) registers &p as a GC root.

The GC will avoid collecting the block, and will update p if the block moves.

```ml
value caml_swap_pair(value p)
{
    CAMLparam1(p); <--
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
Registering roots

\texttt{CAMLparam1}(p) registers \&p as a GC root.

The GC will avoid collecting the block, and will \textit{update} \(p\) if the block moves.

\begin{verbatim}
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);  \\
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
\end{verbatim}
Registering roots

CAMLparam1(p) registers &p as a GC root.
The GC will avoid collecting the block, and will update p if the block moves.

```plaintext
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```

...
Registering roots

CAMLparam1(p) registers \&p as a GC root.
The GC will avoid collecting the block, and will *update* p if the block moves.

```ocaml
def value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
Registering roots

CAMLparam1(p) registers &p as a GC root.

The GC will avoid collecting the block, and will update p if the block moves.

```ml
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
Registering roots

CAMLparam1(p) registers &p as a GC root.
The GC will avoid collecting the block, and will *update* p if the block moves.

```ocaml
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```
Unregistering roots

One subtle **bug** remains!

```ocaml
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```

The GC will continue to update \&p after the function returns, corrupting the stack...!

We must use `CAMLreturn()` to unregister local roots when returning.
Unregistering roots

One subtle **bug** remains!

```c
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    return r;
}
```

The GC will *continue to update* &p after the function returns, corrupting the stack…!

We must use `CAMLreturn()` to unregister local roots when returning.
Our final implementation for swap_pair

external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"

value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}
1. Language-local program logics with external calls
Language-local reasoning

We reuse:

\( \lambda_{ML} \) Semantics

\( \text{Iris}_{ML} \)
Program Logic

\( \lambda_C \) Semantics

\( \text{Iris}_C \)
Program Logic

The one change: a minimal extension allowing 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):

$$e \in \text{Expr} ::= \cdots \mid \text{call } fn \ e$$
We model external calls as a new syntactic construct (inlineing the declaration):

\[ e \in \text{Expr} ::= \cdots \mid \text{call \ } fn \ \bar{e} \]

We assign **no semantics** to external calls: they are simply stuck!
We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

\[
\forall \ell \ n. \ \{ \ell \mapsto_{ML} n \} \text{call caml_update_ref}[\ell] \ {V'}. \ \exists m. \ V' = \emptyset \ast \ell \mapsto_{ML} m \}_{ML}
\]
Interface Specifications

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

\[ \forall \ell n. \{ \ell \rightarrow_{ML} n \} \text{call} \ caml\text{\textunderscore}update\text{\textunderscore}ref [\ell] \{ V'. \exists m. V' = \langle \rangle \ast \ell \rightarrow_{ML} m \} \}_{ML} \]

“\(\ell \rightarrow_{ML} V\)” is a *Separation Logic assertion*

- asserts that the memory location \(\ell\) stores the value \(V\)
- grants the *permission* to access the location (read/write)
Interface Specifications

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

\[
\forall \ell \ n. \ {\ell \mapsto_{ML} n} \ \text{call} \ caml\_update\_ref[\ell] \ {V'. \exists m. \ V' = } \langle \rangle \ast \ell \mapsto_{ML} m \}_{ML}
\]

To do so, we introduce **interfaces** \( \Psi \) and Hoare triples \( \{P\} e @ \Psi \{v. Q\} \) that verify programs against them. For example, for `caml_update_ref`, we assume:

\[
\forall \ell \ n. \ {\ell \mapsto_{ML} n} \ caml\_update\_ref[\ell] \ {V'. \exists m. \ V' = } \langle \rangle \ast \ell \mapsto_{ML} m \} \sqsubseteq \Psi
\]
We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

\[ \forall \ell \, n. \{ \ell \mapsto_{ML} n \} \text{call caml_update_ref}[\ell]\{V'. \exists m. \, V' = \langle \rangle \ast \ell \mapsto_{ML} m\}_{ML} \]

To do so, we introduce **interfaces** \( \Psi \) and Hoare triples \( \{P\} e @ \Psi\{v. Q\} \) that verify programs against them. For example, for `caml_update_ref`, we assume:

\[ \forall \ell \, n. \langle \ell \mapsto_{ML} n \rangle \text{caml_update_ref}[\ell]\langle V'. \exists m. \, V' = \langle \rangle \ast \ell \mapsto_{ML} m \rangle \subseteq \Psi \]

⚠️ This is an assumption, not a Hoare triple ⚠️
Implement interface triples as a predicate transformer $\Psi$:

$$\Psi : \begin{array}{c} \text{Name} \\ \text{Args} \\ \text{Postcondition} \\ \text{Precondition} \end{array} \rightarrow \begin{array}{c} \text{Val} \\ \text{Val} \rightarrow \text{iProp} \end{array} \rightarrow \text{iProp}$$
Implement interface triples as a predicate transformer $\Psi$:

$$
\Psi : \text{FnName} \rightarrow \vec{Val} \rightarrow (\text{Val} \rightarrow \text{iProp}) \rightarrow \text{iProp}
$$

“$\text{iProp}$” is the type of Iris propositions, which includes:

- quantifiers $\forall, \exists, ...$ and pure propositions
- Separation Logic modalities
- memory assertions of both languages $(\ell \leftrightarrow_{\text{ML}} V, a \leftrightarrow_{\text{C}} w)$
- specifications $\{P\} e \circ \Psi \{Q\}$ of both languages
Implement interface triples as a predicate transformer $\Psi$:

$$
\Psi : \begin{array}{c}
\text{name} \\
\text{args} \\
\text{postcondition} \\
\text{precondition}
\end{array}
\rightarrow
\begin{array}{c}
\text{name} \\
\text{args} \\
\text{postcondition} \\
\text{precondition}
\end{array}
\rightarrow
\begin{array}{c}
\text{val} \\
\text{val} \\
\text{iProp} \\
\text{iProp}
\end{array}
\rightarrow
\begin{array}{c}
\text{val} \\
\text{val} \\
\text{iProp} \\
\text{iProp}
\end{array}
\rightarrow
\begin{array}{c}
iProp
\end{array}
$$

We desugar

$$
\forall \ell \, n. \langle \ell \mapsto_{ML} n \rangle \text{caml_update_ref} [\ell] \\
\langle V' \cdot \exists m. V' = \langle \rangle \ast \ell \mapsto_{ML} m \rangle
$$
Implement interface triples as a predicate transformer $\Psi$:

\[
\Psi : \text{FnName} \rightarrow \vec{Val} \rightarrow (Val \rightarrow iProp) \rightarrow iProp
\]

We desugar

\[
\forall \ell. n. \langle \ell \mapsto_{ML} n \rangle \text{caml_update_ref}[\ell] \\
\langle V'. \exists m. V' = \langle \rangle \ast \ell \mapsto_{ML} m \rangle
\]

as follows:

\[
\Psi_{upd\ fn\ \vec{V}}\ \Phi := \exists n. \ell \mapsto_{ML} n \ast fn = \text{caml_update_ref} \ast \vec{V} = [\ell] \\
\ast (\forall V'. m. V' = \langle \rangle \ast \ell \mapsto_{ML} m \ast \Phi(V'))
\]
We parameterize Hoare triples by $\Psi$ (inspired by de Vilhena and Pottier [2021]):

```
{P} \ e @ \Psi \ {Q}
```

"Starting from a state satisfying $P$, $e$ reduces to a value arriving in a state satisfying $Q$ — either by normal reductions, or by making external calls that satisfy $\Psi$"
We parameterize Hoare triples by $\Psi$ (inspired by de Vilhena and Pottier [2021]):

"{$P\} e @ \Psi \{Q\}$" means:

"Starting from a state satisfying $P$, $e$ reduces to a value arriving in a state satisfying $Q$ — either by normal reductions, or by making external calls that satisfy $\Psi$"

**Note:** In a OCaml-and-C program (after linking), adequacy holds for $\Psi \ fn \ V \Phi := \bot$
Implementing Interface Triples

In Iris, we then define Hoare triples in terms of the operational semantics:

\[
\{ P \} \ e @ \Psi \ \{ Q \} \ ::= \ \Box (P \rightarrow^* \ wp\ e @ \Psi \ \{ Q \})
\]

\[
\wp\ e @ \Psi \ \{ Q \} \ ::= \begin{cases} 
Q(v) & \text{e} = v \\
\forall e', (e \rightarrow e') \Rightarrow \ wp\ e' @ \Psi \ \{ Q \} & \text{e reducible} \\
\Psi\ fn\ \vec{V} \ (\lambda \vec{V}'. wp\ K[V'] @ \Psi \ \{ Q \}) & \text{Postcondition} \\
\end{cases}
\]

\[
e = \text{K[call fn \vec{V}]}
\]
Outline: The OCaml FFI

1. Language-local program logics with external calls
2. Glue code and program logic for FFI
In glue code we treat operations of the OCaml FFI as **external functions**.

```c
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** $\Psi_{\text{FFI}}$ for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto _{\text{blk}[t|m]} \nu$.
External Calls in Glue Code

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

```c
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** $\Psi_{\text{FFI}}$ for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$.

\[
\begin{align*}
\langle \text{GC}(\theta) * \gamma \mapsto_{\text{blk}[0|\text{mut}]} \vec{v} * \gamma \sim^\theta_{C} w * v' \sim^\theta_{C} w' \rangle & \\
\text{Store_field}(w, i, w') & \subseteq \Psi_{\text{FFI}} \\
\langle \text{GC}(\theta) * \gamma \mapsto_{\text{blk}[0|\text{mut}]} \vec{v}[i := v'] \rangle
\end{align*}
\]
External Calls in Glue Code

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

```ocaml
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 $\Psi_{FFI}$ for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{blk[t|m]} \vec{v}$.

$$\langle GC(\theta) \cdot \gamma \mapsto_{blk[0|mut]} [n] \cdot \gamma \sim^{\theta} C w \rangle \quad \text{call caml_update_ref}[w] @ \Psi_{FFI}$$

$$\{ w' \cdot \exists m. GC(\theta) \cdot \gamma \mapsto_{blk[0|mut]} [m] \cdot w' \sim^{\theta} C 0 \}$$
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
We assumed an interface for `caml_update_ref` that uses ML points-tos:

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

Meanwhile, we proved the following specification for `caml_update_ref` using \( \Psi_{\text{FFI}} \):

\[
\begin{align*}
\{ \text{GC}(\theta) * \gamma \mapsto_{\text{blk[0|mut]}} [n] * \gamma \sim_{C}^\theta w \} \\
\text{call caml\_update\_ref} [w] @ \Psi_{\text{FFI}} \\
\{ w'. \exists m. \text{GC}(\theta) * w' \sim_{C}^\theta 0 * \gamma \mapsto_{\text{blk[0|mut]}} [m] \}
\end{align*}
\]

These express two different views about the same piece of state!
View Reconciliation: Update Rules

Idea:

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

\[
\text{GC}(\theta) * \ell \mapsto_{\text{ML}} \vec{V} \equiv \exists \vec{v}, \gamma. \text{GC}(\theta) * \gamma \mapsto_{\text{blk[0|mut]}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \quad (\text{ML-TO-FFI})
\]
\[
\text{GC}(\theta) * \gamma \mapsto_{\text{blk[0|mut]}} \vec{v} * \vec{V} \sim_{\text{ML}} \vec{v} \equiv \exists \ell. \text{GC}(\theta) * \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \quad (\text{FFI-TO-ML})
\]
Challenge: **proving** that the view reconciliation rules are sound is hard!
Challenge: **proving** that the view reconciliation rules are sound is hard!

The standard workflow in Iris:

- have Separation Logic memory assertions ($\ell \rightarrow_{ML} V$)
- have the state of the operational semantics (finite map: Location $\rightarrow$ Value)
- *relate the two* ("state interpretation"). **Often** straightforward...
Challenge: **proving** that the view reconciliation rules are sound is hard!

- in the **program logic**, we can hold a mix of $\ell \mapsto_{ML} \vec{V}$ and $\gamma \mapsto_{blk[0|mut]} \vec{v}$
- the **operational semantics** has *only one simultaneous view* of the OCaml state

How can we relate the assertions and the operational semantics state?
In the **operational semantics**, there is *only one simultaneous view* of the OCaml state.
In the **program logic**: what happens to OCaml points-to?

In OCaml:

\[ l \leftrightarrow_{ML} V \]

Physical OCaml heap

In glue code:

\[ \gamma \leftrightarrow_{blk[0|mut]} \bar{v} \]

Physical block heap

External call

Return
In the **program logic**: what happens to OCaml points-to?

**Solution**: track *both* views of the state in the program logic.
Quiz Time: What are the OCaml values of $x$, $b$, and $y$?
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?
We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!
We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

\[
\begin{align*}
\text{wp } e \{ \Phi \} & := \cdots \lor (e \text{ reducible } \cdot \forall e'.e \rightarrow e' \rightarrow\text{ wp } e' \{ \Phi \}) \quad \text{usual Iris} \\
\text{wp } e \{ \Phi \} & := \cdots \lor (\exists X. e \rightarrow X \cdot \forall e'.e' \in X \rightarrow\text{ wp } e' \{ \Phi \}) \quad \text{multi-relations}
\end{align*}
\]

Regular C and ML, not having angelic non-determinism, retain usual SOS
We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

\[
\begin{align*}
\text{wp } e \{\Phi\} & := \cdots \lor (e \text{ reducible } \land \forall e'. e \rightarrow e' \rightarrow \ast \text{ wp } e' \{\Phi\}) \quad \text{usual Iris} \\
\text{wp } e \{\Phi\} & := \cdots \lor (\exists X. e \rightarrow X \land \forall e'. e' \in X \rightarrow \ast \text{ wp } e' \{\Phi\}) \quad \text{multi-relations}
\end{align*}
\]

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

For adequacy, existential needs to be extracted \(\Rightarrow\) 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: https://melocoton-project.github.io

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

```ocaml
define caml_swap_pair(value p) {
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```

**Permissions:**
- `GC(\alpha)`
- `\gamma \mapsto \text{blk}[0\mid \text{imm}][x; y]`
- `\&p \mapsto_c p0`

**Facts:**
- `\text{blkaddr}(\alpha, \gamma) = p0`
A *permission* describes the right to access some resources or memory:

- $\text{GC}(\alpha)$: permission to use C functions of the FFI
  - $\rightarrow \alpha$: an abstract name that identifies a **specific layout** of the GC memory.
  - ($\alpha$ changes when the GC moves or deallocates block)

- $\gamma \mapsto \text{blk}[0|\text{imm}] [x; y; ...]$: permission to access a block in the GC memory
  - $\rightarrow \gamma$: abstract **label** of the block
  - $\rightarrow [x; y; ...]$: contents of the block

- $\&p \mapsto_c p0$: permission to access the C variable $p$
  - $\rightarrow p0$: current value of the variable
Checking `swap_pair`

```ocaml
value caml_swap_pair(value p) {
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```

**Permissions:**

- `GC(α)`
- `γ ↦_{blk[0|imm]} [x; y]`
- `&p ↦_{C} p0`

**Facts:**

- `blkaddr(α, γ) = p0`

We also collect "facts" (mathematical equalities) of the form:

- `blkaddr(α, γ) = p0` means: the block with label `γ` has concrete address `p0`, when the GC memory is in layout `α`.

**Permissions:**

- `GC(α)`
- `γ ↦_{blk[0|imm]} [x; y]`
- `&p ↦_{C} p0`

**Facts:**

- `blkaddr(α, γ) = p0`
We also collect “facts” (mathematical equalities) of the form:

\[ \text{blkaddr}(\alpha, \gamma) = p0 \]

means: the block with label \( \gamma \) has concrete address \( p0 \),

*when the GC memory is in layout \( \alpha \)*
Checking `swap_pair`

```ocaml
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}
```

Permissions:

- `GC(α)`
- `γ ↦ blk[0|imm][x; y]`
- `&p ↦_C p0`

Facts:

- `blkaddr(α, γ) = p0`
Checking `swap_pair`

value `caml_swap_pair(value p)` {
    `CAMLparam1(p);`
    value `r = caml_alloc(0, 2);`
    value `x = Field(p, 0);`
    value `y = Field(p, 1);`
    `Store_field(r, 0, y);`
    `Store_field(r, 1, x);`
    `CAMLreturn(r);`
}

Permissions:

`GC(\alpha)`

`\gamma \mapsto blk[0|imm] [x; y]`

`&p \mapsto_C p0`

Facts:

`blkaddr(\alpha, \gamma) = p0`

\[
\langle GC(\alpha) \ast &p \mapsto_C p0 \ast blkaddr(\alpha, \gamma) = p0 \rangle
\]

\[
CAMLparam1(p)
\]

\[
\langle GC(\alpha) \ast &p \mapsto_{root} \gamma \rangle
\]
Checking swap_pair

value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}

Permissions:
GC(α)
γ ↦ β[0|imm] [x; y]
&p ↦ root γ

Facts:
blkaddr(α, γ) = p0

⟨GC(α) * &p ↦ C p0 * blkaddr(α, γ) = p0⟩
CAMLparam1(p)
⟨GC(α) * &p ↦ root γ⟩
Checking `swap_pair`

```
value caml_swap_pair(value p) {
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```

Permissions:
- `GC(α)`
- `γ \mapsto_{\text{blk}[0|\text{imm}]} [x; y]`
- `&p \mapsto_{\text{root}} γ`

Facts:
- `blkaddr(α, γ) = p0`

```
⟨GC(α)⟩
caml_alloc(0, n)
⟨r. ∃β. blkaddr(β, δ) = r * GC(β) * δ \mapsto_{\text{blk}[0|\text{imm}]} [?; ...; ?]⟩
```
Checking swap_pair

```ocaml
cvalue caml_swap_pair(cvalue p) { 
  CAMLparam1(p);
  cvalue r = caml_alloc(0, 2);
  cvalue x = Field(p, 0);
  cvalue y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```

**Permissions:**

- GC(β)
- γ → blk[0|imm] [x; y]
- &p → root γ
- δ → blk[0|imm] [?; ?]

**Facts:**

- blkaddr(α, γ) = p₀
- blkaddr(β, δ) = r

\[\langle \text{GC}(\alpha) \rangle\]

\[
\text{caml\_alloc}(0, n)
\]

\[
\langle r. \exists \beta. \text{blkaddr}(\beta, \delta) = r \star \text{GC}(\beta) \star \delta \rightarrow \text{blk}[0|imm] \ [?; \ldots; ?]\rangle
\]
Checking `caml_swap_pair`

```plaintext
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}
```

Permissions:

- `GC(\beta)`
- $\gamma \mapsto \text{blk}[0|\text{imm}] [x; y]$
- $\delta \mapsto \text{blk}[0|\text{imm}] [?; ?]$

Facts:

- `blkaddr(\alpha, \gamma) = p0`
- `blkaddr(\beta, \delta) = r`
- `blkaddr(\beta, \gamma) = p1`

Rule for reading a root `&p`

```
\langle GC(\beta) \& p \mapsto_{\text{root}} \gamma \rangle \star (\& p)
\langle p1. \text{blkaddr}(\beta, \gamma) = p1 \star GC(\beta), \& p \mapsto_{\text{root}} \gamma \rangle
```
Checking `swap_pair`

```cpp
value caml_swap_pair(value p)
{
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}
```

Permissions:
- \( GC(\beta) \)
- \( \gamma \mapsto_{\text{blk}[0|\text{imm}]} [x; y] \)
- \( \&p \mapsto_{\text{root}} \gamma \)
- \( \delta \mapsto_{\text{blk}[0|\text{imm}]} [?; ?] \)

Facts:
- \( \text{blkaddr}(\alpha, \gamma) = p_0 \)
- \( \text{blkaddr}(\beta, \delta) = r \)
- \( \text{blkaddr}(\beta, \gamma) = p_1 \)

\[
\langle GC(\beta) \ast \gamma \mapsto_{\text{blk}[0|\text{imm}]} [\ldots; v_i; \ldots] \ast \text{blkaddr}(\beta, \gamma) = p \rangle \\
Field(p, i) \\
\langle v_i \cdot GC(\beta), \gamma \mapsto_{\text{blk}[0|\text{imm}]} [\ldots; v_i; \ldots] \rangle
\]
Checking \texttt{swap\_pair}

\begin{verbatim}
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}
\end{verbatim}

\[\langle \text{GC}(\beta) \ast \gamma \mapsto \text{blk}[0|\text{imm}] \ [\ldots; v_i; \ldots] \ast \text{blkaddr}(\beta, \gamma) = p \rangle \]

\(\text{Field}(p, i)\)

\[\langle v_i. \text{GC}(\beta) , \gamma \mapsto \text{blk}[0|\text{imm}] \ [\ldots; v_i; \ldots] \rangle\]
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}

Permissions:

\[ \text{GC}(\beta) \]
\[ \gamma \mapsto \text{blk}[0\mid \text{imm}] [x; y] \]
\[ \&p \mapsto \text{root} \gamma \]
\[ \delta \mapsto \text{blk}[0\mid \text{imm}] [y; ?] \]

Facts:

\[ \text{blkaddr}(\alpha, \gamma) = p_0 \]
\[ \text{blkaddr}(\beta, \delta) = r \]
\[ \text{blkaddr}(\beta, \gamma) = p_1 \]
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}

Permissions:

\[ \text{GC}(\beta) \]
\[ \gamma \mapsto \text{blk}[0|\text{imm}][x; y] \]
\[ \&p \mapsto \text{root} \gamma \]
\[ \delta \mapsto \text{blk}[0|\text{imm}][y; x] \]

Facts:

\[ \text{blkaddr}(\alpha, \gamma) = p_0 \]
\[ \text{blkaddr}(\beta, \delta) = r \]
\[ \text{blkaddr}(\beta, \gamma) = p_1 \]

\[ \langle \text{GC}(\beta) * \delta \mapsto \text{blk}[0|\text{imm}][..; v_i; ...] \rangle * \text{blkaddr}(\beta, \delta) = r \]

\[ \text{Store_field}(r, i, v) \]

\[ \langle \text{GC}(\beta) * \delta \mapsto \text{blk}[0|\text{imm}][..; v; ...] \rangle \]
value caml_swap_pair(value p) {
    CAMLparam1(p);
    value r = caml_alloc(0, 2);
    value x = Field(p, 0);
    value y = Field(p, 1);
    Store_field(r, 0, y);
    Store_field(r, 1, x);
    CAMLreturn(r);
}