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:
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)?
OCaml

C

Mind the gap!
Mind the gap!

**OCaml**

Structured values

\[
\lambda_{ML} \quad V \in Val ::= (n \in \mathbb{Z}) \mid (\ell \in Loc) \\
\mid \text{true} \mid \text{false} \\
\mid \langle \rangle \mid \langle V, V \rangle \ldots
\]

Garbage collection

\[
\text{Iris}_{ML} \quad \ell \mapsto_{ML} \vec{V}
\]

**C**

Integers and pointers

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

Manual memory management

\[
\text{Iris}_C \quad a \mapsto_C w
\]
Mind the gap!

\[ \text{OCaml} \leftrightarrow \text{OCaml FFI} \rightarrow \text{C} \]

**Structured values**

\[ \lambda_{\text{ML}} \quad V \in \text{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \text{Loc}) \]
\[ \mid \text{true} \mid \text{false} \]
\[ \mid \langle \rangle \mid \langle V, V \rangle \cdots \]

**Integers and pointers**

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

**Garbage collection**

\[ \text{Ir} \text{ris}_{\text{ML}} \quad \ell \mapsto_{\text{ML}} \vec{V} \]

**Manual memory management**

\[ \text{Ir} \text{ris}_{\text{C}} \quad a \mapsto_{\text{C}} w \]
Key Challenge

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

\[ \lambda_{ML} \text{ Semantics} \]
\[ \lambda_C \text{ Semantics} \]
\[ \text{Iris}_{ML} \text{ Program Logic} \]
\[ \text{Iris}_C \text{ Program Logic} \]

given as black box

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

\[ \lambda_{ML} \text{ Semantics} \quad \text{Semantics for the FFI} \quad \lambda_{C} \text{ Semantics} \]

\[ \text{Iris}_{ML} \text{ Program Logic} \quad \text{Program Logic for the FFI} \quad \text{Iris}_{C} \text{ Program Logic} \]

given as black box \quad \text{what we need} \quad \text{given as black box}

**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!
1. Language-local program logics with external calls
Outline

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

```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
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
text 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:

```
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:**
```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) {
  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 () */
}
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 () */
}
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 () */
}
1. Language-local program logics with external calls
Language-local reasoning

We reuse:

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

\[ \lambda_C \text{ Semantics} \]

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

\[ \text{Iris}_C \text{ Program Logic} \]

The one change: a minimal extension allowing \textit{external calls}. 
We model external calls as a new syntactic construct (inlining the declaration):

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

We assign no semantics to external calls: they are simply stuck!
We model external calls as a new syntactic construct (inlining the declaration):

\[
e \in \text{Expr} ::= \cdots \mid \text{call } fn \; \vec{e}
\]
We model external calls as a new syntactic construct (inlining the declaration):

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

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:

$$\forall \ell \ n. \ \{ \ell \mapsto_{ML} n \} \text{call} \ \text{caml_update_ref}[\ell] \ \{ V'. \exists m. \ V' = \langle \rangle * \ell \mapsto_{ML} m \}$$
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 weakest preconditions $\text{wp } e @ \Psi \{v. Q\}$ that verify programs against them. For example, for `caml_update_ref`, we assume:

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

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

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

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

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

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

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

We desugar

$$\forall \ell \ n. \langle \ell \mapsto_{ML} n \rangle \text{caml_update_ref}[\ell]$$

$$\langle V'. \exists m. V' = \langle \rangle \ast \ell \mapsto_{ML} m \rangle$$
Implement interface triples as a predicate transformer $\Psi$:

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

We desugar

$$
\forall \ell \ n. \langle \ell \mapsto_{\text{ML}} n \rangle \text{caml_update_ref} \ [\ell]
$$

$$
\langle V'. \exists m. \ V' = \langle \rangle \ * \ell \mapsto_{\text{ML}} m \rangle
$$

as follows:

$$
\Psi_{\text{upd} \ fn \ \vec{V} \ \Phi} := \exists \ell n. \ell \mapsto_{\text{ML}} n * fn = \text{caml_update_ref} * \vec{V} = [\ell]
$$

$$
* (\forall V' m. \ V' = \langle \rangle * \ell \mapsto_{\text{ML}} m \rightarrow* \Phi(V'))
$$
Implementing Interface Triples

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

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

\[
\begin{align*}
\text{wp } e @ \Psi \{ \Phi \} :=
\begin{cases}
\Phi(v) & e = v \\
\forall e', (e \rightarrow e') \Rightarrow \text{wp } e' @ \Psi \{ \Phi \} & e \text{ reducible} \\
\Psi \ fn \ \vec{V} \ (\lambda \ V'. \text{wp } K[V'] @ \Psi \{ \Phi \}) & e = K[\text{call } fn \ \vec{V}] \\
\end{cases}
\end{align*}
\]
Implementing Interface Triples

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

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

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

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

1. Language-local program logics with external calls
2. Glue code and program logic for FFI
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. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$. 
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. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$. 

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

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

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

$$\langle GC(\theta) * \gamma \mapsto_{blk[0|\mathcal{m}]} \vec{v} * \gamma \sim_C w * v' \sim_C w' \rangle$$

$$\text{Store\_field}(w, i, w')$$

$$\langle GC(\theta) * \gamma \mapsto_{blk[0|\mathcal{m}]} \vec{v}[i := v'] \rangle$$

$$\subseteq \Psi_{FFI}$$
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:

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

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

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

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

Idea:

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

\[
\text{GC}(\theta) * \ell \mapsto_{ML} \vec{V} \equiv* \exists \vec{v}, \gamma. \text{GC}(\theta) * \gamma \mapsto_{blk[0|mut]} \vec{v} * \ell \sim_{ML} \gamma * \vec{V} \sim_{ML} \vec{v} \quad (\text{ML-TO-FFI})
\]
\[
\text{GC}(\theta) * \gamma \mapsto_{blk[0|mut]} \vec{v} * \vec{V} \sim_{ML} \vec{v} \equiv* \exists \ell. \text{GC}(\theta) * \ell \mapsto_{ML} \vec{V} * \ell \sim_{ML} \gamma \quad (\text{FFI-TO-ML})
\]
In operational semantics, there is only one simultaneous view of the OCaml state. But resources do not reflect that!
In **ghost state**: what happens to OCaml points-to?

- **In OCaml**
  - \( \ell \xrightarrow{\text{ML}} \mathbf{V} \)

- **In glue code**
  - \( \gamma \xrightarrow{\text{blk}[0|\text{mut}]} \mathbf{\bar{v}} \)

[Diagram showing physical OCaml heap and physical block heap connected by external call and return operations.]
In **ghost state**: what happens to OCaml points-to?

**Solution**: track *both* views of the state in ghost state.
Quiz Time: What are the OCaml values of $x$, $b$, and $y$?
let x = 1
define as x := 1
let b = true
define as b := true
let y = (1, 2)
define as y := (1, 2)
let x = 1
let b = true
let y = (1, 2)

High-level representation is \textit{not unique}!
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)!

\[
\text{wp } e \{\Phi\} \equiv \cdots \lor (\text{e reducible } \land \forall e'. e \rightarrow e' \rightarrow\!
\!\!\!\rightarrow \text{wp } e' \{\Phi\}) \quad \text{usual Iris}
\]

\[
\text{wp } e \{\Phi\} \equiv \cdots \lor (\exists X. e \rightarrow X \land \forall e'. e' \in X \rightarrow\!
\!\!\!\rightarrow \text{wp } e' \{\Phi\}) \quad \text{multi-relations}
\]

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

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

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:

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