Melocoton
A Program Logic for Verified Interoperability Between OCaml and C

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

Journées SCALP
28 Novembre 2023
Multi-Language Programs Are Everywhere

NumPy
Python

Firefox
C++

OpenSSL
C
Rust
JavaScript

Bindings for:
- Rust
- Python
- OCaml
- Go
- ...

Fortran
C

Multi-Language Programs Are Everywhere

OCaml-SSL - OCaml bindings for the libssl

a mixture of C and OCaml code connected using the OCaml **Foreign Function Interface (FFI)**
OCaml FFI code is **unsafe** and must follow **subtle FFI rules**

Buggy FFI code can produce **segfaults**, **corrupt memory**, break **type safety** and **data abstraction** guarantees of OCaml
How do we verify functional correctness of code written in different languages?
Single-Language Functional Correctness

Hoare Logic for simple imperative languages.
Separation Logic for modularity and aliasing.
Multi-Language Functional Correctness
Multi-Language Functional Correctness

Existing work on Semantics and Logical Relations. How do we prove functional correctness of individual, potentially unsafe libraries?
A Multi-Language Program in OCaml and C
A Multi-Language Program in OCaml and C

C business logic

```c
void hash_ptr(int * x) {
    // Implemented in OpenSSL
    // tedious to port to OCaml
}
```
A Multi-Language Program in OCaml and C

**OCaml business logic**

```ocaml
let main () =
  let r = ref 42 in
  hash_ref r; (*written in C*)
  print_int !r
```

**C business logic**

```c
void hash_ptr(int * x) {
  // Implemented in OpenSSL
  // tedious to port to OCaml
}
```
A Multi-Language Program in OCaml and C

**OCaml** business logic

```ocaml
let main () =
    let r = ref 42 in
    hash_ref r; (*written in C*)
    print_int !r
```

**C** business logic

```c
void hash_ptr(int * x) {
    // Implemented in OpenSSL
    // tedious to port to OCaml
}
```

**C** glue code

```c
value caml_hash_ref(value r) {
    int x = Int_val(Field(r, 0));
    hash_ptr(&x);
    Store_field(r, 0, Val_int(x));
    return Val_unit;
}
```
A Multi-Language Program in OCaml and C

**OCaml** business logic

```ocaml
let main () =
  let r = ref 42 in
hash_ref r; (*written in C*)
print_int !r
```

**C** business logic

```c
void hash_ptr(int * x) {
  // Implemented in OpenSSL
  // tedious to port to OCaml
}
```

**OCaml** glue code

```ocaml
external hash_ref: int ref -> unit = "caml_hash_ref"
```

**C** glue code

```c
value caml_hash_ref(value r) {
  int x = Int_val(Field(r, 0));
  hash_ptr(&x);
  Store_field(r, 0, Val_int(x));
  return Val_unit;
}
```
A multi-language program logic for FFI

Goal: a **program logic** to prove correctness of FFI glue code

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit
  = "caml_hash_ref"
```

**C glue code**

```c
value caml_hash_ref(value r) {
  int x = Int_val(Field(r, 0));
  hash_ptr(&x);
  Store_field(r, 0, Val_int(x));
  return Val_unit;
}
```
A multi-language program logic for FFI

Goal: a **program logic** to prove correctness of FFI glue code

**OCaml glue code**

\[
\begin{aligned}
\{ r \mapsto_{\text{ML}} n \} \\
\text{external } \text{hash\_ref}: \text{int ref } \to \text{unit} \\
= "\text{caml\_hash\_ref}"
\{ r \mapsto_{\text{ML}} m \}
\end{aligned}
\]

**C glue code**

\[
\begin{aligned}
\{ \gamma \mapsto \text{blk}[0|\text{mut}] [n] \} \\
\text{value } \text{caml\_hash\_ref}(\text{value } r) \{ \\
\text{int } x = \text{Int\_val}(\text{Field}(r, 0)); \\
\text{hash\_ptr}(&x); \\
\text{Store\_field}(r, 0, \text{Val\_int}(x)); \\
\text{return } \text{Val\_unit};
\}
\{ \gamma \mapsto \text{blk}[0|\text{mut}] [m] \}
\end{aligned}
\]
Reuse existing program logics for OCaml and C.

Outside of glue code, one can forget about other languages.

Key Design Choice: Preserve Language-Local Reasoning
Our Contribution: Melocoton

<table>
<thead>
<tr>
<th>$\lambda_{ML+C}$ Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue Code Verification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\lambda_{ML+C}$ Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue Code Semantics</td>
</tr>
</tbody>
</table>

“The Usual Approach”: program logic on top of semantics, **but**

- **Language Interaction**: new semantics and logic for glue code
Our Contribution: Melocoton

<table>
<thead>
<tr>
<th>OCaml* Program Logic</th>
<th>$\lambda_{ML+C}$ Program Logic</th>
<th>C* Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue Code Verification</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OCaml* Semantics</th>
<th>$\lambda_{ML+C}$ Semantics</th>
<th>C* Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue Code Semantics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“The Usual Approach”: program logic on top of semantics, but

- **Language Interaction**: new semantics and logic for glue code
- **Language Locality**: embed existing existing semantics and logics

---

* simplified/idealized versions of OCaml and C
Our Contribution: Melocoton

<table>
<thead>
<tr>
<th>OCaml* Program Logic</th>
<th>(\lambda_{ML+C}) Program Logic</th>
<th>C* Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCaml* Semantics</td>
<td>(\lambda_{ML+C}) Semantics</td>
<td>C* Semantics</td>
</tr>
<tr>
<td>Glue Code Verification</td>
<td>Glue Code Semantics</td>
<td></td>
</tr>
</tbody>
</table>

“The Usual Approach”: program logic on top of semantics, but

- **Language Interaction**: new semantics and logic for glue code
- **Language Locality**: embed existing semantics and logics

* simplified/idealized versions of OCaml and C
1. A Crash Course on Building Separation Logics

2. Key Idea: Bridging Separation Logics with View Reconciliation

3. Application: Verifying hash_ref
Melocoton is based on **Separation Logic**

...but what is **Separation Logic**?

**A Crash Course on (Building) Separation Logics**
A logic for *compositional* program verification

Establishes “Hoare triples”: \[ \vdash \{P\} \ C \ \{Q\} \]

Compositional proof rules:

\[
\text{SEQ} \quad \frac{\{P\} \ C_1 \ \{Q\} \quad \{Q\} \ C_2 \ \{R\}}{\{P\} \ C_1; \ C_2 \ \{R\}} \quad \ldots
\]
Hoare Logic

A logic for *compositional* program verification

Establishes “Hoare triples”:

\[ \vdash \{ P \} \ C \ \{ Q \} \]

Code we are verifying

Compositional proof rules:

\[
\text{SEQ} \quad \frac{\{ P \} \ C_1 \ \{ Q \} \quad \{ Q \} \ C_2 \ \{ R \}}{\{ P \} \ C_1; C_2 \ \{ R \}} \]

\[ \ldots \]
Hoare Logic

A logic for compositional program verification

Establishes “Hoare triples”:

$\vdash \{P\} C \{Q\}$

Precondition

Compositional proof rules:

**SEQ**

$\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}

\{P\} C_1; C_2 \{R\}

\ldots$
Hoare Logic

A logic for \textit{compositional} program verification

Establishes “Hoare triples”: \[ \vdash \{ P \} \ C \ \{ Q \} \]

Postcondition

Compositional proof rules:

\[ \text{SEQ} \]
\[
\begin{array}{c}
\{ P \} \ C_1 \ \{ Q \} \\
\{ Q \} \ C_2 \ \{ R \} \\
\{ P \} \ C_1; C_2 \ \{ R \}
\end{array}
\]

\[ \ldots \]
Separation Logic

- Extension of Hoare Logic for reasoning about pointer-manipulating programs (C, OCaml, …)

- Assertions $P, Q$ denote **ownership** of state (=memory)
  
  $\rightarrow \ "x \mapsto v" = \ We\ own\ x\ (and\ it\ points\ to\ v)$

- $P \ast Q$ means $P$ and $Q$ own **disjoint** state
  
  $\rightarrow \ e.g.\ if\ we\ can\ assert\ x \mapsto v \ast y \mapsto w,$

  it means that $x \neq y$, i.e. **they do not alias**
Separation Logic for OCaml: Example Rules

“$r \mapsto v$” = We own the OCaml reference $r$ (and it points to $v$)

CREATEREF

\[ \frac{}{\{\text{True}\} \text{ ref } v \{\lambda r. \, r \mapsto v\}} \]

READREF

\[ \frac{}{\{r \mapsto v\} !r \{\lambda v'. \, v' = v \land r \mapsto v\}} \]

WRITEREF

\[ \frac{}{\{r \mapsto v\} \ r := w \{r \mapsto w\}} \]
“r \leftrightarrow v” = We own the OCaml reference r (and it points to v)

CREATEREF

\[ \{\text{True}\} \text{ref } v \{\lambda r. \ r \leftrightarrow v\} \]

program creating a new reference

READREF

\[ \{r \leftrightarrow v\} !r \{\lambda v'. \ v' = v \land r \leftrightarrow v\} \]

WRITEREF

\[ \{r \leftrightarrow v\} r := w \{r \leftrightarrow w\} \]
“$r \mapsto v$” = We own the OCaml reference $r$ (and it points to $v$)

**CREATEREF**

\[
\frac{}{\{\text{True}\} \text{ ref } v \{\lambda r. \ r \mapsto v\}}
\]

No precondition

**READREF**

\[
\frac{}{\{r \mapsto v\} \! r \{\lambda v'. \ v' = v \land r \mapsto v\}}
\]

**WRITEREF**

\[
\frac{}{\{r \mapsto v\} r := w \{r \mapsto w\}}
\]
Separation Logic for OCaml: Example Rules

“\( r \mapsto v \)” = We own the OCaml reference \( r \) (and it points to \( v \))

\[
\text{CREATERef} \quad \begin{array}{l}
\{ \text{True} \} \ r \ref v \{ \lambda r. \ r \mapsto v \}
\end{array}
\]

Ownership over the new reference

\[
\text{READRef} \quad \{ r \mapsto v \} !r \{ \lambda v'. \ v' = v \land r \mapsto v \}
\]

\[
\text{WRITERef} \quad \{ r \mapsto v \} r := w \{ r \mapsto w \}
\]
Separation Logic: The Frame Rule

Hoare Logic is compositional wrt. different parts of a program.

Separation Logic is also compositional wrt. disjoint parts of memory. SL specifications are “small footprint”.

The following holds:

\[
\frac{\{\text{True}\} \text{ ref } v \{\lambda r. \ r \mapsto v\}}{\{x \mapsto w\} \text{ ref } v \{\lambda r. \ x \mapsto w \ast r \mapsto v\}}
\]
Separation Logic: The Frame Rule

Hoare Logic is compositional wrt. different parts of a program.

Separation Logic is also compositional wrt. disjoint parts of memory. SL specifications are “small footprint”.

More generally, the frame rule holds:

\[
\text{\textsc{frame}} \quad \frac{\{ P \} e \{ Q \}}{\{ P \ast R \} e \{ Q \ast R \}}
\]
Fifty Shades of Separation Logics

These core principles are very versatile.

Melocoton: a new SL for the OCaml FFI, embedding existing SLs for OCaml and C

Other SLs successfully built for many programming languages:

- Multicore OCaml
- Rust
- C11+Weak Memory
- WASM
- Distributed systems
- ...
A Separation Logic For Your Language: Checklist

- **A Base Logic** of assertions:
  - generic connectives:
    \[ \exists/\forall x. P(x), P \lor Q, P \land Q, P \Rightarrow Q, P \ast Q, P \dashv\vdash Q, \ldots \]
  - language-specific assertions: \[ x \mapsto v \]

- Triples \( \vdash \{ P \} \; e \; \{ Q \} \) + FRAME + **proof rules** for language constructs
A Separation Logic For Your Language: Checklist

- **A Base Logic** of assertions:
  - generic connectives:
    \( \exists / \forall x. P(x), \, P \lor Q, \, P \land Q, \, P \Rightarrow Q, \, P \ast Q, \, P \triangleright Q, \ldots \)
  - language-specific assertions: \( x \mapsto v \)

- Triples \( \vdash \{ P \} \, e \, \{ Q \} + \text{FRAME} + \text{proof rules for language constructs} \)
A Separation Logic For Your Language: Checklist

- **A Base Logic** of assertions:
  - generic connectives: 
    \[ \exists x P(x), P \lor Q, P \land Q, P \Rightarrow Q, P \ast Q, P \dashv\vdash Q, \ldots \]
  - language-specific assertions: \[ x \mapsto v \]

- Triples \( \vdash \{P\} e \{Q\} \) + FRAME + **proof rules** for language constructs

- An **Adequacy Theorem**: \( \vdash \{\text{True}\} e \{\text{True}\} \Rightarrow e \) is safe.
  “safe”: according to the **language semantics**
Prove Adequacy by defining a **model** of:

base assertions: $\llbracket P \rrbracket$  
validity of triples: $\models \{P\} e \{Q\}$

such that:

$\vdash \{P\} e \{Q\} \iff \models \{P\} e \{Q\}$  
(the hard part)

$\models \{\text{True}\} e \{\text{True}\} \implies e \text{ is safe}$  
(trivial by def. of $\models$)
Prove Adequacy by defining a **model** of:

- **base assertions:** $\llbracket P \rrbracket$
- **validity of triples:** $\vdash \{ P \} e \{ Q \}$

such that:

- $\vdash \{ P \} e \{ Q \} \implies \models \{ P \} e \{ Q \}$ (the hard part)
- $\models \{ \text{True} \} e \{ \text{True} \} \implies e \text{ is safe}$ (trivial by def. of $\models$)

similar to Hoare logic
Models of Separation Logic

Prove Adequacy by defining a \textbf{model} of:

- \text{base assertions: } \llbracket P \rrbracket
- \text{validity of triples: } \models \{ P \} e \{ Q \}

\[ \models \{ P \} e \{ Q \} \triangleq \forall \sigma. \llbracket P \rrbracket(\sigma) \Rightarrow e \text{ safe } \land \forall v\sigma'. (e, \sigma) \rightsquigarrow^* (v, \sigma') \Rightarrow \llbracket Q(v) \rrbracket(\sigma') \]

similar to Hoare logic
Models of Separation Logic

Prove Adequacy by defining a **model** of:

- **base assertions**: $\llbracket P \rrbracket$
- **validity of triples**: $\models \{ P \} e \{ Q \}$

\[
\models \{ P \} e \{ Q \} \triangleq \forall \sigma. \llbracket P \rrbracket(\sigma) \Rightarrow
\]

\[
e \text{ safe} \land \forall \upsilon \sigma'. (e, \sigma) \rightsquigarrow^* (\upsilon, \sigma') \Rightarrow \llbracket Q(\upsilon) \rrbracket(\sigma')
\]

**operational semantics**

*similar to Hoare logic*
Models of Separation Logic

Prove Adequacy by defining a model of:

- base assertions: $\models [P]$
- validity of triples: $\models \{P\} e \{Q\}$

\[ \models \{P\} e \{Q\} \triangleq \forall \sigma. [P](\sigma) \Rightarrow e \text{ safe} \land \forall v\sigma'. (e, \sigma) \rightsquigarrow^* (v, \sigma') \Rightarrow [Q(v)](\sigma') \]

similar to Hoare logic
The **interesting part**: interpretation of base assertions $[P]$

In general: $[P] : R \rightarrow \text{Prop}$

with $R$ **any** *Partial Commutative Monoid* equipped with $\oplus$

\[
[\text{True}](r) \triangleq \text{True} \\
[P \times Q](r) \triangleq \exists r_1, r_2. r = r_1 \oplus r_2 \land [P](r_1) \land [Q](r_2)
\]

For OCaml, pick $R = \text{Loc} \overset{\text{fin}}{\rightarrow} \text{Val}$:

\[
[\ell \mapsto v](\sigma) \triangleq \ell \in \text{dom}(\sigma) \land \sigma(\ell) = v
\]
Models of Separation Logics: Base Logic

The **interesting part**: interpretation of base assertions \([P]_R\)

In general: \([P]_R + R \rightarrow \text{Prop}\)

You are free to pick any PCM \(R\) that:

- is customized for your language semantics
- includes extra resources (“ghost state”) not directly tied to program execution

For OCaml, pick \(R = \text{Loc} \overset{\text{fin}}{\rightarrow} \text{Val}\):

\[
[P]_R(\ell \mapsto v)(\sigma) \overset{\Delta}{=} \ell \in \text{dom}(\sigma) \land \sigma(\ell) = v
\]
Iris: a Framework for Building Separation Logics

Iris provides SL building blocks as **reusable, language agnostic** Coq libraries:

- **expressive base logic** parameterized by an arbitrary PCM
- modular **pre-built PCMs**
- predefined **triples** and their **adequacy theorem**

**Most steps of the Checklist become** `Require Import iris`!
Bridging Separation Logics with View Reconciliation
Bridging Separation Logics with View Reconciliation
Melocoton brings the Checklist to a multi-language setting

**OCaml SL**
\[
\{P\} e_{\text{ML}} \{Q\}
\]
\[
\gamma \mapsto_{\text{ML}} v
\]

**FFI SL**
\[
\{P\} e_{\text{FFI}} \{Q\}
\]
\[
\gamma \mapsto_{\text{blk[0|mut]}} \text{blk}
\]

**C SL**
\[
\{P\} e_{\text{C}} \{Q\}
\]
\[
a \mapsto_{\text{C}} w
\]

**OCaml semantics**
\[
(e_{\text{ML}}, \sigma) \leadsto (e_{\text{ML}}, \sigma)
\]

**FFI semantics**
\[
(e_{\text{FFI}}, \rho) \leadsto (e_{\text{FFI}}, \rho)
\]

**C semantics**
\[
(e_{\text{C}}, m) \leadsto (e_{\text{C}}, m)
\]

**Problem:** how do we connect the different languages/logics?
Language Interaction: Different Views of the Same Data

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit
  = "caml_hash_ref"
```

**C glue code**

```c
value caml_hash_ref(value r) {
    int x = Int_val(Field(r, 0));
    hash_ptr(&x);
    Store_field(r, 0, Val_int(x));
    return Val_unit;
}
```

How is **OCaml** data accessed from **C glue code**?
OCaml glue code

```ocaml
external hash_ref: int ref -> unit = "caml_hash_ref"
```

C glue code

```c
value caml_hash_ref(value r) {
    int x = Int_val(Field(r, 0));
    hash_ptr(&x);
    Store_field(r, 0, Val_int(x));
    return Val_unit;
}
```

How is OCaml data accessed from C glue code?

High-level OCaml values are accessed through a low-level block representation.
High-level OCaml value $\sim_{\text{ML}}$ Low-level block representation
High-level \textbf{OCaml} value $\sim_{\text{ML}}$ Low-level \textbf{block} representation

integers $\sim_{\text{ML}}$ integers

booleans $\sim_{\text{ML}}$ integers (0 or 1)

\texttt{true} $\sim_{\text{ML}}$ 1
High-level OCaml value $\sim_{\text{ML}}$ Low-level block representation

- integers $\sim_{\text{ML}}$ integers
- booleans $\sim_{\text{ML}}$ integers (0 or 1)
- arrays, refs $\sim_{\text{ML}}$ blocks

$\text{true} \sim_{\text{ML}} 1$

$l \sim_{\text{ML}} \gamma$
High-level **OCaml** value $\sim_{\text{ML}}$ Low-level **block** representation

- integers $\sim_{\text{ML}}$ integers
- booleans $\sim_{\text{ML}}$ integers (0 or 1)
- arrays, refs $\sim_{\text{ML}}$ blocks
- pairs $\sim_{\text{ML}}$ blocks (of size 2)

true $\sim_{\text{ML}}$ 1

$\ell \sim_{\text{ML}} \gamma$
Language Interaction: Semantics

High-level **OCaml** value $\sim_{\text{ML}}$ Low-level **block** representation

- integers $\sim_{\text{ML}}$ integers
- booleans $\sim_{\text{ML}}$ integers (0 or 1)
- arrays, refs $\sim_{\text{ML}}$ blocks
- pairs $\sim_{\text{ML}}$ blocks (of size 2)
- lists $\sim_{\text{ML}}$ block-based linked lists
Language Interaction: Semantics

High-level **OCaml** value $\sim_{ML}$ Low-level **block** representation

- integers $\sim_{ML}$ integers
- booleans $\sim_{ML}$ integers (0 or 1)
- arrays, refs $\sim_{ML}$ blocks
- pairs $\sim_{ML}$ blocks (of size 2)
- lists $\sim_{ML}$ block-based linked lists

**$\lambda_{ML+C}$ Semantics**

$\sigma : \text{Heap}_{ML}$

$\zeta : \text{BlockHeap}$
Language Interaction: Semantics

High-level **OCaml** value $\sim_{\text{ML}}$ Low-level block representation

- integers $\sim_{\text{ML}}$ integers
- booleans $\sim_{\text{ML}}$ integers (0 or 1)
- arrays, refs $\sim_{\text{ML}}$ blocks
- pairs $\sim_{\text{ML}}$ blocks (of size 2)
- lists $\sim_{\text{ML}}$ block-based linked lists

$\lambda_{\text{ML+C}}$ Semantics

\[ \sigma : \text{Heap}_{\text{ML}} \leftrightarrow \zeta : \text{BlockHeap} \]

switch at the language barrier
Language Interaction: Semantics

High-level **OCaml** value \(\sim_{ML}\) Low-level **block** representation

- integers \(\sim_{ML}\) integers
- booleans \(\sim_{ML}\) integers (0 or 1)
- arrays, refs \(\sim_{ML}\) blocks
- pairs \(\sim_{ML}\) blocks (of size 2)
- lists \(\sim_{ML}\) block-based linked lists

\[\text{switch at the language barrier}\]
Language Interaction: Semantics

High-level **OCaml** value $\sim_{ML}$ Low-level **block** representation

- integers $\sim_{ML}$ integers
- booleans $\sim_{ML}$ integers (0 or 1)
- arrays, refs $\sim_{ML}$ blocks
- pairs $\sim_{ML}$ blocks (of size 2)
- lists $\sim_{ML}$ block-based linked lists

$$\lambda_{ML+C} \textbf{Semantics}$$

$\sigma : \texttt{Heap}_{ML}$

switch at the language barrier

$\zeta : \texttt{BlockHeap}$
Language Interaction: Semantics

High-level **OCaml** value $\sim_{\text{ML}}$ Low-level **block** representation

- integers $\sim_{\text{ML}}$ integers
- booleans $\sim_{\text{ML}}$ integers (0 or 1)
- arrays, refs $\sim_{\text{ML}}$ blocks
- pairs $\sim_{\text{ML}}$ blocks (of size 2)
- lists $\sim_{\text{ML}}$ block-based linked lists

\[ \lambda_{\text{ML+C}} \textbf{Semantics} \]

\[ \sigma : \text{Heap}_{\text{ML}} \quad \vdash \quad \zeta : \text{BlockHeap} \]

switch at the language barrier
The \( \lambda_{ML+C} \) Semantics operate globally on the state \( E \). The \( \lambda_{ML+C} \) Program Logic needs local reasoning rules.
Language Interaction: Program Logic, Take 1

\( \lambda_{ML+C} \) Program Logic

\[ \sigma : \text{Heap}_{ML} \quad \xrightarrow{\lambda_{ML+C} \text{ Semantics}} \quad \zeta : \text{BlockHeap} \]
The \( \lambda_{ML+C} \) Semantics operate globally on the state \( E \). The \( \lambda_{ML+C} \) Program Logic needs local reasoning rules.
The $\lambda_{ML+C}$ Semantics operate globally on the state $E$. The $\lambda_{ML+C}$ Program Logic needs local reasoning rules.
The λ_{ML+C} Semantics operate globally on the state \( E \). The λ_{ML+C} Program Logic needs local reasoning rules.
The $\lambda_{ML+C}$ Semantics operate globally on the state $E$

The $\lambda_{ML+C}$ Program Logic needs local reasoning rules
Language Interaction: Program Logic, Take 1

\begin{itemize}
  \item All the \( \ell \mapsto_{\text{ML}} \overrightarrow{V} \)
  \item \( \lambda_{\text{ML}+C} \text{ Program Logic} \)
  \item All the \( \gamma \mapsto_{\text{blk}} \overrightarrow{v} \)
  \item \( \sigma : \text{Heap}_{\text{ML}} \)
  \item \( \lambda_{\text{ML}+C} \text{ Semantics} \)
  \item \( \zeta : \text{BlockHeap} \)
\end{itemize}
Language Interaction: Program Logic, Take 1

$\lambda_{\text{ML+C}}$ Program Logic

$\lambda_{\text{ML+C}}$ Semantics

$\sigma : \text{Heap}_{\text{ML}}$

$\gamma \mapsto \text{blk} \ V$

$\zeta : \text{BlockHeap}$

$\ell \mapsto_{\text{ML}} V$

EXTCALL

{all} C function body {all}

{all} call into C {all}
The $\lambda_{\text{ML+C}}$ Semantics operate globally on the state $E$. The $\lambda_{\text{ML+C}}$ Program Logic needs local reasoning rules.
The $\lambda_{ML+C}$ Semantics operate globally on the state $E$. The $\lambda_{ML+C}$ Program Logic needs local reasoning rules.
The $\lambda_{ML+C}$ Semantics operate globally on the state $E$. The $\lambda_{ML+C}$ Program Logic needs local reasoning rules.

**Language Interaction: Program Logic, Take 1**

**EXTCALL**
{all} C function body {all}
{all} call into C {all}

**FRAME**
{P} call into C {Q}
{\ell \mapsto_{ML} \vec{V} \ast P} call into C {Q \ast \ell \mapsto_{ML} \vec{V}}
Language Interaction: Program Logic, Take 1

The \( \lambda_{ML+C} \) Program Logic operates globally on the state \( E \). The \( \lambda_{ML+C} \) Program Logic needs local reasoning rules.

EXTCALL

\[
\{ \text{all} \} \text{C function body} \{ \text{all} \}
\begin{align*}
\{ \text{all} \} & \text{call into C} \{ \text{all} \}
\end{align*}
\]

FRAME

\[
\{ P \} \text{call into C} \{ Q \}
\begin{align*}
\{ \ell \mapsto_{ML} \vec{V} \ast P \} & \text{call into C} \{ Q \ast \ell \mapsto_{ML} \vec{V} \}
\end{align*}
\]
Language Interaction: Program Logic, Take 1

$\lambda_{ML+C}$ Program Logic

$\lambda_{ML+C}$ Semantics

EXTCALL

{ all } C function body { all } call into C { all }

{ all } call into C { all }

{ P } call into C { Q }

{ $\ell \mapsto_{ML} \vec{V} \ast P$ } call into C { $Q \ast \ell \mapsto_{ML} \vec{V}$ }

RAME

26
The $\lambda_{\text{ML+C}}$ Semantics operate **globally** on the state

The $\lambda_{\text{ML+C}}$ Program Logic needs **local** reasoning rules
OCaml points-tos remain valid when switching to C!
**OCaml** points-tos *remain valid* when switching to C!
OCaml points-tos remain valid when switching to C!
OCaml points-tos *remain valid* when switching to C!
**OCaml** points-tos *remain valid* when switching to **C**!
OCaml points-tos remain valid when switching to C!

\[ \gamma_2 \mapsto_{\text{blk}} \vec{v}_2 \]

\[ \ell \mapsto_{\text{ML}} \vec{V} \]
OCaml points-tos remain valid when switching to C!
OCaml points-tos remain valid when switching to C!

\[
\ell \mapsto_{\text{ML}} \vec{V}
\]

View Reconciliation Rules for Converting On-Demand:

\[
\ell \mapsto_{\text{ML}} \vec{V} \implies \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} \ast \ell \sim_{\text{ML}} \gamma \ast \vec{V} \sim_{\text{ML}} \vec{v}
\]
\[
\vec{V} \sim_{\text{ML}} \vec{v} \ast \gamma \mapsto_{\text{blk}} \vec{v} \implies \exists \ell . \ell \mapsto_{\text{ML}} \vec{V} \ast \ell \sim_{\text{ML}} \gamma
\]
Language Interaction: View Reconciliation

**View Reconciliation Rules**

\[ \ell \rightarrow_{ML} \vec{V} \equiv \exists \gamma \vec{v}. \gamma \rightarrow_{blk} \vec{v} \ast \ell \sim_{ML} \gamma \ast \vec{V} \sim_{ML} \vec{v} \]

\[ \vec{V} \sim_{ML} \vec{v} \ast \gamma \rightarrow_{blk} \vec{v} \equiv \exists \ell. \ell \rightarrow_{ML} \vec{V} \ast \ell \sim_{ML} \gamma \]
Language Interaction: View Reconciliation

View Reconciliation Rules

\[ \ell \mapsto_{ML} \vec{V} \equiv \exists \gamma \vec{v}. \gamma \mapsto_{blk} \vec{v} * \ell \sim_{ML} \gamma * \vec{V} \sim_{ML} \vec{v} \]

\[ \vec{V} \sim_{ML} \vec{v} * \gamma \mapsto_{blk} \vec{v} \equiv \exists \ell. \ell \mapsto_{ML} \vec{V} * \ell \sim_{ML} \gamma \]

\[\lambda_{ML+C} \text{ Program Logic}\]

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

all the \( \ell \mapsto_{ML} \vec{V} \)

\(\sigma : Heap_{ML}\)

all the \( \gamma \mapsto_{blk} \vec{v} \)

\(\zeta : BlockHeap\)
Language Interaction: View Reconciliation

View Reconciliation Rules

\[ \ell \mapsto_{ML} \vec{V} \quad \Rightarrow\quad \exists \gamma \vec{v}. \, \vec{v} \mapsto_{blk} \vec{v} \ast \ell \sim_{ML} \gamma \ast \vec{V} \sim_{ML} \vec{v} \]

\[ \vec{V} \sim_{ML} \vec{v} \ast \gamma \mapsto_{blk} \vec{v} \quad \Rightarrow\quad \exists \ell . \, \ell \mapsto_{ML} \vec{V} \ast \ell \sim_{ML} \gamma \]

\[ \lambda_{ML+C} \text{ Program Logic} \]

all the \[ \ell \mapsto_{ML} \vec{V} \]

all the \[ \gamma \mapsto_{blk} \vec{v} \]

\[ \sigma : \text{Heap}_{ML} \]

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

\[ \zeta : \text{BlockHeap} \]
Language Interaction: View Reconciliation

View Reconciliation Rules

\[ \ell \mapsto_{\text{ML}} \vec{V} \equiv \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} \times \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \]

\[ \vec{V} \sim_{\text{ML}} \vec{v} \times \gamma \mapsto_{\text{blk}} \vec{v} \equiv \exists \ell . \ell \mapsto_{\text{ML}} \vec{V} \times \ell \sim_{\text{ML}} \gamma \]

\[
\begin{align*}
\lambda_{\text{ML+C}} \quad \text{Program Logic} \\
\sigma_{\text{ghost}} : \text{Heap}_{\text{ML}} \\
\zeta : \text{BlockHeap}
\end{align*}
\]

\[
\begin{align*}
\lambda_{\text{ML+C}} \quad \text{Semantics} \\
\sigma : \text{Heap}_{\text{ML}} \\
\end{align*}
\]
Language Interaction: View Reconciliation

**View Reconciliation Rules**

\[
\ell \mapsto_{\text{ML}} \vec{V} \iff \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} \ast \ell \sim_{\text{ML}} \gamma \ast \vec{V} \sim_{\text{ML}} \vec{v}
\]

\[
\vec{V} \sim_{\text{ML}} \vec{v} \ast \gamma \mapsto_{\text{blk}} \vec{v} \iff \exists \ell . \ell \mapsto_{\text{ML}} \vec{V} \ast \ell \sim_{\text{ML}} \gamma
\]

\[\sigma_{\text{ghost}} : \text{Heap}_{\text{ML}}\]

\[
\zeta : \text{BlockHeap}
\]

\[\lambda_{\text{ML+C}} \text{ Program Logic}\]

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

\[\sigma : \text{Heap}_{\text{ML}}\]

all the \(\ell \mapsto_{\text{ML}} \vec{V}\)

all the \(\gamma \mapsto_{\text{blk}} \vec{v}\)
View Reconciliation Rules

\[
\ell \mapsto_{\text{ML}} \vec{V} \equiv \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} \ast \ell \sim_{\text{ML}} \gamma \ast \vec{V} \sim_{\text{ML}} \vec{v}
\]

\[
\vec{V} \sim_{\text{ML}} \vec{v} \ast \gamma \mapsto_{\text{blk}} \vec{v} \equiv \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} \ast \ell \sim_{\text{ML}} \gamma
\]
Application: Verifying `hash_ref` with Melocoton
Verifying hash_ref with Melocoton

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit
  = "caml_hash_ref"
```

**C glue code**

```c
value caml_hash_ref(value v) {
  int x = Int_val(Field(v, 0));
  hash_ptr(&x);
  Store_field(v, 0, Val_int(x));
  return Val_unit;
}
```
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref : int ref -> unit = "caml_hash_ref"
  {r ↦₁ ML n}
  hash_ref(r)
  {∃m. r ↦₁ ML m}
```

**C glue code**

```c
value caml_hash_ref(value v) {
    int x = Int_val(Field(v, 0));
    hash_ptr(&x);
    Store_field(v, 0, Val_int(x));
    return Val_unit;
}
```
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit = "caml_hash_ref"

{r ↦ₚ n}
hash_ref(r)
{∃m. r ↦ₚ m}
```

**C glue code**

```c
value caml_hash_ref(value v) {
    int x = Int_val(Field(v, 0));
    hash_ptr(&x);
    Store_field(v, 0, Val_int(x));
    return Val_unit;
}
```

**Formal Explanation**

\[
\text{EXTCALL} \quad \begin{cases}
P \times x \simₚ v \quad f(v) \{ \lambda v'. \exists y. y \simₚ v' \times Q(y) \}
\end{cases}
\]

\[
\{ P \} \text{ external } "f"(x) \{ \lambda y. Q(y) \}
\]
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit = "caml_hash_ref"
  {r \mapsto_{ML} n}
  hash_ref(r)
  {\exists m. r \mapsto_{ML} m}
```

**C glue code**

```c
value caml_hash_ref(value v) {
  {r \mapsto_{ML} n \ast r \sim_{ML} v}
  int x = Int_val(Field(v, 0));
  hash_ptr(&x);
  Store_field(v, 0, Val_int(x));
  return Val_unit;
  {\exists m. r \mapsto_{ML} m \ast \exists y. y \sim_{ML} Val_unit}
}
```

**ExtCall**

```latex
\[
\begin{array}{c}
\{ P \ast x \sim_{ML} v \} \ f(v) \ \{ \lambda v'. \exists y. y \sim_{ML} v' \ast Q(y) \} \\
\{ P \} \ \text{external} \ "f"(x) \ \{ \lambda y. Q(y) \}
\end{array}
\]```
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit
  = "caml_hash_ref"

  \{ r \mapsto_{\text{ML}} n \}\n  hash_ref(r)

  \{ \exists m. r \mapsto_{\text{ML}} m \}\n```

**C glue code**

```c
value caml_hash_ref(value v) {
  \{ r \mapsto_{\text{ML}} n * r \sim_{\text{ML}} v \}\n  int x = Int_val(Field(v, 0));
  hash_ptr(&x);
  Store_field(v, 0, Val_int(x));
  return Val_unit;

  \{ \exists m. r \mapsto_{\text{ML}} m * () \sim_{\text{ML}} Val_unit \}\n}
```

\[ \text{EXTCALL} \]

\[
\begin{align*}
\{ P \star x \sim_{\text{ML}} v \}\ f(v) \{ \lambda v'. \exists y. y \sim_{\text{ML}} v' * Q(y) \} \\
\{ P \} \text{external } "f"(x) \{ \lambda y. Q(y) \}\n\end{align*}
\]
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref: int ref -> unit = "caml_hash_ref"

\{ r \mapsto_{ML} n \}\n
hash_ref(r)

\{ \exists m. r \mapsto_{ML} m \}\n```

**C glue code**

```c
value caml_hash_ref(value v) {
  \{ r \mapsto_{ML} n * r \sim_{ML} v \}\n  \{ v \mapsto_{blk} [n] * r \sim_{ML} v \}\n  int x = Int_val(Field(v, 0));
  hash_ptr(&x);
  Store_field(v, 0, Val_int(x));
  return Val_unit;

  \{ \exists m. r \mapsto_{ML} m * () \sim_{ML} Val_unit \}\n}
```

**View Reconciliation (1)**

\[ \ell \mapsto_{ML} \vec{V} \iff \exists_{\gamma \vec{v}}. \gamma \mapsto_{blk} \vec{v} * \ell \sim_{ML} \gamma * \vec{V} \sim_{ML} \vec{v} \]
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref : int ref -> unit = "caml_hash_ref"
  {r ↦_{ML} n}
  hash_ref(r)
  {∃m. r ↦_{ML} m}
```

**C glue code**

```c
value caml_hash_ref(value v) {
  {r ↦_{ML} n ∗ r ∼_{ML} v}
  {v ↦_{blk} [n] ∗ r ∼_{ML} v}
  int x = Int_val(Field(v, 0));
  hash_ptr(&x);
  Store_field(v, 0, Val_int(x));
  return Val_unit;
  {∃m. v ↦_{blk} [m] ∗ r ∼_{ML} v}
  {∃m. r ↦_{ML} m ∗ () ∼_{ML} Val_unit}
}
```

**View Reconciliation (2)**

\[ \vec{V} \sim_{ML} \vec{v} \ast \gamma \mapsto_{blk} \vec{v} \ast \gamma \Rightarrow \exists \ell . \ell \mapsto_{ML} \vec{V} \ast \ell \sim_{ML} \gamma \]
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```
external hash_ref: int ref -> unit = "caml_hash_ref"

\{r \mapsto_{ML} n\}
hash_ref(r)
\{\exists m. r \mapsto_{ML} m\}
```

**C glue code**

```
value caml_hash_ref(value v) {
\{r \mapsto_{ML} n \ast r \sim_{ML} v\}
\{v \mapsto_{blk} [n] \ast r \sim_{ML} v\}

int x = Int_val(Field(v, 0));
hash_ptr(&x);
Store_field(v, 0, Val_int(x));
return Val_unit;

\{\exists m. v \mapsto_{blk} [m] \ast r \sim_{ML} v\}
\{\exists m. r \mapsto_{ML} m \ast () \sim_{ML} Val_unit\}
}
```

**Field SPECIFICATION**

```
\{\gamma \mapsto_{blk} [...v_i...]\} Field(\gamma, i) \{\lambda v'. v' = v_i \land \gamma \mapsto_{blk} [...v_i...]\}
```
Verifying `hash_ref` with Melocoton

**OCaml glue code**

```ocaml
external hash_ref : int ref -> unit
    = "caml_hash_ref"

    \{ r \mapsto_{\text{ML}} n \}\n
    hash_ref(r)

    \{ \exists m. r \mapsto_{\text{ML}} m \}\n```

**C glue code**

```c
value caml_hash_ref(value v) { 

    \{ r \mapsto_{\text{ML}} n \ast r \sim_{\text{ML}} v \}\n
    \{ v \mapsto_{\text{blk}} \lfloor n \rfloor \ast r \sim_{\text{ML}} v \}\n
    int x = Int_val(Field(v, 0));
    hash_ptr(&x);
    Store_field(v, 0, Val_int(x));
    return Val_unit;

    \{ \exists m. v \mapsto_{\text{blk}} \lfloor m \rfloor \ast r \sim_{\text{ML}} v \}\n
    \{ \exists m. r \mapsto_{\text{ML}} m \ast () \sim_{\text{ML}} \text{Val_unit} \}\n}
```

**Store_field SPECIFICATION**

```latex
\{ \gamma \mapsto_{\text{blk}} \ldots v_i \ldots \} \text{Store_field}(\gamma, i, v') \{ \gamma \mapsto_{\text{blk}} \ldots v' \ldots \}\n```
But wait, there is more!

- Language-local reasoning for **external calls**.
- Additional **OCaml FFI features**: garbage collection, registering roots, custom blocks, callbacks, etc.
- **Case studies** utilising all of these features.
- **Semantic model of OCaml types** (a logical relation) to verify type safety and encapsulation of FFI code.
But wait, there is more!

- Language-local reasoning for **external calls**.
- Additional **OCaml FFI features**: garbage collection, registering roots, custom blocks, callbacks, etc.
- **Case studies** utilising all of these features.
- **Semantic model of OCaml types** (a logical relation) to verify type safety and encapsulation of FFI code.
Future work

Extend Melocoton to model all of the OCaml FFI exceptions, multithreading, “zero-copy” operations on byte arrays, ...

Build code-analysis tools based on Melocoton

Allow OCaml programmers to check correctness of their FFI glue code
Ideally, both verification and bug finding tools
**Language Locality: Embed Existing Languages**

<table>
<thead>
<tr>
<th>OCaml Program Logic</th>
<th>( \lambda_{ML+C} ) <strong>Program Logic</strong></th>
<th>C Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue Code Verification</td>
<td>( \lambda_{ML+C} ) <strong>Semantics</strong></td>
<td>C Semantics</td>
</tr>
<tr>
<td>OCaml Semantics</td>
<td><strong>Semantics</strong></td>
<td></td>
</tr>
<tr>
<td>Glue Code Semantics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Language Interaction: View Reconciliation Rules**

\[
\ell \mapsto_{ML} \vec{V} \equiv * \exists \gamma \vec{v}. \gamma \mapsto_{blk} \vec{v} * \ell \sim_{ML} \gamma * \vec{V} \sim_{ML} \vec{v}
\]

\[
\vec{V} \sim_{ML} \vec{v} * \gamma \mapsto_{blk} \vec{v} \equiv * \exists \ell . \ell \mapsto_{ML} \vec{V} * \ell \sim_{ML} \gamma
\]

https://melocotong-project.github.io