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

Inria

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Multi-Language Programs Are Everywhere

NumPy

Python

C

Fortran

Firefox

C++

Rust

JavaScript

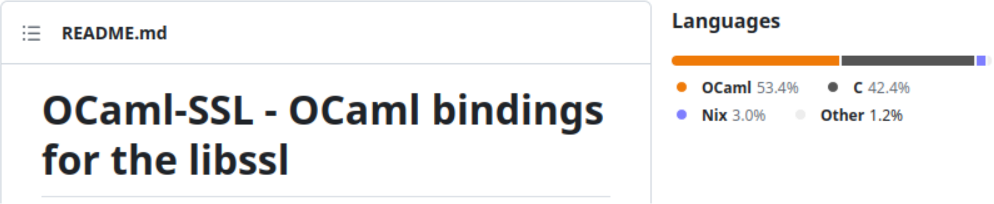
OpenSSL
Cryptography and SSL/TLS Toolkit

C

Bindings for:

- Rust
- Python
- OCaml
- Go
- ...

Multi-Language Programs Are Everywhere



README.md

OCaml-SSL - OCaml bindings for the libssl

Languages

● OCaml 53.4% ● C 42.4%
● Nix 3.0% ● Other 1.2%

a mixture of C and OCaml code
connected using the OCaml Foreign Function Interface (FFI)

- Go
- ...

Multi-Language Code is Unsafe

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

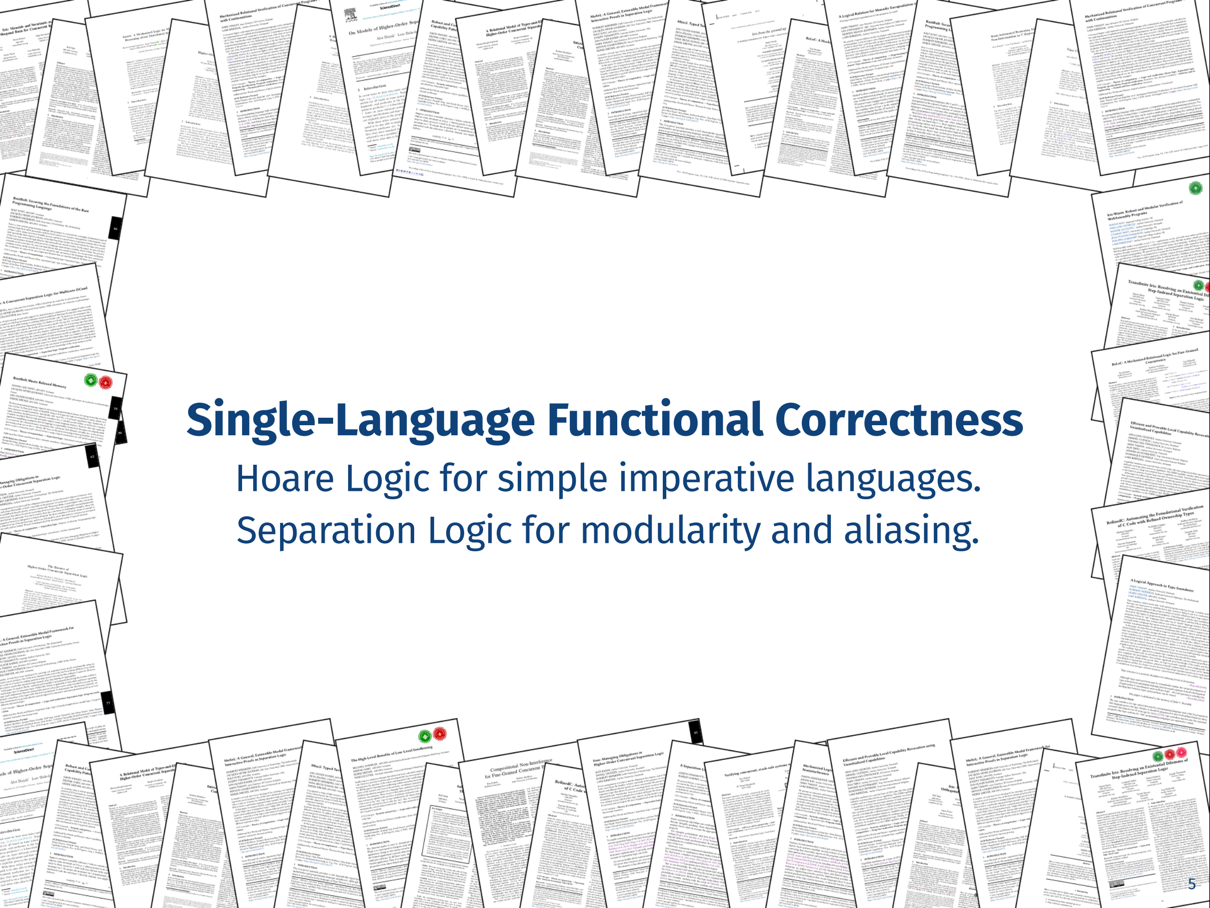
The Goal: Verifying Multi-Language Code

How do we

verify functional correctness

of code written in

different languages?

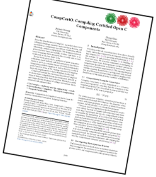


Single-Language Functional Correctness

Hoare Logic for simple imperative languages.

Separation Logic for modularity and aliasing.

Multi-Language Functional Correctness


CompCertO: Compiling Certified Open C Components

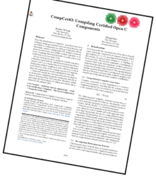

Verifying an Open Compiler Using Multi-language Semantics


Semantic Soundness for Language Interoperability


DimSum: A Decentralized Approach to Multi-lan
Semantics and Verification


perational Semantics for Multi-Language Programs


FunTAL: Reasonably Mixing a Functional Language with Assembly


CompCertO: Compiling Certified Open C Components


Verifying an Open Compiler Using Multi-language Semantics


Semantic Soundness for Language Interoperability

Multi-Language Functional Correctness

Existing work on Semantics and Logical Relations.

How do we prove functional correctness of individual, potentially unsafe libraries?


DimSum: A Decentralized Approach to Multi-lan
Semantics and Verification


perational Semantics for Multi-Language Programs


FunTAL: Reasonably Mixing a Functional Language with Assembly

A Multi-Language Program in OCaml and C

A Multi-Language Program in OCaml and C

C business logic

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

A Multi-Language Program in OCaml and C

OCaml business logic

C business logic

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

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

A Multi-Language Program in OCaml and C

OCaml business logic

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

C business logic

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

C glue code

```
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

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

C business logic

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

OCaml glue code

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

C glue code

```
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

C glue code

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

```
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

$\{r \mapsto_{\text{ML}} n\}$

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

$\{r \mapsto_{\text{ML}} m\}$

C glue code

$\{\gamma \mapsto_{\text{blk}[0|\text{mut}]} [n]\}$

```
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;
}
```

$\{\gamma \mapsto_{\text{blk}[0|\text{mut}]} [m]\}$

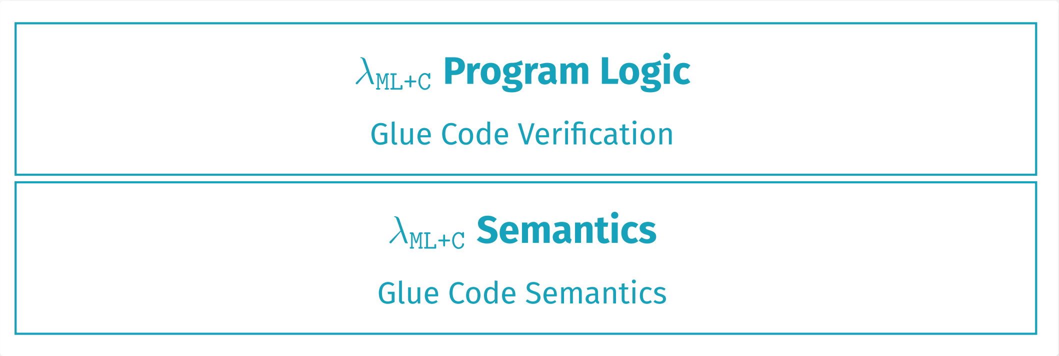
Design choice: Language-Local Reasoning

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



$\lambda_{\mathrm{ML+C}}$ **Program Logic**

Glue Code Verification

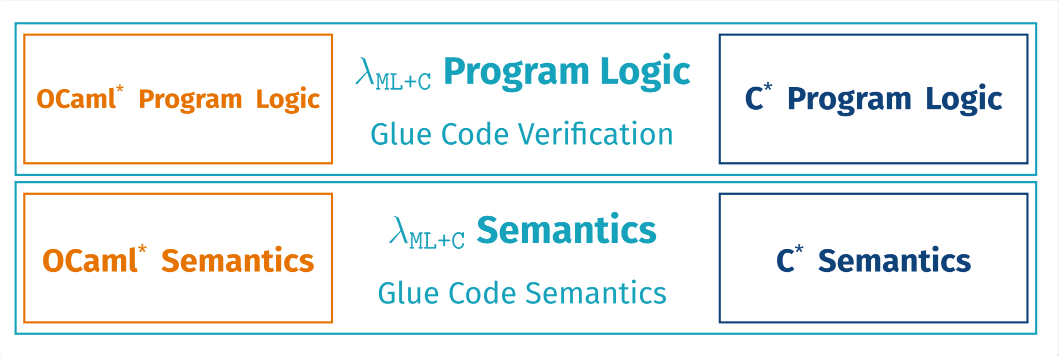
$\lambda_{\mathrm{ML+C}}$ **Semantics**

Glue Code Semantics

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

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

Our Contribution: Melocoton



OCaml[*] Program Logic

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

Glue Code Verification

C[*] Program Logic

OCaml[*] Semantics

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

Glue Code Semantics

C[*] Semantics

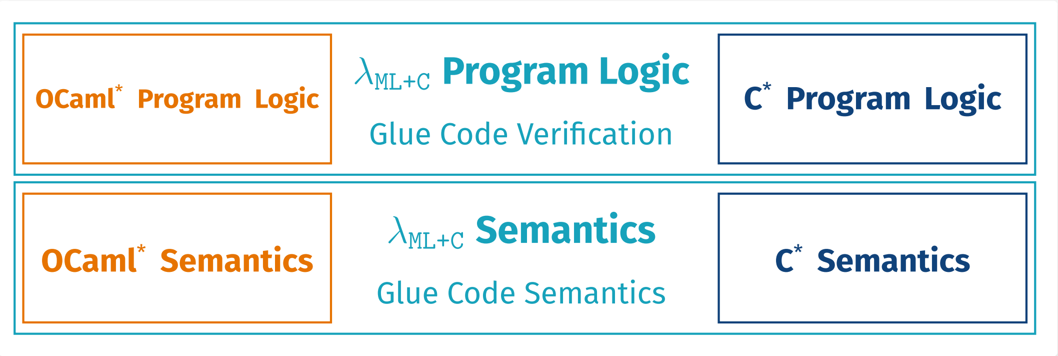
“The Usual Approach”: program logic on top of semantics, **but**

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

---

[*] simplified/idealized versions of **OCaml** and **C**

Our Contribution: Melocoton


OCaml* Program Logic

$\lambda_{\mathrm{ML+C}}$ **Program Logic**

Glue Code Verification

C* Program Logic

OCaml* Semantics

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

Glue Code Semantics

C* Semantics

"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

Transfinite Iris

---

*simplified/idealized versions of **OCaml** and **C**

The rest of this talk

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

Hoare Logic

A logic for *compositional* program verification

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

Compositional proof rules:

$$\textsc{Seq}\ \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}} \qquad \ldots$$

Hoare Logic

A logic for *compositional* program verification

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

Code we are verifying

Compositional proof rules:

$$\textsc{Seq}\ \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}} \qquad \ldots$$

Hoare Logic

A logic for *compositional* program verification

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

Precondition

Compositional proof rules:

$$\textsc{Seq}\quad \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}} \qquad \ldots$$

Hoare Logic

A logic for *compositional* program verification

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

Postcondition

Compositional proof rules:

Seq

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}} \qquad \ldots$$

Separation Logic

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

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

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

$$\frac{}{\{\mathrm{True}\}\ \textsf{ref}\ v\ \{\lambda r.\ r \mapsto v\}}$$

ReadRef

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

WriteRef

$$\frac{}{\{r \mapsto v\}\ r := w\ \{r \mapsto w\}}$$

Separation Logic for OCaml: Example Rules

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

CreateRef

$$\frac{}{\{\mathrm{True}\}\ \mathsf{ref}\ v\ \{\lambda r.\ r \mapsto v\}}$$

program creating a new reference

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)

CreateRef

$$\frac{}{\{\mathrm{True}\}\ \mathsf{ref}\ v\ \{\lambda r.\ r \mapsto v\}}$$

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)

$$\frac{\textsc{CreateRef}}{\{\mathrm{True}\}\ \mathsf{ref}\ v\ \{\lambda r.\ r \mapsto v\}}$$

Ownership over the new reference

$$\frac{\textsc{ReadRef}}{\{r \mapsto v\}\ !r\ \{\lambda v'.\ v' = v \land r \mapsto v\}}$$

$$\frac{\textsc{WriteRef}}{\{r \mapsto v\}\ r := w\ \{r \mapsto w\}}$$

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:

$$\frac{\overline{\{\mathrm{True}\}\ \mathsf{ref}\ v\ \{\lambda r.\ r \mapsto v\}}}{\{x \mapsto w\}\ \mathsf{ref}\ v\ \{\lambda r.\ x \mapsto w * r \mapsto v\}}$$

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:

$$\textsc{Frame}\ \frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}$$

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: $\exists/\forall x. P(x), P \vee Q, P \wedge Q, P \Rightarrow Q, P * Q, P \mathrel{-\!\!*} Q, \ldots$
    - 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 bunch of definitions

- A **Base Logic** of assertions:
  - generic connectives: $\exists/\forall x.P(x), P \vee Q, P \wedge Q, P \Rightarrow Q, P * Q, P \mathrel{-\!\!*} Q, \ldots$
  - 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 bunch of definitions

- A **Base Logic** of assertions:
  - generic connectives: $\exists/\forall x. P(x), P \vee Q, P \wedge Q, P \Rightarrow Q, P * Q, P \mathrel{-\!\!*} Q, \ldots$
  - language-specific assertions: $x \mapsto v$
- Triples $\vdash \{P\}\ e\ \{Q\}$ + FRAME + **proof rules** for language constructs

- An **Adequacy Theorem:** $\vdash \{\mathrm{True}\}\ e\ \{\mathrm{True}\} \Rightarrow e \text{ is safe.}$

“safe”: according to the language **semantics**

Models of Separation Logic

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

base assertions: $[\![P]\!]$ validity of triples: $\vDash \{P\}\, e\, \{Q\}$

such that:

$\vdash \{P\}\, e\, \{Q\} \implies \vDash \{P\}\, e\, \{Q\}$ (the hard part)

$\vDash \{\text{True}\}\, e\, \{\text{True}\} \implies e \text{ is safe}$ (trivial by def. of $\vDash$)

Models of Separation Logic

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

base assertions: $[\![P]\!]$

validity of triples: $\vDash \{P\}\, e\, \{Q\}$

such that:

$\vdash \{P\}\, e\, \{Q\} \implies \vDash \{P\}\, e\, \{Q\}$ (the hard part)

$\vDash \{\mathrm{True}\}\, e\, \{\mathrm{True}\} \implies e \text{ is safe}$ (trivial by def. of $\vDash$)

similar to Hoare logic

Models of Separation Logic

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

base assertions: $[\![P]\!]$

validity of triples: $\vDash \{P\}\, e\, \{Q\}$

$$\vDash \{P\}\, e\, \{Q\} \triangleq \forall \sigma.\, [\![P]\!](\sigma) \Rightarrow$$
$$e \text{ safe} \wedge \forall v \sigma'.\, (e, \sigma) \leadsto^{*} (v, \sigma') \Rightarrow [\![Q(v)]\!](\sigma')$$

similar to Hoare logic

Models of Separation Logic

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

base assertions: $[\![P]\!]$

validity of triples: $\vDash \{P\}\, e\, \{Q\}$

$$\vDash \{P\}\, e\, \{Q\} \triangleq \forall \sigma.\, [\![P]\!](\sigma) \Rightarrow \underbrace{e \text{ safe}} \wedge \forall v \sigma'.\, \underbrace{(e, \sigma) \leadsto^* (v, \sigma')} \Rightarrow [\![Q(v)]\!](\sigma')$$

operational semantics

similar to Hoare logic

Models of Separation Logic

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

base assertions: $[\![P]\!]$

validity of triples: $\vDash \{P\}\, e\, \{Q\}$

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

operational semantics

predicate on memories

similar to Hoare logic

Models of Separation Logics: Base Logic

The **interesting part**: interpretation of base assertions $[\![P]\!]$

In general: $[\![P]\!] : R \to \mathsf{Prop}$
with R **any** *Partial Commutative Monoid* equipped with $\uplus$

$$
\begin{array}{rcl}
[\![\mathrm{True}]\!](r) & \triangleq & \mathrm{True} \\
[\![P * Q]\!](r) & \triangleq & \exists r_1, r_2.\, r = r_1 \uplus r_2 \wedge [\![P]\!](r_1) \wedge [\![Q]\!](r_2)
\end{array}
$$

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

$$
[\![\ell \mapsto v]\!](\sigma) \triangleq \ell \in \mathrm{dom}(\sigma) \wedge \sigma(\ell) = v
$$

Models of Separation Logics: Base Logic

The **interesting part**: interpretation of base assertions $[\![P]\!]$

In general: $[\![P]\!] : R \to \mathrm{Prop}$

wi

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 $R = Loc \overset{\text{fin}}{\rightharpoonup} Val$:

$$[\![\ell \mapsto v]\!](\sigma) \triangleq \ell \in \mathrm{dom}(\sigma) \land \sigma(\ell) = v$$

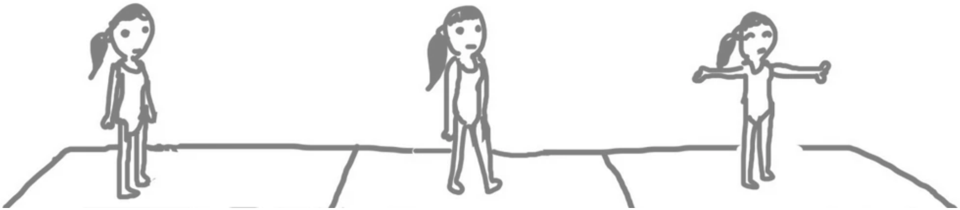
Iris: a Framework for Building Separation Logics

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**

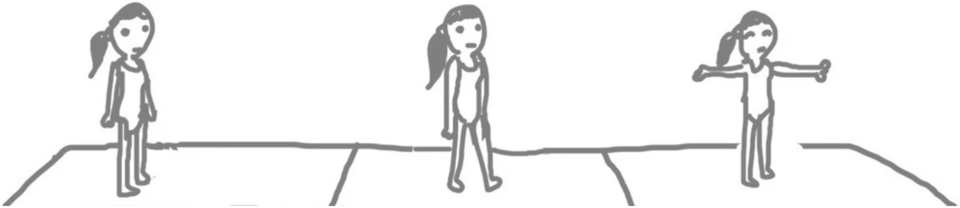
Most steps of the Checklist become `Require Import iris`!

mono-language separation logic gymnastics



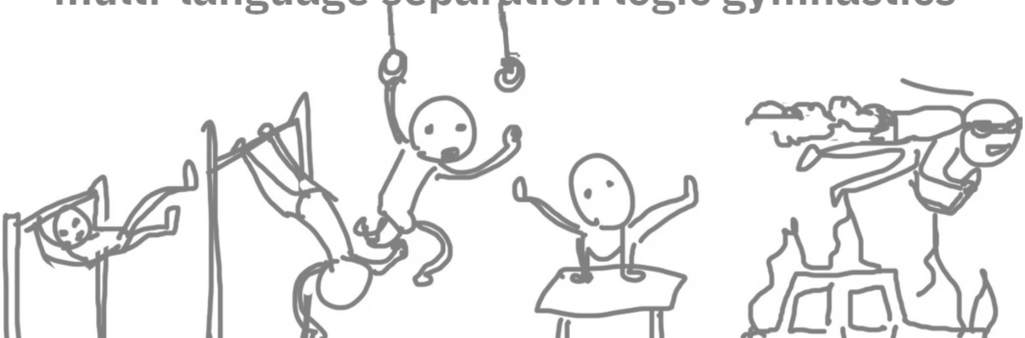
Bridging Separation Logics with View Reconciliation

mono-language separation logic gymnastics



Bridging Separation Logics with View Reconciliation

multi-language separation logic gymnastics



Melocoton brings the Checklist to a multi-language setting

OCaml SL

$\{P\}\, e_{\text{ML}}\, \{Q\}$

$r \mapsto_{\text{ML}} v$

+

FFI SL

$\{P\}\, e_{\text{FFI}}\, \{Q\}$

$\gamma \mapsto_{\text{blk}[0|\mathsf{mut}]} blk$

+

C SL

$\{P\}\, e_{\text{C}}\, \{Q\}$

$a \mapsto_{\text{C}} w$

OCaml semantics

$(e_{\text{ML}}, \sigma) \rightsquigarrow (e_{\text{ML}}, \sigma)$

FFI semantics

$(e_{\text{FFI}}, \rho) \rightsquigarrow (e_{\text{FFI}}, \rho)$

C semantics

$(e_{\text{C}}, m) \rightsquigarrow (e_{\text{C}}, m)$

Problem: how do we connect the different languages/logics?

Language Interaction: Different Views of the Same Data

OCaml glue code

C glue code

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

```
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?

Language Interaction: Different Views of the Same Data

OCaml glue code

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

C glue code

```
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.

Language Interaction: Semantics

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

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

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

Language Interaction: Semantics

High-level **OCaml** value $\sim_{\mathtt{ML}}$ Low-level **block** representation
integers $\sim_{\mathtt{ML}}$ integers
booleans $\sim_{\mathtt{ML}}$ integers (0 or 1)
arrays, refs $\sim_{\mathtt{ML}}$ blocks

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

$\ell \sim_{\mathtt{ML}} \gamma$

Language Interaction: Semantics

High-level **OCaml** value $\sim_{\mathtt{ML}}$ Low-level **block** representation
integers $\sim_{\mathtt{ML}}$ integers
booleans $\sim_{\mathtt{ML}}$ integers (0 or 1)
arrays, refs $\sim_{\mathtt{ML}}$ blocks
pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

`true` $\sim_{ML} 1$

$\ell \sim_{ML} \gamma$

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

arrays, refs $\sim_{\mathtt{ML}}$ blocks

pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

lists $\sim_{\mathtt{ML}}$ block-based linked lists

`true` $\sim_{\mathtt{ML}} 1$

$\ell \sim_{\mathtt{ML}} \gamma$

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

arrays, refs $\sim_{\mathtt{ML}}$ blocks

pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

lists $\sim_{\mathtt{ML}}$ block-based linked lists

`true` $\sim_{\mathtt{ML}}$ 1

$\ell \sim_{\mathtt{ML}} \gamma$

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

$\sigma : \mathit{Heap}_{\mathtt{ML}}$

$\zeta : \mathit{BlockHeap}$

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

arrays, refs $\sim_{\mathtt{ML}}$ blocks

pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

lists $\sim_{\mathtt{ML}}$ block-based linked lists

`true` $\sim_{\mathtt{ML}}$ 1

$\ell \sim_{\mathtt{ML}} \gamma$

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

$$\begin{tikzcd}
\sigma : \mathit{Heap}_{\mathtt{ML}} \arrow[r, shift left] & \zeta : \mathit{BlockHeap} \arrow[l, shift left]
\end{tikzcd}$$

switch at the language barrier

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

arrays, refs $\sim_{\mathtt{ML}}$ blocks

pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

lists $\sim_{\mathtt{ML}}$ block-based linked lists

`true` $\sim_{\mathtt{ML}} 1$

$\ell \sim_{\mathtt{ML}} \gamma$

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

$$\begin{tikzcd} \sigma : \mathit{Heap}_{\mathtt{ML}} \arrow[r, bend left] & \zeta : \mathit{BlockHeap} \arrow[l, bend left] \end{tikzcd}$$

switch at the language barrier

Language Interaction: Semantics

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

integers $\sim_{\texttt{ML}}$ integers

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

arrays, refs $\sim_{\texttt{ML}}$ blocks

pairs $\sim_{\texttt{ML}}$ blocks (of size 2)

lists $\sim_{\texttt{ML}}$ block-based linked lists

`true` $\sim_{\texttt{ML}}$ 1

$\ell \sim_{\texttt{ML}} \gamma$

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

$$\begin{tikzcd}
\sigma : \mathit{Heap}_{\texttt{ML}} \arrow[r, bend left] & \zeta : \mathit{BlockHeap} \arrow[l, bend left]
\end{tikzcd}$$

switch at the language barrier

Language Interaction: Semantics

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

integers $\sim_{\mathtt{ML}}$ integers

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

arrays, refs $\sim_{\mathtt{ML}}$ blocks

pairs $\sim_{\mathtt{ML}}$ blocks (of size 2)

lists $\sim_{\mathtt{ML}}$ block-based linked lists

`true` $\sim_{\mathtt{ML}}$ 1

$\ell \sim_{\mathtt{ML}} \gamma$

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

$$\begin{tikzcd} \sigma : \mathit{Heap}_{\mathtt{ML}} \arrow[r, bend left] & \zeta : \mathit{BlockHeap} \arrow[l, bend left] \end{tikzcd}$$

switch at the language barrier

Language Interaction: Program Logic, Take 1

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

$$\begin{tikzcd} \sigma : Heap_{\mathtt{ML}} \arrow[r, shift left] & \zeta : BlockHeap \arrow[l, shift left] \end{tikzcd}$$

Language Interaction: Program Logic, Take 1

$\lambda_{\texttt{ML+C}}$ **Program Logic**

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

$$\begin{tikzcd}
\sigma : \mathit{Heap}_{\texttt{ML}} \arrow[r, bend left] & \zeta : \mathit{BlockHeap} \arrow[l, bend left]
\end{tikzcd}$$

Language Interaction: Program Logic, Take 1

$\lambda_{\texttt{ML+C}}$ **Program Logic**

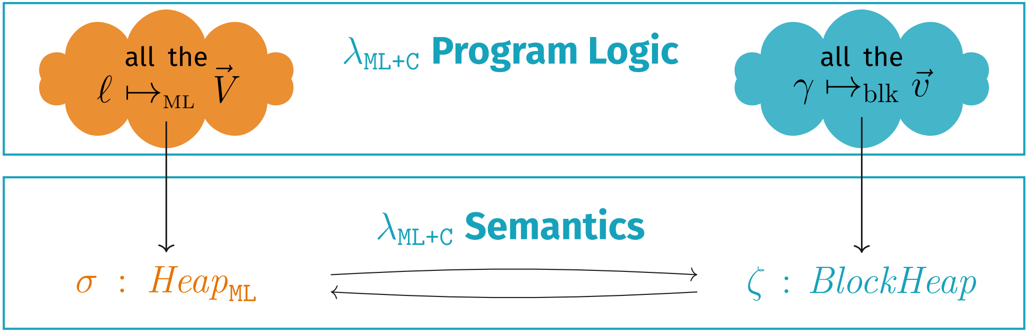
$\ell \mapsto_{\texttt{ML}} \vec{V}$

$\gamma \mapsto_{\text{blk}} \vec{v}$

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

$$\begin{tikzcd} \sigma : \mathit{Heap}_{\texttt{ML}} \arrow[r, bend left] & \zeta : \mathit{BlockHeap} \arrow[l, bend left] \end{tikzcd}$$

Language Interaction: Program Logic, Take 1



$\lambda_{\texttt{ML+C}}$ **Program Logic**

all the $\ell \mapsto_{\text{ML}} \vec{V}$

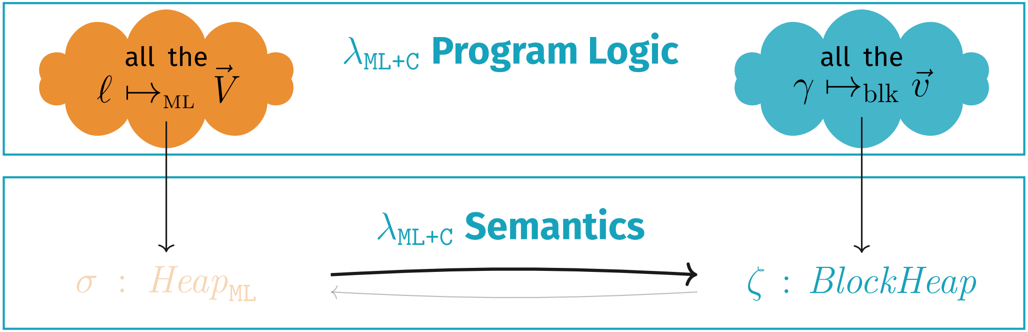
all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

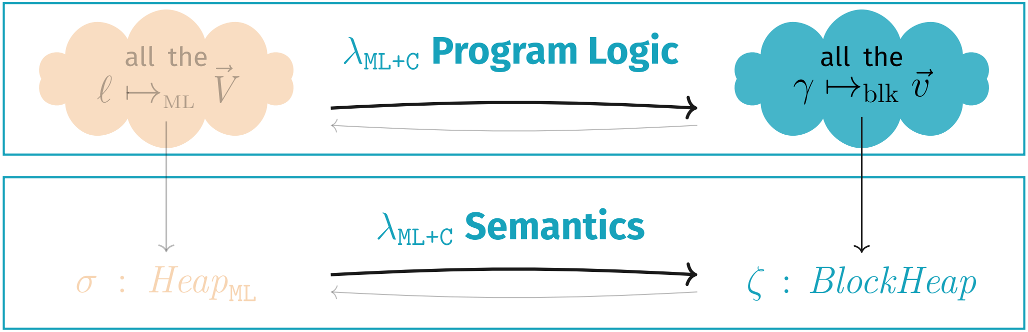
all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta \,:\, \mathit{BlockHeap}$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

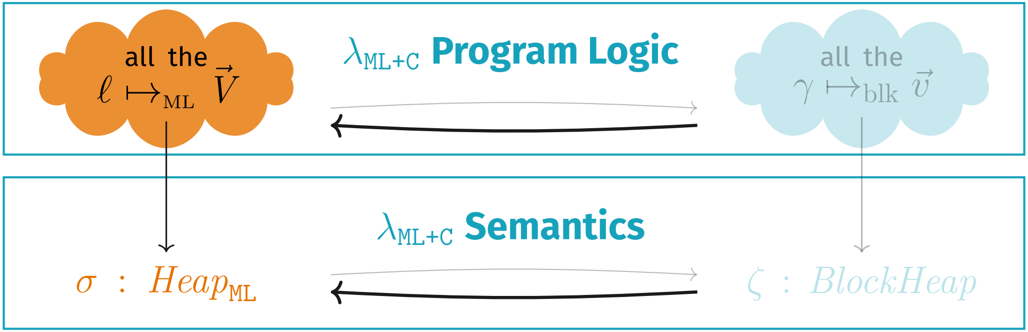
all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

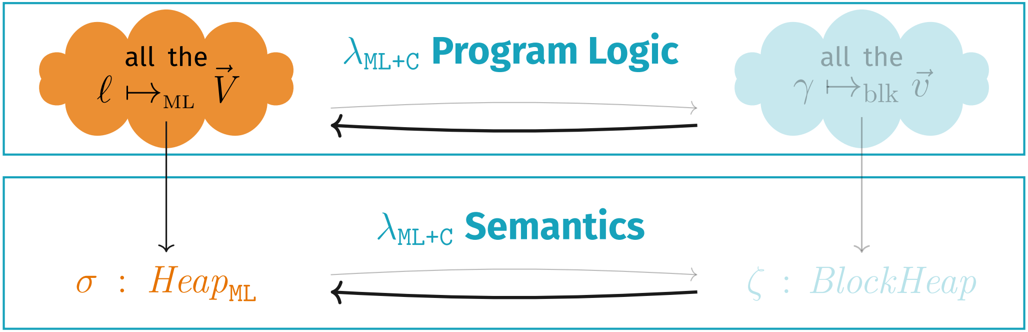
all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

Language Interaction: Program Logic, Take 1

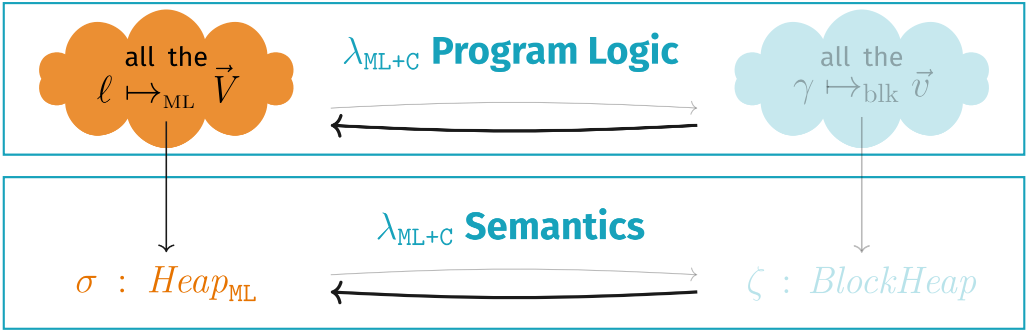


ExtCall

{ all } **C** function body { all }

{ all } call into **C** { all }

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

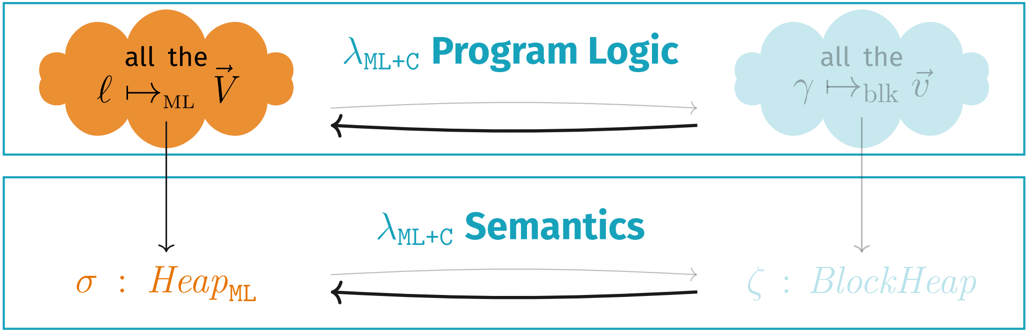
EXTCALL

$$\frac{\{\text{all}\}\ \textbf{C}\ \text{function body}\ \{\text{all}\}}{\{\text{all}\}\ \text{call into}\ \textbf{C}\ \{\text{all}\}}$$

FRAME

$$\frac{\{P\}\, e\, \{Q\}}{\{R * P\}\, e\, \{Q * R\}}$$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

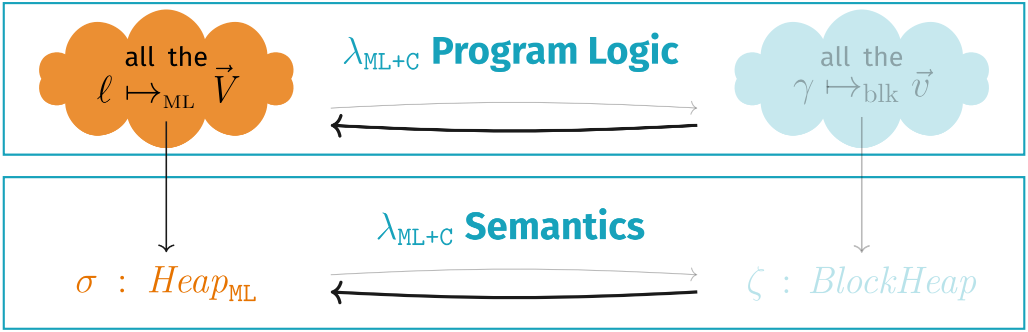
ExtCall

$$\frac{\{\text{all}\}\ \textbf{C}\ \text{function body}\ \{\text{all}\}}{\{\text{all}\}\ \text{call into}\ \textbf{C}\ \{\text{all}\}}$$

Frame

$$\frac{\{P\}\ \text{call into}\ \textbf{C}\ \{Q\}}{\{R * P\}\ \text{call into}\ \textbf{C}\ \{Q * R\}}$$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\mathrm{ML}} \vec{V}$

$\lambda_{\mathrm{ML+C}}$ **Program Logic**

all the $\gamma \mapsto_{\mathrm{blk}} \vec{v}$

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

$\sigma : \mathit{Heap}_{\mathrm{ML}}$

$\zeta : \mathit{BlockHeap}$

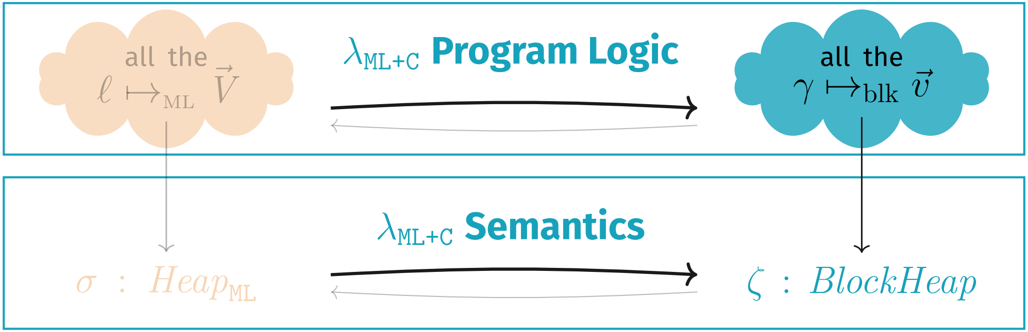
ExtCall

$$\frac{\{\text{all}\}\ \textbf{C}\ \text{function body}\ \{\text{all}\}}{\{\text{all}\}\ \text{call into}\ \textbf{C}\ \{\text{all}\}}$$

Frame

$$\frac{\{P\}\ \text{call into}\ \textbf{C}\ \{Q\}}{\{\ell \mapsto_{\mathrm{ML}} \vec{V} * P\}\ \text{call into}\ \textbf{C}\ \{Q * \ell \mapsto_{\mathrm{ML}} \vec{V}\}}$$

Language Interaction: Program Logic, Take 1



all the $\ell \mapsto_{\text{ML}} \vec{V}$

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

all the $\gamma \mapsto_{\text{blk}} \vec{v}$

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

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

$\zeta : \mathit{BlockHeap}$

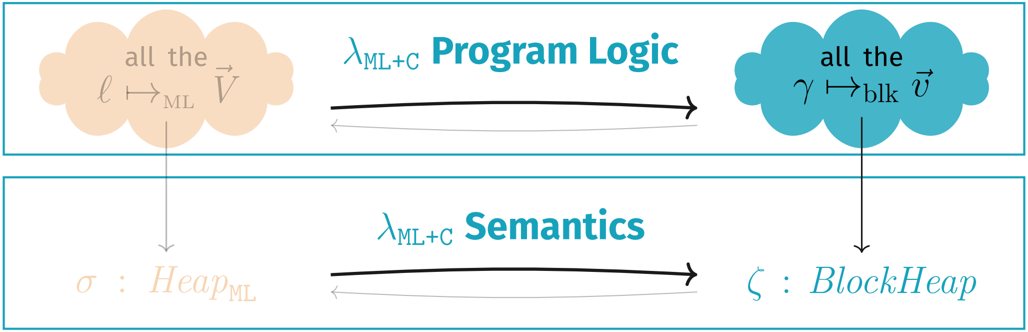
ExtCall

$$\frac{\{\text{all}\}\ \textbf{C}\text{ function body }\{\text{all}\}}{\{\text{all}\}\text{ call into }\textbf{C}\ \{\text{all}\}}$$

Frame

$$\frac{\{P\}\text{ call into }\textbf{C}\ \{Q\}}{\{\ell \mapsto_{\text{ML}} \vec{V} * P\}\text{ call into }\textbf{C}\ \{Q * \ell \mapsto_{\text{ML}} \vec{V}\}}$$

Language Interaction: Program Logic, Take 1



ExtCall

$$\frac{\{\text{all}\}\ \textbf{C}\ \text{function body}\ \{\text{all}\}}{\{\text{all}\}\ \text{call into }\textbf{C}\ \{\text{all}\}}$$

RAME

$$\frac{\{P\}\ \text{call into }\textbf{C}\ \{Q\}}{\{\ell \mapsto_{\mathrm{ML}} \vec{V} * P\}\ \text{call into }\textbf{C}\ \{Q * \ell \mapsto_{\mathrm{ML}} \vec{V}\}}$$

Language Interaction: Program Logic, Take 1

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

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

↯

The $\lambda_{\text{ML+C}}$ Program Logic needs **local** reasoning rules

EXTCALL

$$\frac{\{\text{all}\}\ \textsf{C function body}\ \{\text{all}\}}{\{\text{all}\}\ \textsf{call into C}\ \{\text{all}\}}$$

RAME

$$\frac{\{P\}\ \textsf{call into C}\ \{Q\}}{\{\ell \mapsto_{\text{ML}} \vec{V} * P\}\ \textsf{call into C}\ \{Q * \ell \mapsto_{\text{ML}} \vec{V}\}}$$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

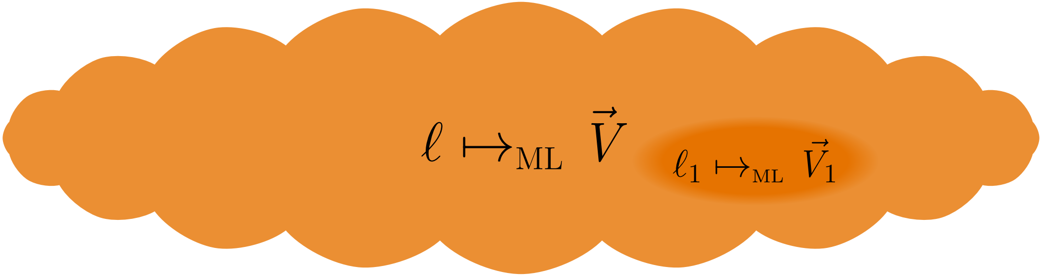
Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

$$\ell \mapsto_{\mathrm{ML}} \vec{V}$$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

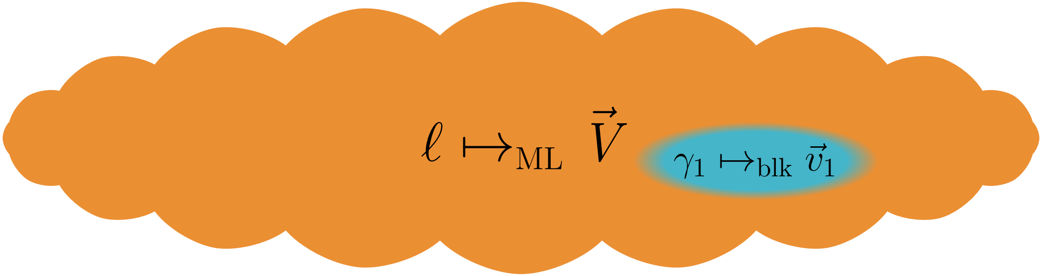


$\ell \mapsto_{\mathrm{ML}} \vec{V}$

$\ell_1 \mapsto_{\mathrm{ML}} \vec{V}_1$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

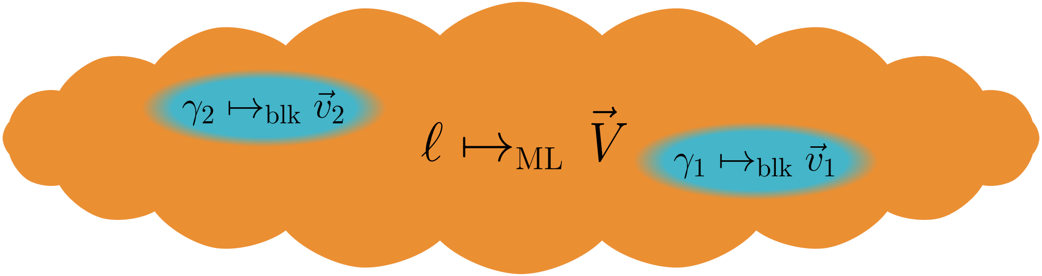


$\ell \mapsto_{\mathrm{ML}} \vec{V}$

$\gamma_1 \mapsto_{\mathrm{blk}} \vec{v}_1$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!



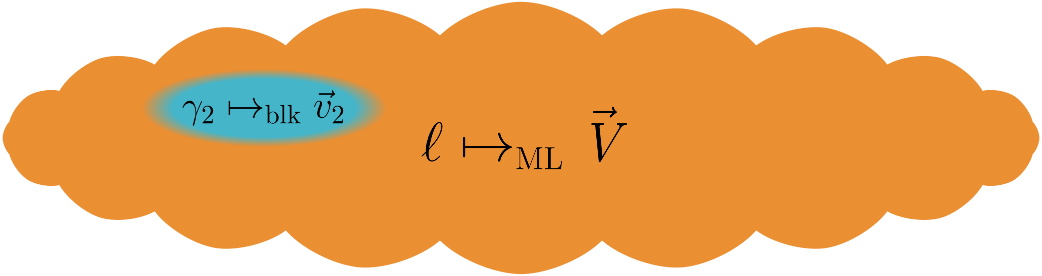
$\gamma_2 \mapsto_{\mathrm{blk}} \vec{v}_2$

$\ell \mapsto_{\mathrm{ML}} \vec{V}$

$\gamma_1 \mapsto_{\mathrm{blk}} \vec{v}_1$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!



$\gamma_2 \mapsto_{\mathrm{blk}} \vec{v}_2$

$\ell \mapsto_{\mathrm{ML}} \vec{V}$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

$$\ell \mapsto_{\mathrm{ML}} \vec{V}$$

Language Interaction: More Gradual Rules

OCaml points-tos *remain valid* when switching to **C**!

$$\ell \mapsto_{\mathrm{ML}} \vec{V}$$

View Reconciliation Rules for Converting On-Demand:

$$\ell \mapsto_{\mathrm{ML}} \vec{V} \Rrightarrow\!\!\ast\; \exists \gamma \vec{v}.\, \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast\; \exists \ell\ .\, \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$

Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \equiv\!\!\!\ast \exists \gamma \vec{v}.\ \gamma \mapsto_{\mathrm{blk}} \vec{v} \ast \ell \sim_{\mathtt{ML}} \gamma \ast \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

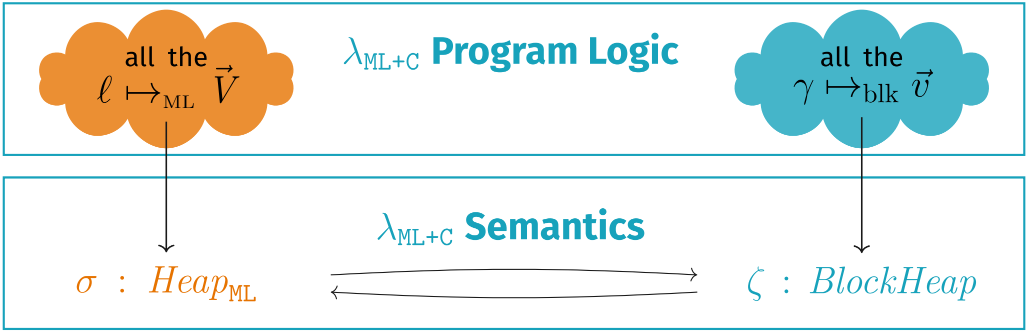
$$\vec{V} \sim_{\mathtt{ML}} \vec{v} \ast \gamma \mapsto_{\mathrm{blk}} \vec{v} \equiv\!\!\!\ast \exists \ell.\ \ell \mapsto_{\mathtt{ML}} \vec{V} \ast \ell \sim_{\mathtt{ML}} \gamma$$

Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \equiv\!\!\ast\ \exists \gamma \vec{v}.\ \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \equiv\!\!\ast\ \exists \ell.\ \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$



all the $\ell \mapsto_{\mathtt{ML}} \vec{V}$

$\lambda_{\mathtt{ML+C}}$ **Program Logic**

all the $\gamma \mapsto_{\mathrm{blk}} \vec{v}$

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

$\sigma : \mathit{Heap}_{\mathtt{ML}}$

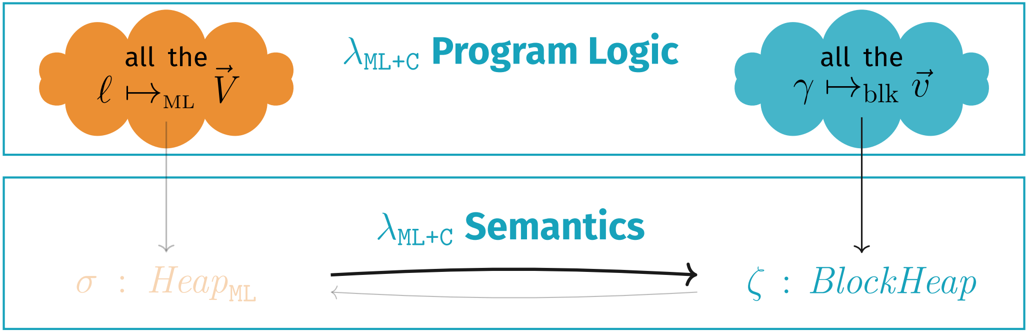
$\zeta : \mathit{BlockHeap}$

Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \;\Rrightarrow\!\!\ast\; \exists \gamma \vec{v}.\, \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \;\Rrightarrow\!\!\ast\; \exists \ell .\, \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$



all the $\ell \mapsto_{\mathtt{ML}} \vec{V}$

$\lambda_{\mathtt{ML+C}}$ **Program Logic**

all the $\gamma \mapsto_{\mathrm{blk}} \vec{v}$

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

$\sigma : \mathit{Heap}_{\mathtt{ML}}$

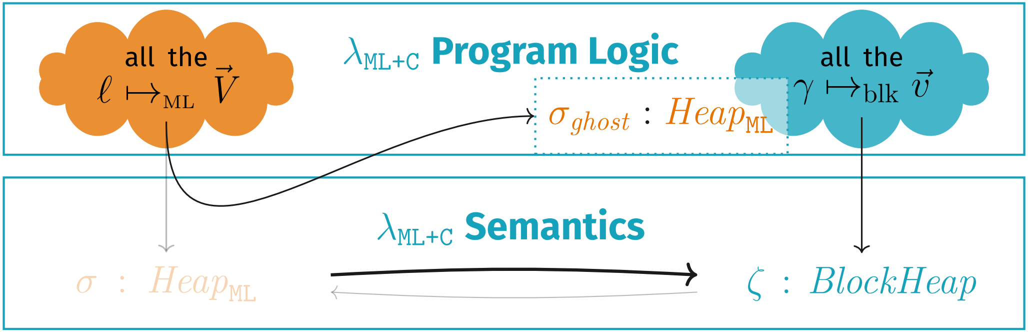
$\zeta : \mathit{BlockHeap}$

Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \Rrightarrow\!\!\ast \exists \gamma \vec{v}.\, \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast \exists \ell.\, \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$



all the $\ell \mapsto_{\mathtt{ML}} \vec{V}$

$\lambda_{\mathtt{ML+C}}$ **Program Logic**

all the $\gamma \mapsto_{\mathrm{blk}} \vec{v}$

$\sigma_{ghost} : Heap_{\mathtt{ML}}$

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

$\sigma : Heap_{\mathtt{ML}}$

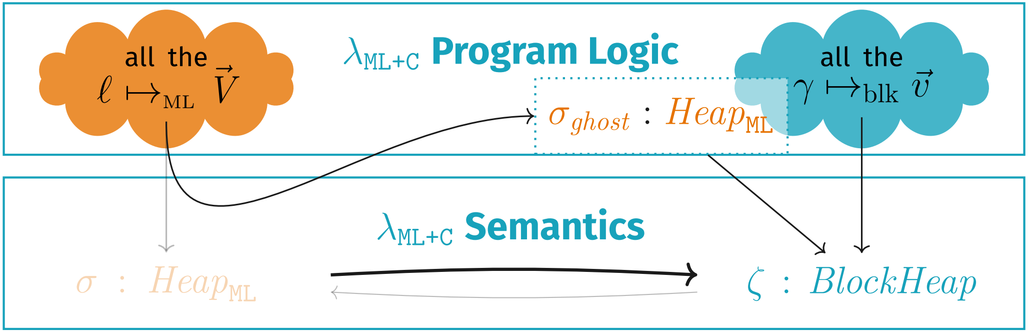
$\zeta : BlockHeap$

Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \;\Rrightarrow\!\!\ast\; \exists \gamma \vec{v}.\; \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \;\Rrightarrow\!\!\ast\; \exists \ell\;.\; \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$

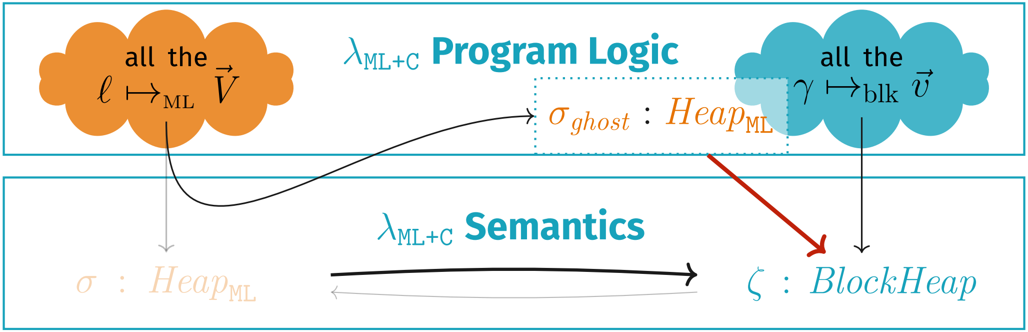


Language Interaction: View Reconciliation

View Reconciliation Rules

$$\ell \mapsto_{\mathtt{ML}} \vec{V} \Rrightarrow\!\!\ast \exists \gamma \vec{v}.\, \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathtt{ML}} \gamma * \vec{V} \sim_{\mathtt{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast \exists \ell.\, \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$



all the $\ell \mapsto_{\mathtt{ML}} \vec{V}$

$\lambda_{\mathtt{ML+C}}$ **Program Logic**

all the $\gamma \mapsto_{\mathrm{blk}} \vec{v}$

$\sigma_{ghost} : Heap_{\mathtt{ML}}$

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

$\sigma : Heap_{\mathtt{ML}}$

$\zeta : BlockHeap$

Application: Verifying `hash_ref` with Melocoton

Verifying `hash_ref` with Melocoton

OCaml glue code

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

C glue code

```
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

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
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

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
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;
}
```

ExtCall

$$\frac{\{P * x \sim_{\mathrm{ML}} v\}\; \texttt{f}(v)\; \{\lambda v'.\, \exists y.\, y \sim_{\mathrm{ML}} v' * Q(y)\}}{\{P\}\; \texttt{external "f"}(x)\; \{\lambda y.\, Q(y)\}}$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathrm{ML}} n * r \sim_{\mathrm{ML}} v\}$$

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

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m * \exists y.\, y \sim_{\mathrm{ML}} \texttt{Val\_unit}\}$$

```
}
```

$$\textsc{ExtCall}\quad \frac{\{P * x \sim_{\mathrm{ML}} v\}\ \texttt{f}(v)\ \{\lambda v'.\, \exists y.\, y \sim_{\mathrm{ML}} v' * Q(y)\}}{\{P\}\ \texttt{external "f"}(x)\ \{\lambda y.\, Q(y)\}}$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathrm{ML}} n * r \sim_{\mathrm{ML}} v\}$$

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

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m * () \sim_{\mathrm{ML}} \texttt{Val\_unit}\}$$

```
}
```

ExtCall

$$\frac{\{P * x \sim_{\mathrm{ML}} v\}\ \texttt{f}(v)\ \{\lambda v'.\, \exists y.\, y \sim_{\mathrm{ML}} v' * Q(y)\}}{\{P\}\ \texttt{external "f"}(x)\ \{\lambda y.\, Q(y)\}}$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\ r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathrm{ML}} n * r \sim_{\mathrm{ML}} v\}$$

$$\{v \mapsto_{\mathrm{blk}} [n] * r \sim_{\mathrm{ML}} v\}$$

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

$$\{\exists m.\ r \mapsto_{\mathrm{ML}} m * () \sim_{\mathrm{ML}} \texttt{Val\_unit}\}$$

```
}
```

VIEW RECONCILIATION (1)

$$\ell \mapsto_{\mathrm{ML}} \vec{V} \equiv\!\!\!\ast\ \exists \gamma \vec{v}.\ \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathrm{ML}} \gamma * \vec{V} \sim_{\mathrm{ML}} \vec{v}$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathrm{ML}} n * r \sim_{\mathrm{ML}} v\}$$

$$\{v \mapsto_{\mathrm{blk}} [n] * r \sim_{\mathrm{ML}} v\}$$

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

$$\{\exists m.\, v \mapsto_{\mathrm{blk}} [m] * r \sim_{\mathrm{ML}} v\}$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m * () \sim_{\mathrm{ML}} \texttt{Val\_unit}\}$$

```
}
```

VIEW RECONCILIATION (2)

$$\vec{V} \sim_{\mathrm{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast\ \exists \ell\ .\, \ell \mapsto_{\mathrm{ML}} \vec{V} * \ell \sim_{\mathrm{ML}} \gamma$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathtt{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathtt{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathtt{ML}} n * r \sim_{\mathtt{ML}} v\}$$

$$\{v \mapsto_{\mathrm{blk}} [n] * r \sim_{\mathtt{ML}} v\}$$

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

$$\{\exists m.\, v \mapsto_{\mathrm{blk}} [m] * r \sim_{\mathtt{ML}} v\}$$

$$\{\exists m.\, r \mapsto_{\mathtt{ML}} m * () \sim_{\mathtt{ML}} \texttt{Val\_unit}\}$$

```
}
```

`Field` SPECIFICATION

$$\{\gamma \mapsto_{\mathrm{blk}} [\ldots v_i \ldots]\}\ \texttt{Field}(\gamma, i)\ \{\lambda v'.\, v' = v_i \wedge \gamma \mapsto_{\mathrm{blk}} [\ldots v_i \ldots]\}$$

Verifying `hash_ref` with Melocoton

OCaml glue code

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

$$\{r \mapsto_{\mathrm{ML}} n\}$$

$$\texttt{hash\_ref}(r)$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m\}$$

C glue code

```
value caml_hash_ref(value v) {
```

$$\{r \mapsto_{\mathrm{ML}} n * r \sim_{\mathrm{ML}} v\}$$

$$\{v \mapsto_{\mathrm{blk}} [n] * r \sim_{\mathrm{ML}} v\}$$

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

$$\{\exists m.\, v \mapsto_{\mathrm{blk}} [m] * r \sim_{\mathrm{ML}} v\}$$

$$\{\exists m.\, r \mapsto_{\mathrm{ML}} m * () \sim_{\mathrm{ML}} \texttt{Val\_unit}\}$$

```
}
```

`Store_field` SPECIFICATION

$$\{\gamma \mapsto_{\mathrm{blk}} [\ldots v_i \ldots]\}\ \texttt{Store\_field}(\gamma, i, v')\ \{\gamma \mapsto_{\mathrm{blk}} [\ldots v' \ldots]\}$$

But wait, there is more!

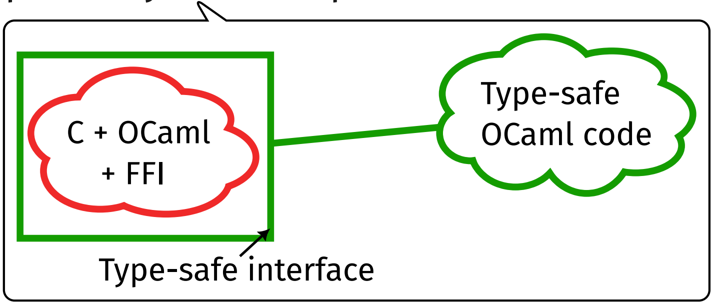
- Language-local reasoning for **external calls**.
- Additional **OCaml FFI features**: garbage collection, registering roots, custom blocks, callbacks, etc.
- **Case studies** utilising all of these features.
- **Semantic model of OCaml types** (a logical relation) to verify type safety and encapsulation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

Transfinite

Iris

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, ...

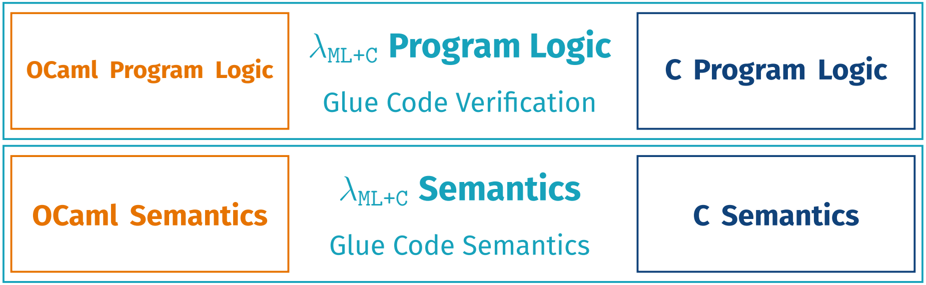
Build code-analysis tools based on Melocoton

Allow OCaml programmers to check correctness of their FFI glue code

Ideally, both verification and bug finding tools

Melocoton

Language Locality: Embed Existing Languages



OCaml Program Logic

$\lambda_{\mathrm{ML+C}}$ **Program Logic**

Glue Code Verification

C Program Logic

OCaml Semantics

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

Glue Code Semantics

C Semantics

Language Interaction: View Reconciliation Rules

$$\ell \mapsto_{\mathrm{ML}} \vec{V} \Rrightarrow\!\!\ast\; \exists \gamma \vec{v}.\, \gamma \mapsto_{\mathrm{blk}} \vec{v} * \ell \sim_{\mathrm{ML}} \gamma * \vec{V} \sim_{\mathrm{ML}} \vec{v}$$

$$\vec{V} \sim_{\mathrm{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast\; \exists \ell .\, \ell \mapsto_{\mathrm{ML}} \vec{V} * \ell \sim_{\mathrm{ML}} \gamma$$

https://melocoton-project.github.io