# Melocoton

## A Program Logic for Verified Interoperability Between OCaml and C

Armaël Guéneau, Johannes Hostert, Simon Spies,

Michael Sammler, Lars Birkedal, Derek Dreyer

CoqPL

January 20th, 2024

Inria

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

# Multi-Language Programs Are Everywhere

NumPy

- Python
- C
- Fortran

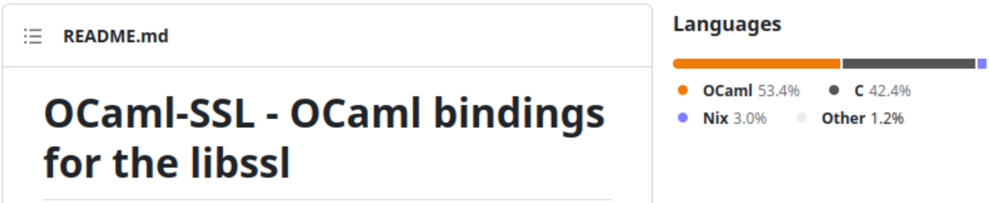
Firefox

- C++
- Rust
- JavaScript

OpenSSL
Cryptography and SSL/TLS Toolkit

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

# Multi-Language Programs Are Everywhere




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

- Go
- . . .

# Mind the gap!

| **OCaml** | ← **OCaml FFI** → | **C** |
|---|---|---|
| Structured values | | Integers and pointers |
| Garbage collection | | Manual memory management |

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

# Goal: Verifying Multi-Language Code

How do we

**verify functional correctness**

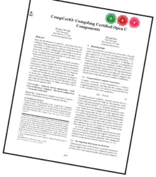
of code written in

**different languages**?

# Single-Language Functional Correctness

Hoare Logic for simple imperative languages.

Separation Logic for modularity and aliasing.


CompCertO: Compiling Certified Open C Components


Verifying an Open Compiler Using Multi-language Semantics

# Multi-Language Functional Correctness


Semantic Soundness for Language Interoperability


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


Operational Semantics for Multi-Language Programs


Reasonably Mixing a Functional Language with Assembly



# Multi-Language Functional Correctness

Existing work on Semantics and Logical Relations.

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





CompCertO: Compiling Certified Open C Components


Verifying an Open Compiler Using Multi-language Semantics


Operational Semantics for Multi-Language Programs

# Multi-Language Functional Correctness

Existing work on Semantics and Logical Relations.

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

**Melocoton is the first program logic for multiple languages with different memory models**


Semantic Soundness for Language Interoperability


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

# 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

```
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
}
```

# 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

```
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

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

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

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

**C** glue code

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

```
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;
}
```

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

# A Multi-Language Program Logic for FFI

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

**OCaml** glue code

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

```
external hash_ref: int
  = "caml_hash_ref"
```

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


**C** glue code

$[n]\}$

```
ref(value r) {
val(Field(r, 0));
;
(r, 0, Val_int(x));
_unit;
```

$k[\mathrm{U}|\mathrm{mut}]\ [m]\}$

# We Have A Tool For That: It Is Called Iris

Iris

an expressive Separation Logic Framework
implemented in Coq

The Iris Methodology for **building your own program logic**:

- define operational semantics of your language
- define interpretation of program state in the Iris logic
- prove reasoning rules for operations of the language

# Solution: Just Do That?

Solution?: Apply the methodology to **“OCaml + C + FFI”**?

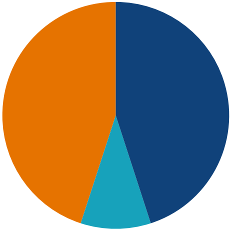
# Solution: Just Do That?

Solution?: Apply the methodology to **"OCaml + C + FFI"**?

One Big Language:
unsatisfactory for **engineering** and **conceptual** reasons

Most multi-language programs look like this:

**OCaml** business logic
oblivious of C

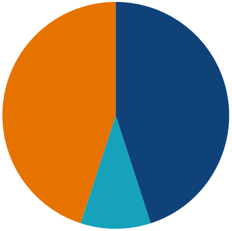


**C** business logic
oblivious of OCaml

**glue code**
where the languages actually interact

Most multi-language programs look like this:

**OCaml** business logic
oblivious of C



**C** business logic
oblivious of OCaml

**glue code**
where the languages actually interact

**Design Principle: Language-Local Reasoning**

**Reuse** existing single-language semantics and program logics

# Our Contribution: Melocoton

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

Glue Code Verification

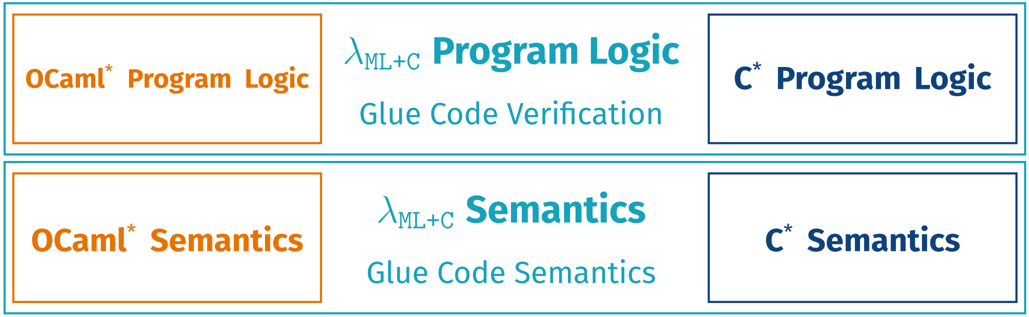
$\lambda_{\texttt{ML+C}}$ **Semantics**

Glue Code Semantics

**“Iris Methodology”:** program logic on top of semantics, **but**

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

# Our Contribution: Melocoton




**"Iris Methodology":** 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* | $\lambda_{\text{ML+C}}$ | C* |
|---|---|---|
| OCaml* Program Logic | $\lambda_{\text{ML+C}}$ Program Logic<br>Glue Code Verification | C* Program Logic |
| OCaml* Semantics | $\lambda_{\text{ML+C}}$ Semantics<br>Glue Code Semantics | C* Semantics |

**"Iris Methodology":** program logic on top of semantics, **but**

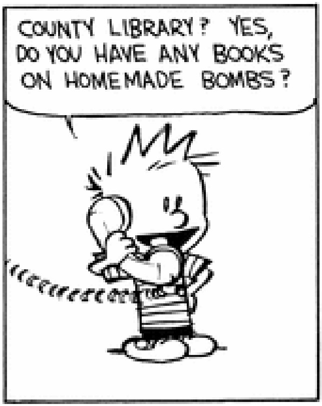
- **Language Interaction:** new semantics and logic for glue code
- **Language Locality:** embed existing semantics and logics

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

# The rest of this talk

1. Language-Local Reasoning with External Calls
2. Bridging Languages with View Reconciliation
3. A Tour of the Coq Formalization

# Language-Local Reasoning with External Calls



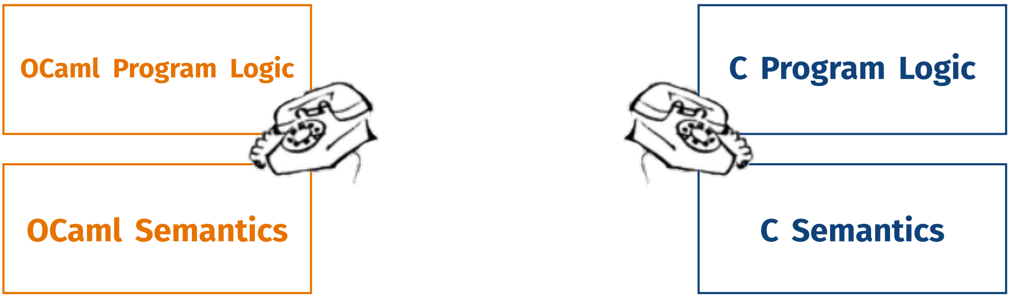
# Language-local Reasoning for Existing Languages

We reuse:

| | |
|---|---|
| OCaml Program Logic | C Program Logic |
| OCaml Semantics | C Semantics |

# Language-local Reasoning for Existing Languages

We reuse:




with a minimal extension: we add **external calls**

# Modeling External Calls

OCaml 

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

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

- operational semantics: **none** (i.e. stuck)
- program logic: **assume** a specification for the call

# Modeling External Calls

OCaml 

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

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

- operational semantics: **none** (i.e. stuck)
- program logic: **assume** a specification for the call

Assuming specification: $\{r \mapsto_{\text{ML}} n\}\ \texttt{hash\_ref}(r)\ \{\exists m.\ r \mapsto_{\text{ML}} m\}$

Use the **language-local** OCaml program logic to verify `main`

# Modeling External Calls, Formally

Standard Separation Logic triple:

$$\{P\}\ e\ \{Q\}$$

Melocoton language-local triple:

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

interface: specs assumed for external calls

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

# FFI Operations are External Calls for C

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_int(0);
}
```

“glue code” verified using the **language-local C program logic**

against interface $\Psi_{\mathrm{FFI}}$ and FFI assertions e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$

# FFI Operations are External Calls for C

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_int(0);
}
```

“glue code” verified using the **language-local C program logic**

against interface $\Psi_{\mathrm{FFI}}$ and FFI assertions e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$

$$\Psi_{\mathrm{FFI}} \triangleq \left\{\gamma \mapsto_{\mathrm{blk}[t|\mathsf{mut}]} [\ldots v_i \ldots]\right\} \texttt{Store\_field}(\gamma, i, v') \left\{\gamma \mapsto_{\mathrm{blk}[t|\mathsf{mut}]} [\ldots v' \ldots]\right\}$$

$\sqcup$ specs for `Field`, `Int_val`, etc...

# FFI Operations are External Calls for C

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_int(0);
}
```

"glue code" verified using the **language-local C program logic**

against interface $\Psi_{\mathrm{FFI}}$ and FFI assertions e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$

$$\Psi_{\mathrm{FFI}} \triangleq \left\{\gamma \mapsto_{\mathrm{blk}[t|\mathsf{mut}]} [\dots v_i \dots]\right\} \texttt{Store\_field}(\gamma, i, v') \left\{\gamma \mapsto_{\mathrm{blk}[t|\mathsf{mut}]} [\dots v' \dots]\right\}$$

$\sqcup$ specs for `Field`, `Int_val`, etc...

Verify the code in the **language-local** C program logic:

$$\{\gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n]\}\ \texttt{caml\_hash\_ref}(r) \mathbin{@} \Psi_{\mathrm{FFI}}\ \{\exists m.\, \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m]\}$$

# What we have so far

## 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;
}
```

# What we have so far

## OCaml business logic

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

## C business logic

```
void hash_ptr                    {
    // Imple               penSSL
    // tediou               to OCaml
}
```

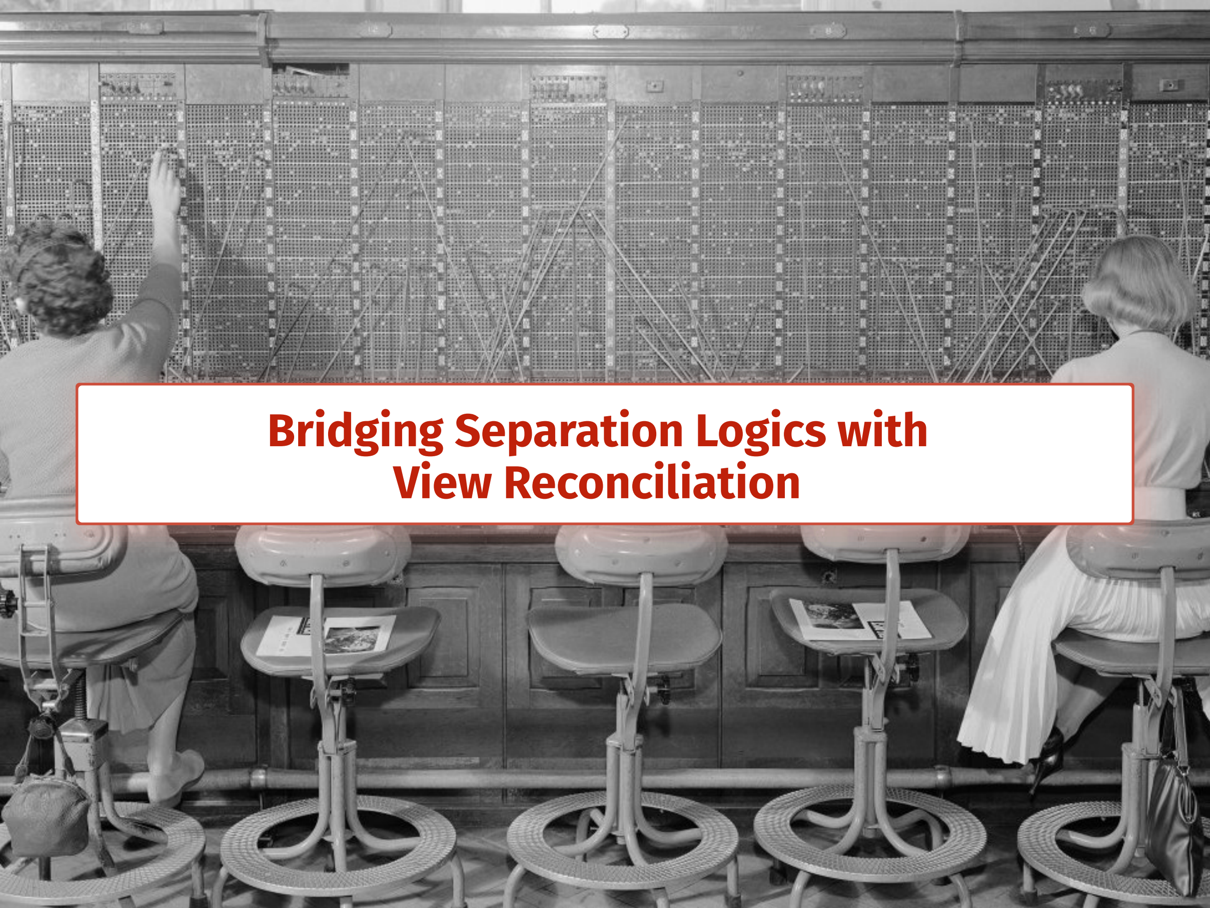
## OCaml glue code

```
external ha       ef : int ref -> unit
  = "cam          "
```

## C glue code

```
value caml_hash        (value r) {
    int x =            eld(r, 0));
    hash_pt
    Store_fi           Val_int(x));
    return Val_    t;
}
```

**Missing: connecting the semantics and proofs!**



# Bridging Separation Logics with View Reconciliation

## We assumed:

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

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

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

## We proved:

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

```
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;
}
```

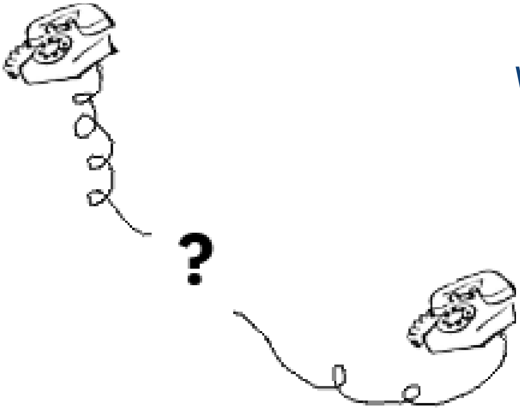
$\{\exists m.\, \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m]\}$

## We assumed:

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

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

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




## We proved:

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

```
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;
}
```

$\{\exists m.\, \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m]\}$

Two **different views** about the same **piece of state**!

# 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?

# 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` $\sim_{\mathtt{ML}}$ $\mathit{1}$

# 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 |

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

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

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

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

# Language Interaction: Semantics

| High-level **OCaml** value | | 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 |

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

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

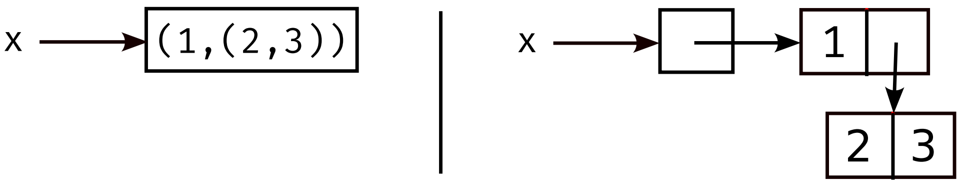
# Language Interaction: Semantics

| High-level **OCaml** value | | 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 |

true $\sim_{ML}$ $1$

$\ell \sim_{ML} \gamma$

```
let x = ref (1, (2, 3))
```



# Language Interaction: Semantics (2)

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

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

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\texttt{ML}}$ ⇄ $\zeta : BlockHeap$

switch at the language barrier

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\texttt{ML}} \rightleftarrows \zeta : BlockHeap$

switch at the language barrier

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\texttt{ML}}$ $\longrightarrow$ $\zeta : BlockHeap$

switch at the language barrier

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\mathtt{ML}}$ ⇄ $\zeta : BlockHeap$

switch at the language barrier

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\mathtt{ML}} \longleftarrow \zeta : BlockHeap$

switch at the language barrier

Whole program state: ML + C state (+ extra omitted FFI state):

$$
\begin{aligned}
& (\sigma_{\mathtt{ML}} : Heap_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : Heap_{\mathtt{C}}) \\
\text{(run ML code)} \quad & \longrightarrow^* (\sigma_{\mathtt{ML}}' : Heap_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : Heap_{\mathtt{C}})
\end{aligned}
$$

# Language Interaction: Semantics (2)

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

$\sigma : \mathit{Heap}_{\mathtt{ML}}$ $\longleftarrow$ $\zeta : \mathit{BlockHeap}$

switch at the language barrier

Whole program state: ML + C state (+ extra omitted FFI state):

$$
\begin{aligned}
& && (\sigma_{\mathtt{ML}} : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
&\text{(run ML code)} && \longrightarrow^{*} (\sigma_{\mathtt{ML}}{}' : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
&\text{(extcall ML} \to \text{C)} && \longrightarrow (\zeta : \mathit{BlockHeap},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}})
\end{aligned}
$$

# Language Interaction: Semantics (2)

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

$\sigma : Heap_{\mathtt{ML}} \longleftarrow \zeta : BlockHeap$

switch at the language barrier

Whole program state: ML + C state (+ extra omitted FFI state):

$$
\begin{array}{rll}
 & & (\sigma_{\mathtt{ML}} : Heap_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : Heap_{\mathtt{C}}) \\
\text{(run ML code)} & \longrightarrow^{*} & (\sigma_{\mathtt{ML}}' : Heap_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : Heap_{\mathtt{C}}) \\
\text{(extcall ML} \rightarrow \text{C)} & \longrightarrow & (\zeta : BlockHeap,\ \sigma_{\mathtt{C}} : Heap_{\mathtt{C}}) \\
\text{(run C code)} & \longrightarrow^{*} & (\zeta : BlockHeap,\ \sigma_{\mathtt{C}}' : Heap_{\mathtt{C}})
\end{array}
$$

# Language Interaction: Semantics (2)

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

$\sigma : \mathit{Heap}_{\mathtt{ML}} \quad \rightleftarrows \quad \zeta : \mathit{BlockHeap}$

switch at the language barrier

Whole program state: ML + C state (+ extra omitted FFI state):

$$
\begin{array}{rcl}
 & & (\sigma_{\mathtt{ML}} : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(run ML code)} & \longrightarrow^{*} & (\sigma_{\mathtt{ML}}' : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(extcall ML} \to \text{C)} & \longrightarrow & (\zeta : \mathit{BlockHeap},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(run C code)} & \longrightarrow^{*} & (\zeta : \mathit{BlockHeap},\ \sigma_{\mathtt{C}}' : \mathit{Heap}_{\mathtt{C}}) \\
\text{(call FFI op)} & \longrightarrow & (\zeta' : \mathit{BlockHeap},\ \sigma_{\mathtt{C}}' : \mathit{Heap}_{\mathtt{C}})
\end{array}
$$

# Language Interaction: Semantics (2)

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

$\sigma : \mathit{Heap}_{\mathtt{ML}} \quad \leftrightarrows \quad \zeta : \mathit{BlockHeap}$

switch at the language barrier

Whole program state: ML + C state (+ extra omitted FFI state):

$$
\begin{array}{rll}
 & & (\sigma_{\mathtt{ML}} : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(run ML code)} & \longrightarrow^{*} & (\sigma_{\mathtt{ML}}' : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(extcall ML} \to \text{C)} & \longrightarrow & (\zeta : \mathit{BlockHeap},\ \sigma_{\mathtt{C}} : \mathit{Heap}_{\mathtt{C}}) \\
\text{(run C code)} & \longrightarrow^{*} & (\zeta : \mathit{BlockHeap},\ \sigma_{\mathtt{C}}' : \mathit{Heap}_{\mathtt{C}}) \\
\text{(call FFI op)} & \longrightarrow & (\zeta' : \mathit{BlockHeap},\ \sigma_{\mathtt{C}}' : \mathit{Heap}_{\mathtt{C}}) \\
\text{(return from extcall)} & \longrightarrow & (\sigma_{\mathtt{ML}}' : \mathit{Heap}_{\mathtt{ML}},\ \sigma_{\mathtt{C}}' : \mathit{Heap}_{\mathtt{C}})
\end{array}
$$

# Language Interaction: Program Logic, Take 1

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

$\sigma : \mathit{Heap}_{\texttt{ML}} \rightleftarrows \zeta : \mathit{BlockHeap}$

# Language Interaction: Program Logic, Take 1

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

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

$\sigma : \mathit{Heap}_{\texttt{ML}} \rightleftarrows \zeta : \mathit{BlockHeap}$

# Language Interaction: Program Logic, Take 1

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

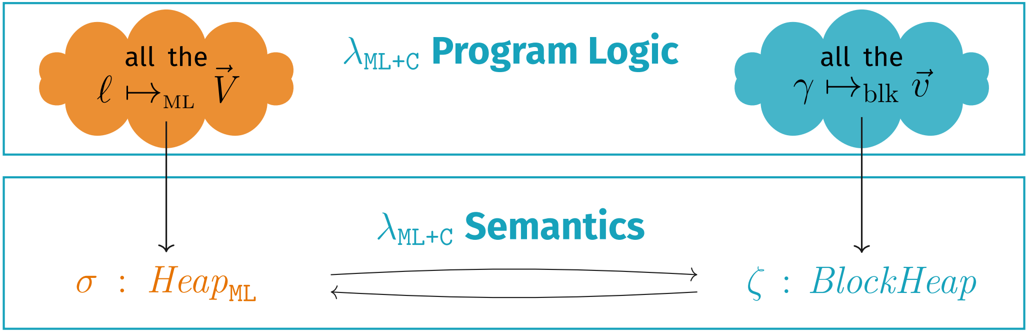
$\ell \mapsto_{\texttt{ML}} \vec{V}$

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

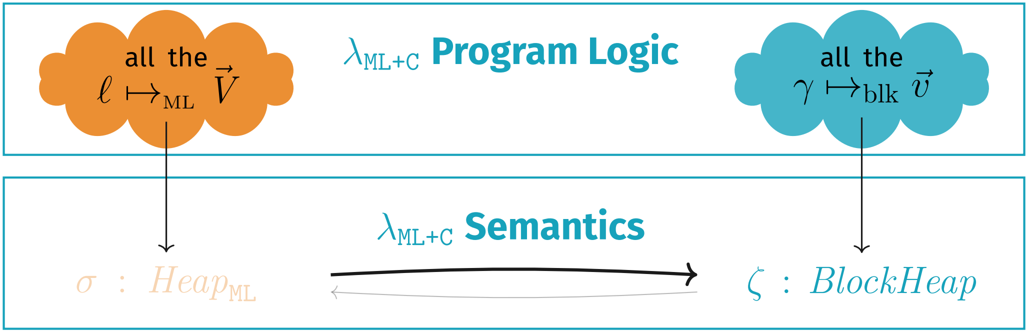
## $\lambda_{\texttt{ML+C}}$ Semantics

$\sigma : \mathit{Heap}_{\texttt{ML}} \rightleftarrows \zeta : \mathit{BlockHeap}$

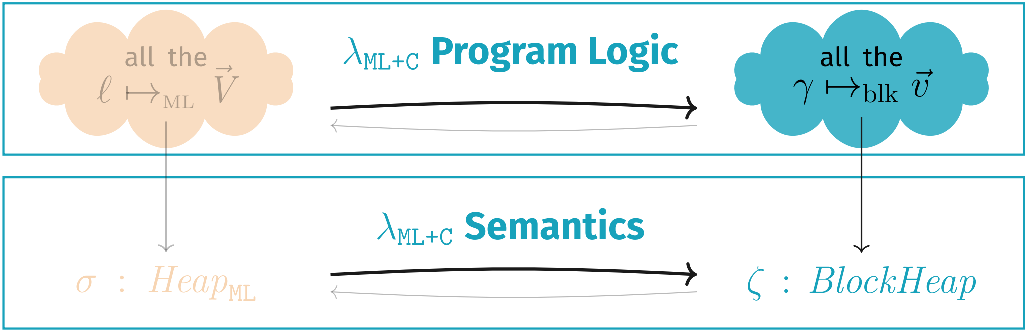
# Language Interaction: Program Logic, Take 1



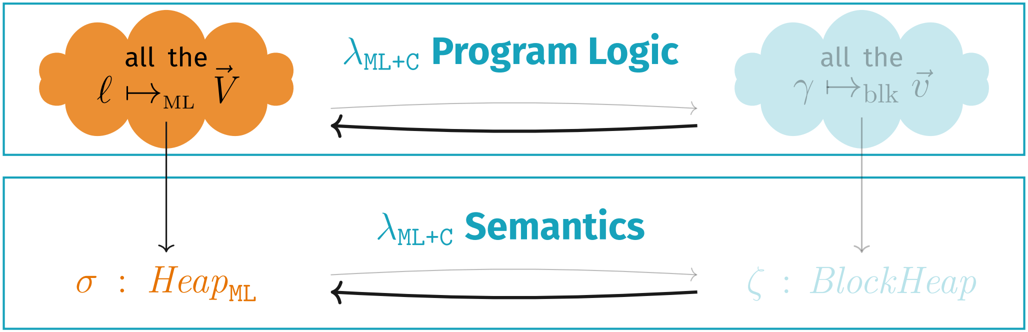
# Language Interaction: Program Logic, Take 1



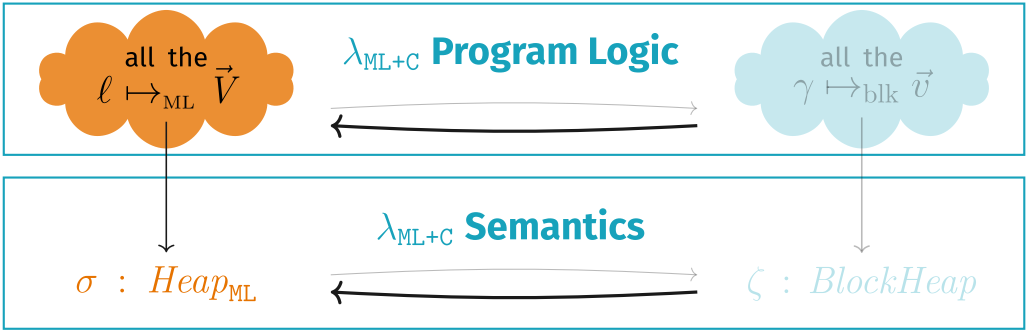
# Language Interaction: Program Logic, Take 1



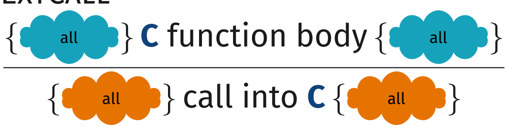
# Language Interaction: Program Logic, Take 1



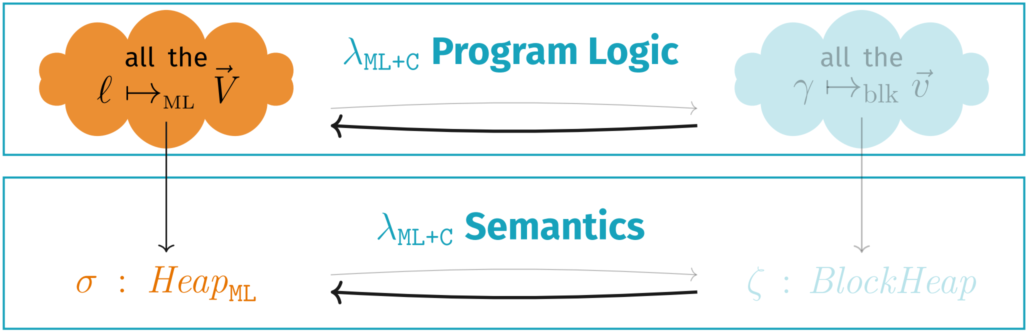
# Language Interaction: Program Logic, Take 1

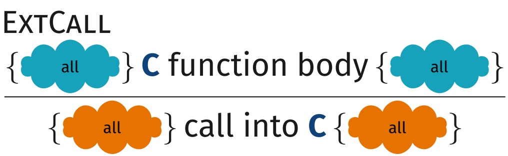



ExtCall



# Language Interaction: Program Logic, Take 1

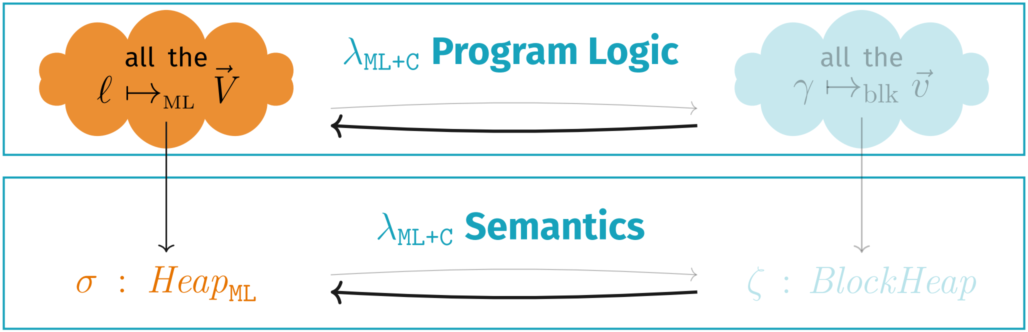



Frame

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

# Language Interaction: Program Logic, Take 1




**ExtCall**

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

---

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

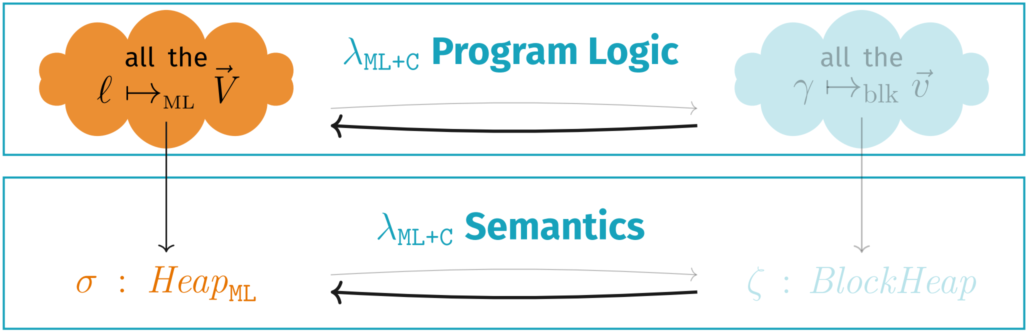
**Frame**

$\{P\}$ call into **C** $\{Q\}$

---

$\{R * P\}$ call into **C** $\{Q * R\}$

# Language Interaction: Program Logic, Take 1




ExtCall

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

---

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

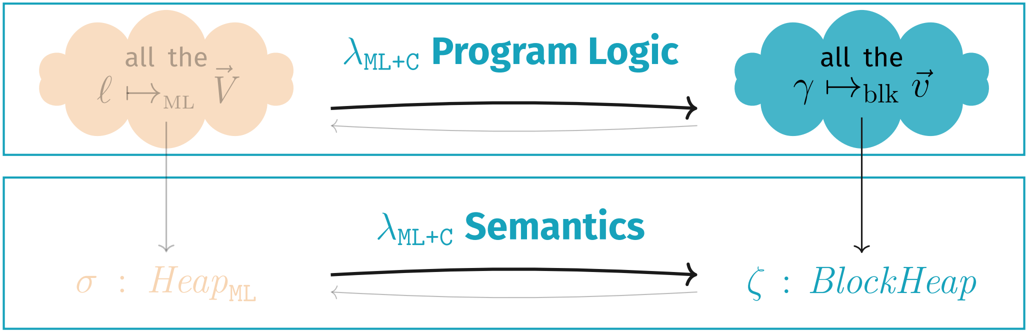
Frame

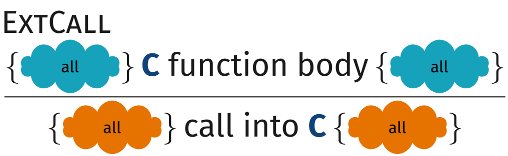
$\{P\}$ call into **C** $\{Q\}$

---

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

# Language Interaction: Program Logic, Take 1

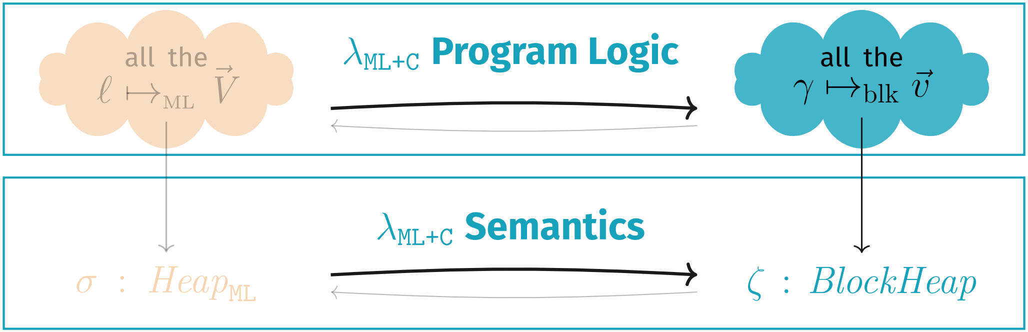


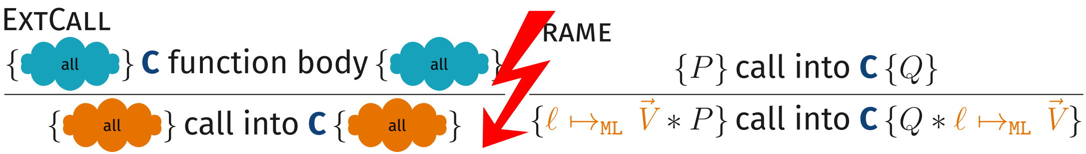




Frame

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

# Language Interaction: Program Logic, Take 1



# 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

{ all } C function body { all }

{ all } call into C { all }

Frame

$\{P\}$ call into C $\{Q\}$

$\{\ell \mapsto_{\text{ML}} \vec{V} * P\}$ 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**!

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

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

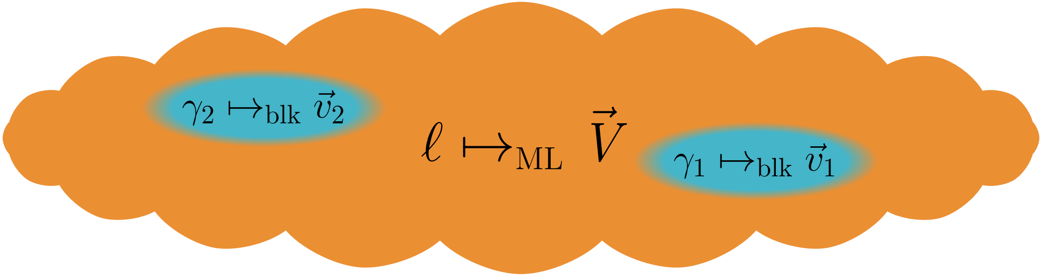
# Language Interaction: More Gradual Rules

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

$$\ell \mapsto_{\mathrm{ML}} \vec{V} \quad \gamma_1 \mapsto_{\mathrm{blk}} \vec{v}_1$$

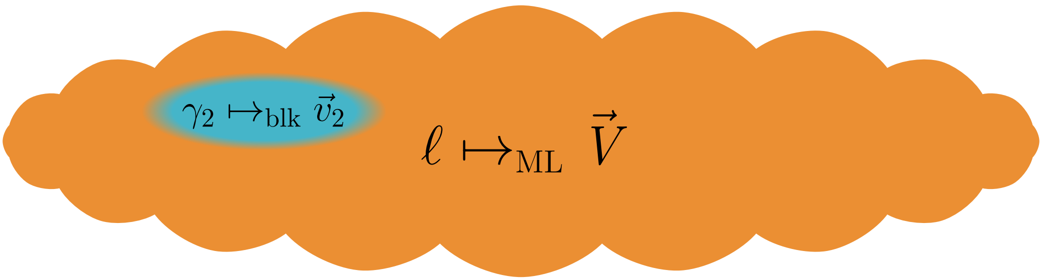
# Language Interaction: More Gradual Rules

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



# Language Interaction: More Gradual Rules

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



# 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

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

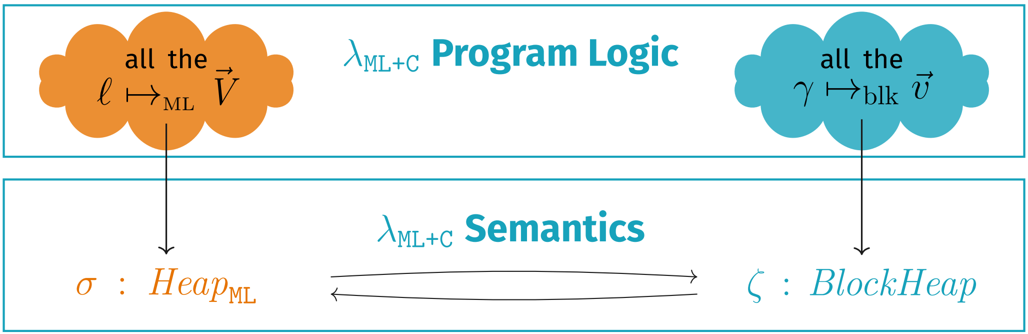
$$\vec{V} \sim_{\mathtt{ML}} \vec{v} * \gamma \mapsto_{\mathrm{blk}} \vec{v} \Rrightarrow\!\!\ast\ \exists \ell\ .\ \ell \mapsto_{\mathtt{ML}} \vec{V} * \ell \sim_{\mathtt{ML}} \gamma$$

# Language Interaction: View Reconciliation

## View Reconciliation Rules

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

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

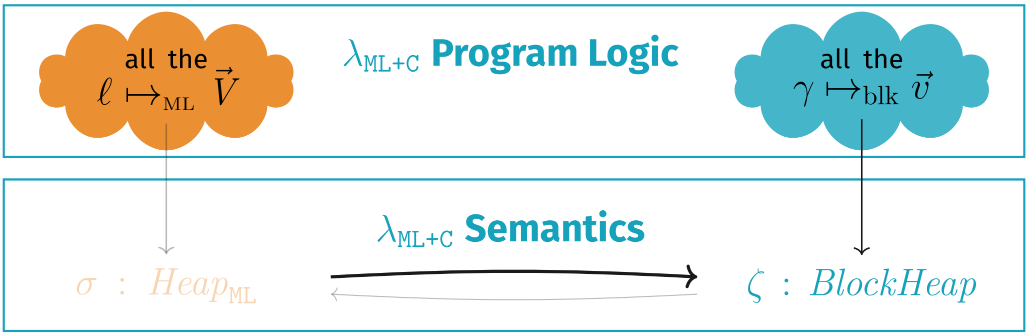


# Language Interaction: View Reconciliation

## View Reconciliation Rules

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

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

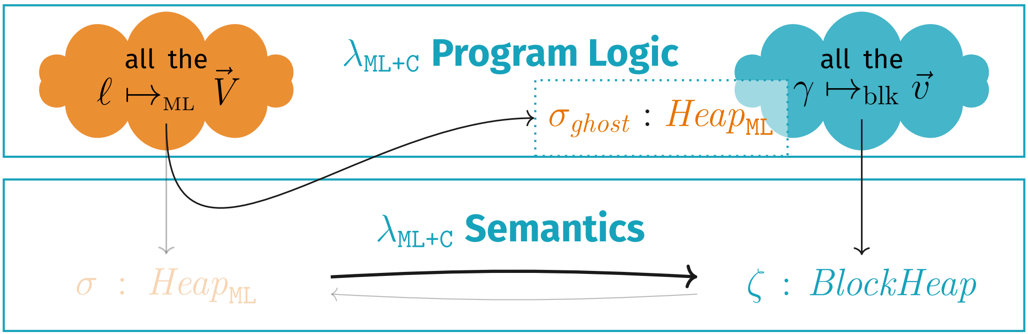


# Language Interaction: View Reconciliation

## View Reconciliation Rules

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

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

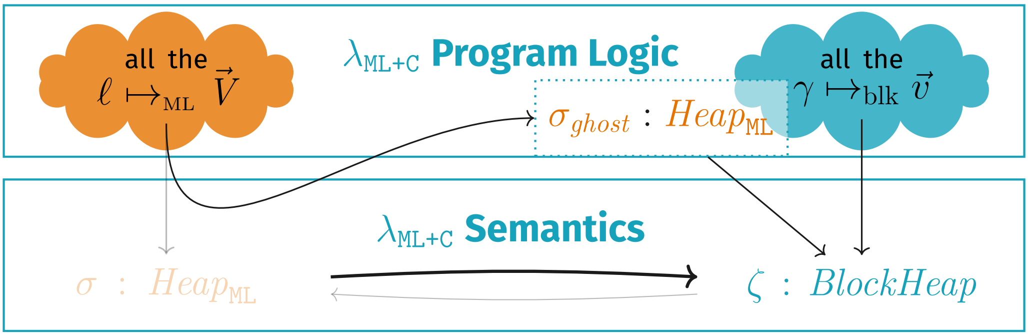


# Language Interaction: View Reconciliation

## View Reconciliation Rules

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

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

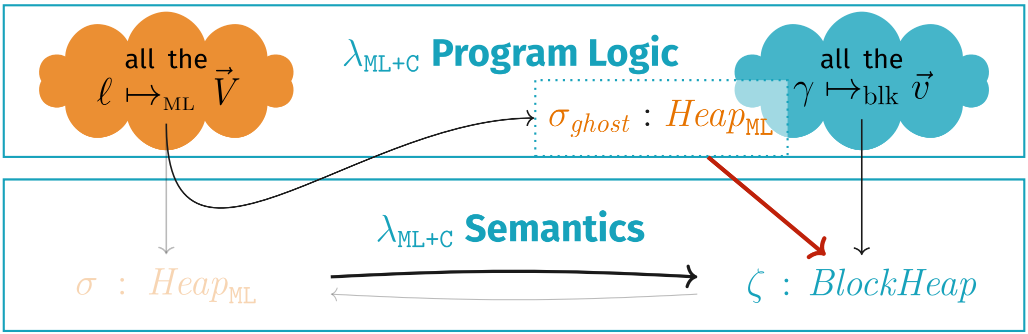


# Language Interaction: View Reconciliation

## View Reconciliation Rules

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

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



# Application: Finishing the Proof for `hash_ref`

# 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"
```

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

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

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

## 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

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

# 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;
```

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

```
}
```

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;
```

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

```
}
```

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)

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

# 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)

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

# A Tour of the Coq Formalization

# Syntax and Semantics

```
Structure language (val : Type) := Language {
  (* small-step operational semantics *)
  expr : Type;
  state : Type;
  head_step : prog → expr → state → expr → state → Prop;

  (* top-level functions *)
  func : Type;
  apply_func : func → list val → option expr;

  (* external call expressions *)
  cont : Type; (* evaluation context *)
  is_call : expr → string → list val → cont → Prop;
  (* ... *)
}.
(* a program is a set of toplevel functions *)
Notation prog Λ := (gmap string Λ.(func)).
```

# Cross-language linking?

`ld -o p.exe p1.o p2.o`

```
Context (val : Type) (Λ1 Λ2 : language val).

Definition p1 : prog Λ1 := {[
  "f1" := Fun ["x"] (... (ExtCall "f2" ["z"]) ...);
]}.
Definition p2 : prog Λ2 := {[
  "f2" := Fun ["y"] (... (ExtCall "f1" ["u"]) ...);
]}.
```

# Cross-language linking?

```
ld -o p.exe p1.o p2.o
```

```
Context (val : Type) (Λ1 Λ2 : language val).

Definition p1 : prog Λ1 := {[
  "f1" := Fun ["x"] (... (ExtCall "f2" ["z"]) ...);
]}.
Definition p2 : prog Λ2 := {[
  "f2" := Fun ["y"] (... (ExtCall "f1" ["u"]) ...);
]}.
```

We wish to **link** `p1` and `p2` together into a program:

- that implements both `"f1"` and `"f2"`
- with no remaining external calls

```
Definition p : prog (* ??? *) := link_prog p1 p2.
```

# Cross-language linking!

```
Definition link_lang {val} (Λ1 Λ2 : language val) :
  language val.

Definition link_prog {val} {Λ1 Λ2 : language val} :
  prog Λ1 → prog Λ2 → prog (link_lang Λ1 Λ2).
```

Idea: a `link_lang` expression is of the form:

$$(e, [k_1; \ldots; k_n])$$

`Λ1.(expr) + Λ2.(expr)`

`list (Λ1.(cont) + Λ2.(cont))`

(Omitted: how we can exchange state between `Λ1` and `Λ2`)

# Linking C and ML

```
Canonical Structure C_lang : language C_val := ...
Canonical Structure ML_lang : language ML_val := ...
```

# Linking C and ML

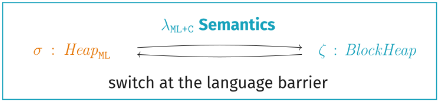
```
Canonical Structure C_lang : language C_val := ...
Canonical Structure ML_lang : language ML_val := ...
```

```
Check (link_lang ML_lang C_lang).
```

```
Error:
The term "C_lang" has type "language C_val"
while it is expected to have type "language ML_val".
```

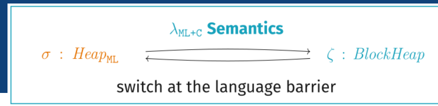
We need to add FFI semantics to translate between ML and C!

# FFI as wrapper semantics




```
(* embeds ML_lang + adds FFI semantics *)
Definition wrap_lang : language C_val.
Definition wrap_prog : ML_lang.(expr) → prog wrap_lang.
```

# FFI as wrapper semantics




```
(* embeds ML_lang + adds FFI semantics *)
Definition wrap_lang : language C_val.
Definition wrap_prog : ML_lang.(expr) → prog wrap_lang.
```

`wrap_prog e` emits:

- same external calls as `e`, translated to use C values/state

`wrap_prog e` implements:

- FFI operations
- a `main()` function that runs `e`

# Our full multi-language semantics


```
Notation combined_lang := (link_lang wrap_lang C_lang).
Definition combined_prog (e: prog ML_lang) (p: prog C_lang) :=
  link_prog (wrap_prog e) p.
```

# Program logic Building Blocks

$$\Psi \vDash p : \Pi$$

"**assuming** interface $\Psi$, program $p$ **implements** interface $\Pi$"

# Program logic Building Blocks

$$\Psi \vDash p : \Pi$$

"**assuming** interface $\Psi$, program $p$ **implements** interface $\Pi$"

```
Lemma link_correct p1 p2 Ψ1 Ψ2 :
  dom p1 ## dom p2 →
  Π ⊨ p1 :: Ψ →
  Ψ ⊨ p2 :: Π →
  ∅ ⊨ link_prog p1 p2 :: Ψ ⊔ Π.
```

# Program logic Building Blocks

$$\Psi \vDash p : \Pi$$

"**assuming** interface $\Psi$, program $p$ **implements** interface $\Pi$"

```
Lemma link_correct p1 p2 Ψ1 Ψ2 :
  dom p1 ## dom p2 →
  Π ⊨ p1 :: Ψ →
  Ψ ⊨ p2 :: Π →
  ∅ ⊨ link_prog p1 p2 :: Ψ ⊔ Π.
```

```
Lemma wrap_correct e Ψ :
  Ψ on prim_names ⊑ ⊥ →
  { True } e @ Ψ { True } →
  wrap_intf Ψ ⊨ wrap_prog e :: prims_intf Ψ ⊔ main_intf.
```

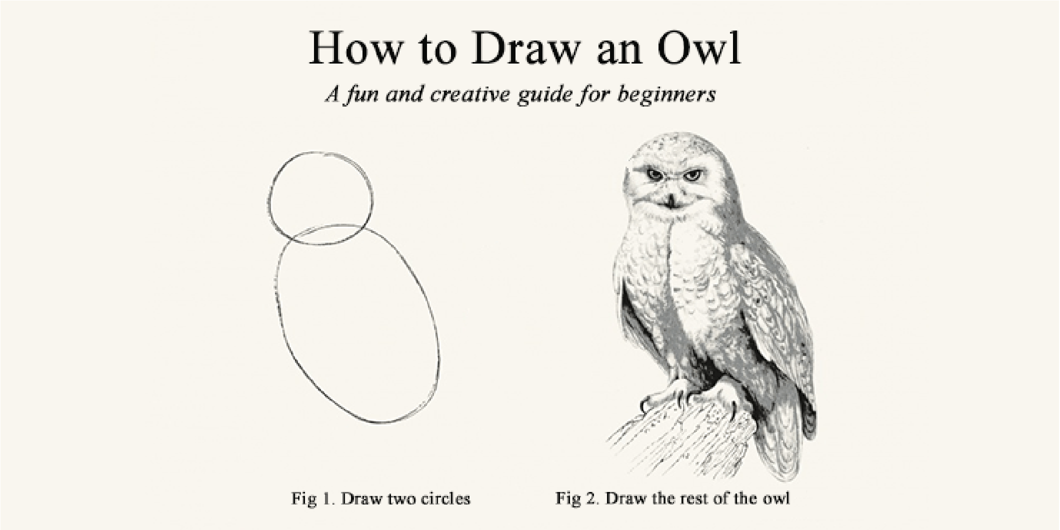
# Adequacy Theorem

```
Lemma adequacy p :
  ∅ ⊨ p :: main_intf →
  is_safe p (call "main" (), σ_init)
```

Converts correctness **in the logic** into safety **in the semantics**

# Conclusion:

# How to Build Melocoton: Key Ideas (recap) And The Rest



# Key Ideas

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

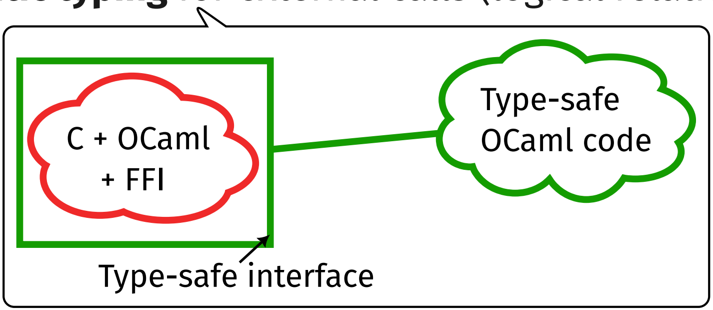
- Abstract over "the other side" using **interfaces and external calls**
- Formalize the **semantics of the FFI**
- Bridge between memory models using **view reconciliation**

# Also in Melocoton...

- Use **angelic nondeterminism** in the FFI semantics ($\mathit{BlockHeap} \to \mathit{Heap}_{\mathtt{ML}}$ step). Requires Transfinite Iris.
- **More detailed FFI**: GC interaction, callbacks, custom blocks
- **Semantic typing** for external calls (logical relation)

# Also in Melocoton...

- Use **angelic nondeterminism** in the FFI semantics ($BlockHeap \to Heap_{\mathrm{ML}}$ step). Requires Transfinite Iris.
- **More detailed FFI**: GC interaction, callbacks, custom blocks
- **Semantic typing** for external calls (logical relation)



# Future Work

## Planned/Ongoing

- Extend Melocoton with remaining OCaml 4 FFI features
- Static analysis tool for FFI glue code

## Ideas

- Model the Multicore OCaml FFI
- Verification/bug finding/runtime analysis for FFI code
- Domain-specific language for FFI with built-in verification?
- Reusable Iris libraries for multi-language program logics?

# Melocoton

## Language Locality: Embed Existing Languages

| OCaml Program Logic | $\lambda_{\mathrm{ML+C}}$ **Program Logic**<br>Glue Code Verification | C Program Logic |
|---|---|---|
| **OCaml Semantics** | $\lambda_{\mathrm{ML+C}}$ **Semantics**<br>Glue Code Semantics | **C Semantics** |

## Language Interaction: View Reconciliation Rules

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

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

https://melocoton-project.github.io