

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



MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS



AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Multi-Language Programs Are Everywhere



Python

C

Fortran



C++

Rust

JavaScript



C

Bindings for:

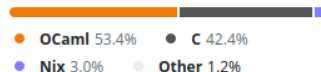
- Rust
- Python
- OCaml
- Go
- ...

Multi-Language Programs Are Everywhere

☰ README.md

OCaml-SSL - OCaml bindings for the libssl

Languages



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

Go

...

Mind the gap!



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.

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?

The background of the slide features a collage of several academic papers, tilted at various angles. The papers are mostly white with black text and some colored accents. One paper at the top left has a green and red logo. Another paper on the right has a green and red logo. The papers appear to be related to computer science, specifically focusing on semantics, logic, and program verification.

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**

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



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



C glue code

```
 $\{r \mapsto_{ML} n\}$   
external hash_ref: int ref -> unit  
    = "caml_hash_ref"  
 $\{r \mapsto_{ML} m\}$ 
```

```
 $\{\gamma \mapsto_{blk[0|mut]} [n]\}$   
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;  
}  
 $\{\gamma \mapsto_{blk[0|mut]} [m]\}$ 
```


A Multi-Language Program Logic for FFI

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

OCaml glue code

```
 $\{r \mapsto_{ML} n\}$   
external hash_ref: int  
  = "caml_hash_ref"  
 $\{r \mapsto_{ML} m\}$ 
```



C glue code

```
 $[n]$   
ref(value r) {  
  val(Field(r, 0));  
  ;  
  (r, 0, Val_int(x));  
  _unit;  
   $[m]$ 
```

We Have A Tool For That: It Is Called 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”?

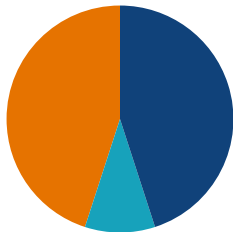
One Big Language:

unsatisfactory for **engineering** and **conceptual** reasons



Most multi-language programs look like this:

OCaml business logic
oblivious of C

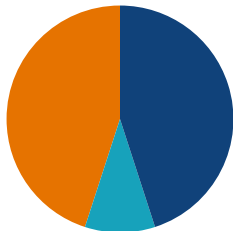


C business logic
oblivious of OCaml

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

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

Glue Code Verification

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

OCaml* Program Logic

λ_{ML+C} **Program Logic**

Glue Code Verification

C* Program Logic

OCaml* Semantics

λ_{ML+C} **Semantics**

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**

Our Contribution: Melocoton

OCaml* Program Logic

λ_{ML+C} Program Logic

Glue Code Verification

C* Program Logic

OCaml* Semantics

λ_{ML+C} Semantics

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



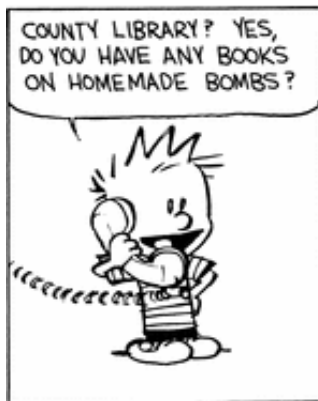
Transfinite



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

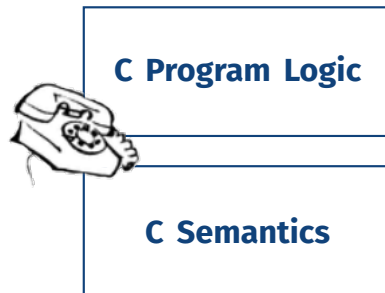
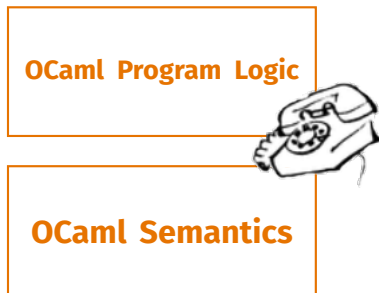
OCaml Semantics

C Program Logic

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: $\{r \mapsto_{\text{ML}} n\} \text{hash_ref}(r) \{\exists m. r \mapsto_{\text{ML}} m\}$

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

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

FFI Operations are External Calls for 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 Ψ_{FFI} and FFI
assertions e.g. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$

FFI Operations are External Calls for 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 Ψ_{FFI} and FFI
assertions e.g. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$

$$\Psi_{\text{FFI}} \triangleq \langle \gamma \mapsto_{\text{blk}[t|\text{mut}]} [\dots v_i \dots] \rangle \text{Store_field}(\gamma, i, v') \langle \gamma \mapsto_{\text{blk}[t|\text{mut}]} [\dots v' \dots] \rangle$$

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

FFI Operations are External Calls for 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 Ψ_{FFI} and FFI
assertions e.g. $\gamma \mapsto_{\text{blk}[t|m]} \vec{v}$

$$\Psi_{\text{FFI}} \triangleq \langle \gamma \mapsto_{\text{blk}[t|\text{mut}]} [\dots v_i \dots] \rangle \text{Store_field}(\gamma, i, v') \langle \gamma \mapsto_{\text{blk}[t|\text{mut}]} [\dots v' \dots] \rangle$$

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

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

$$\{ \gamma \mapsto_{\text{blk}[0|\text{mut}]} [n] \} \text{caml_hash_ref}(r) @ \Psi_{\text{FFI}} \{ \exists m. \gamma \mapsto_{\text{blk}[0|\text{mut}]} [m] \}$$

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;  
  print_int !r
```



in C)*

C business logic

```
void hash_ptr {  
  // Implement OpenSSL  
  // tedious to OCaml  
}
```




OCaml glue code

```
external hash_ptr of: int ref -> unit  
= "caml_hash_ptr"
```

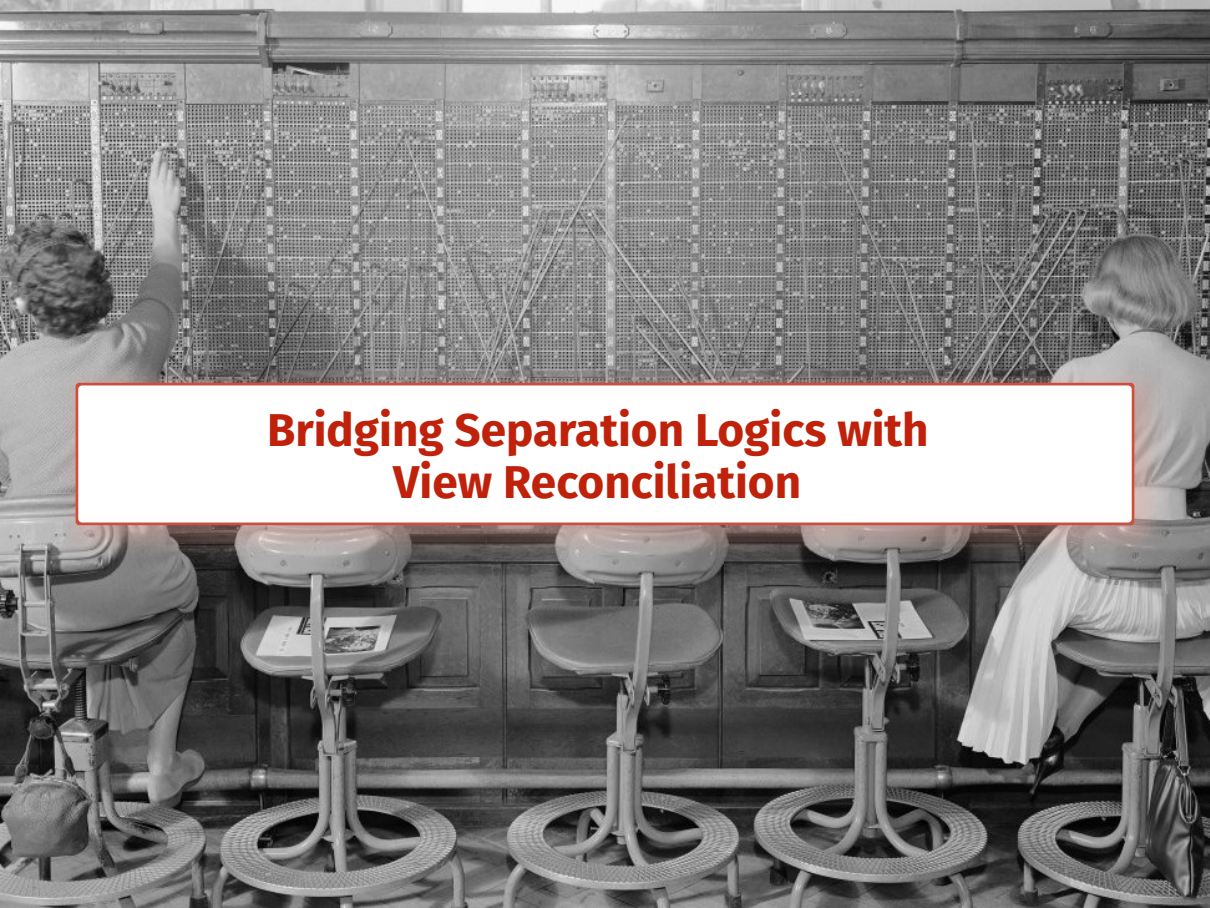


C glue code

```
value caml_hash_ptr (value r) {  
  int x = hash_ptr(r, 0);  
  hash_ptr(r, 0);  
  Store_int(r, Val_int(x));  
  return Val_unit;  
}
```



Missing: connecting the semantics and proofs!



Bridging Separation Logics with View Reconciliation

We assumed:

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

We proved:

```
 $\{\gamma \mapsto_{\text{blk}[0|\text{mut}]} [n]\}$   
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;  
}  
 $\{\exists m. \gamma \mapsto_{\text{blk}[0|\text{mut}]} [m]\}$ 
```

We assumed:

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



We proved:

```
{ $\gamma \mapsto_{\text{blk}[0|\text{mut}]} [n]$ }  
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;  
}  
{ $\exists m. \gamma \mapsto_{\text{blk}[0|\text{mut}]} [m]$ }
```

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_{ML} Low-level **block** representation

Language Interaction: Semantics

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

integers \sim_{ML} integers

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

true \sim_{ML} *1*

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

true \sim_{ML} 1

ℓ \sim_{ML} *γ*

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)

true \sim_{ML} 1

ℓ \sim_{ML} *γ*

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

true \sim_{ML} 1

ℓ \sim_{ML} *γ*

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

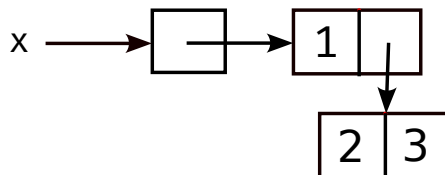
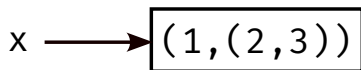
pairs \sim_{ML} blocks (of size 2)

lists \sim_{ML} block-based linked lists

true \sim_{ML} 1

ℓ \sim_{ML} γ

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



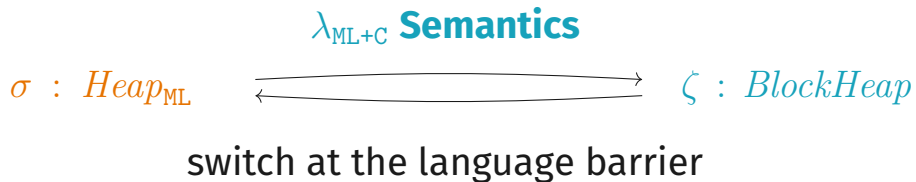
Language Interaction: Semantics (2)

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

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

$\zeta : \text{BlockHeap}$

Language Interaction: Semantics (2)



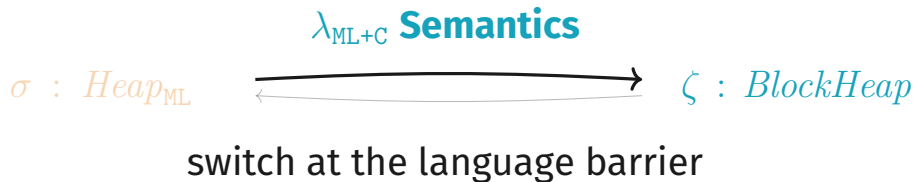
Language Interaction: Semantics (2)

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

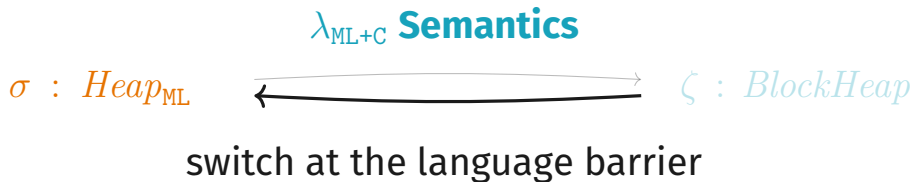
$\sigma : \text{Heap}_{\text{ML}}$ \longleftrightarrow $\zeta : \text{BlockHeap}$

switch at the language barrier

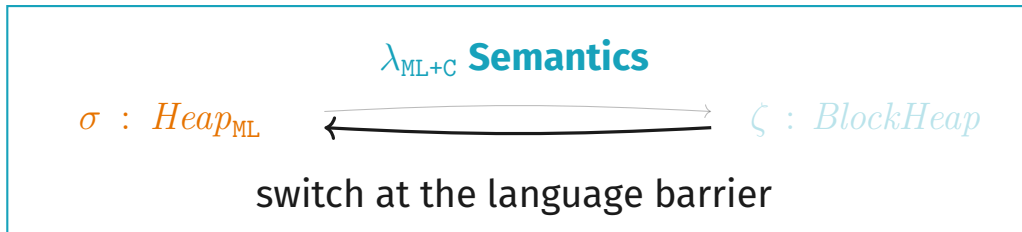
Language Interaction: Semantics (2)



Language Interaction: Semantics (2)



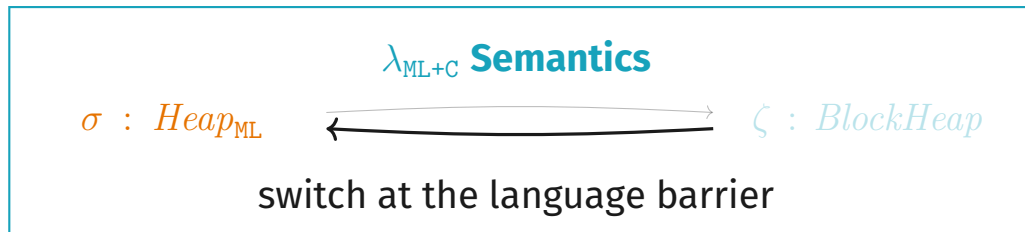
Language Interaction: Semantics (2)



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

$$\begin{array}{l} (\sigma_{\text{ML}} : \textit{Heap}_{\text{ML}}, \sigma_{\text{C}} : \textit{Heap}_{\text{C}}) \\ \text{(run ML code)} \quad \longrightarrow^* (\sigma_{\text{ML}}' : \textit{Heap}_{\text{ML}}, \sigma_{\text{C}} : \textit{Heap}_{\text{C}}) \end{array}$$

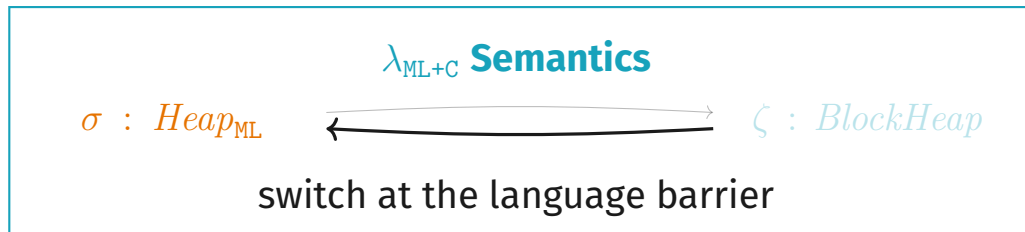
Language Interaction: Semantics (2)



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

$$\begin{array}{ll} & (\sigma_{\text{ML}} : \textit{Heap}_{\text{ML}}, \sigma_{\text{C}} : \textit{Heap}_{\text{C}}) \\ \text{(run ML code)} & \longrightarrow^* (\sigma_{\text{ML}}' : \textit{Heap}_{\text{ML}}, \sigma_{\text{C}} : \textit{Heap}_{\text{C}}) \\ \text{(extcall ML} \rightarrow \text{C)} & \longrightarrow (\zeta : \textit{BlockHeap}, \sigma_{\text{C}} : \textit{Heap}_{\text{C}}) \end{array}$$

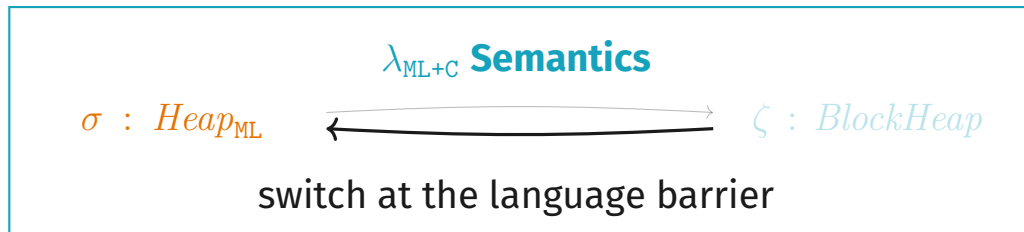
Language Interaction: Semantics (2)



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

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

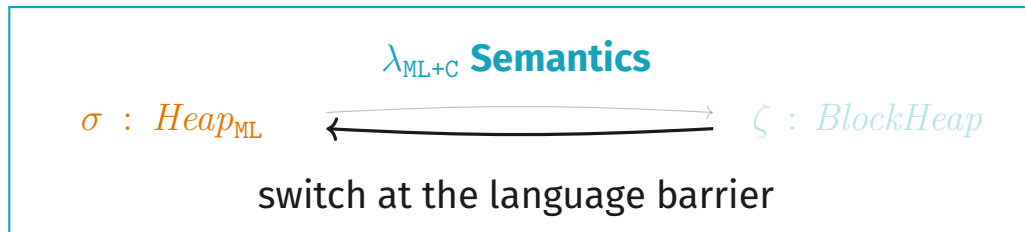
Language Interaction: Semantics (2)



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

	$(\sigma_{\text{ML}} : \text{Heap}_{\text{ML}}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(run ML code)	$\longrightarrow^* (\sigma_{\text{ML}}' : \text{Heap}_{\text{ML}}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(extcall ML \rightarrow C)	$\longrightarrow (\zeta : \text{BlockHeap}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(run C code)	$\longrightarrow^* (\zeta : \text{BlockHeap}, \sigma_{\text{C}}' : \text{Heap}_{\text{C}})$
(call FFI op)	$\longrightarrow (\zeta' : \text{BlockHeap}, \sigma_{\text{C}}' : \text{Heap}_{\text{C}})$

Language Interaction: Semantics (2)



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

	$(\sigma_{\text{ML}} : \text{Heap}_{\text{ML}}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(run ML code)	$\longrightarrow^* (\sigma_{\text{ML}}' : \text{Heap}_{\text{ML}}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(extcall ML \rightarrow C)	$\longrightarrow (\zeta : \text{BlockHeap}, \sigma_{\text{C}} : \text{Heap}_{\text{C}})$
(run C code)	$\longrightarrow^* (\zeta : \text{BlockHeap}, \sigma_{\text{C}}' : \text{Heap}_{\text{C}})$
(call FFI op)	$\longrightarrow (\zeta' : \text{BlockHeap}, \sigma_{\text{C}}' : \text{Heap}_{\text{C}})$
(return from extcall)	$\longrightarrow (\sigma_{\text{ML}}' : \text{Heap}_{\text{ML}}, \sigma_{\text{C}}' : \text{Heap}_{\text{C}})$

Language Interaction: Program Logic, Take 1

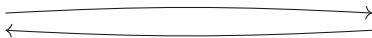


Language Interaction: Program Logic, Take 1

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

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

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



$\zeta : \text{BlockHeap}$

Language Interaction: Program Logic, Take 1

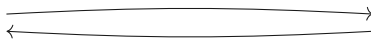
$$\ell \mapsto_{\text{ML}} \vec{V}$$

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

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

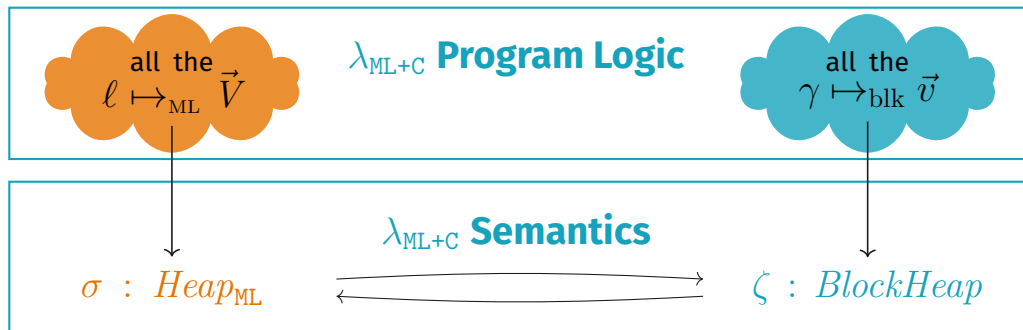
$$\sigma : \text{Heap}_{\text{ML}}$$

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

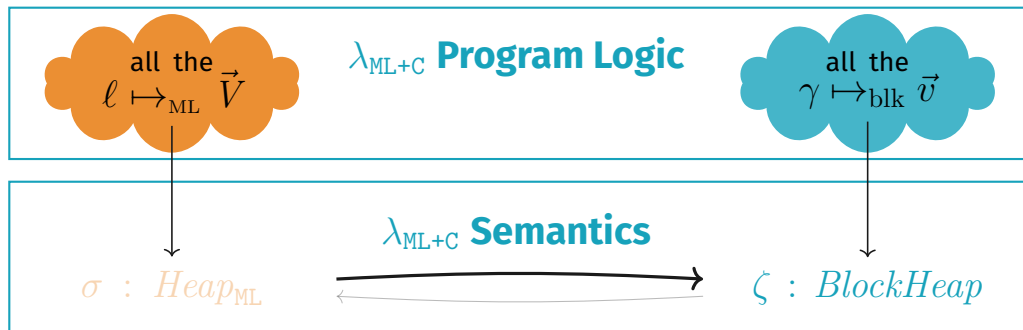


$$\zeta : \text{BlockHeap}$$

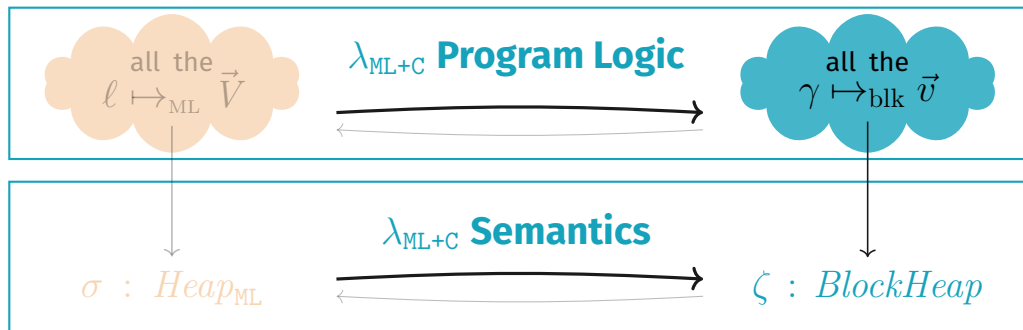
Language Interaction: Program Logic, Take 1



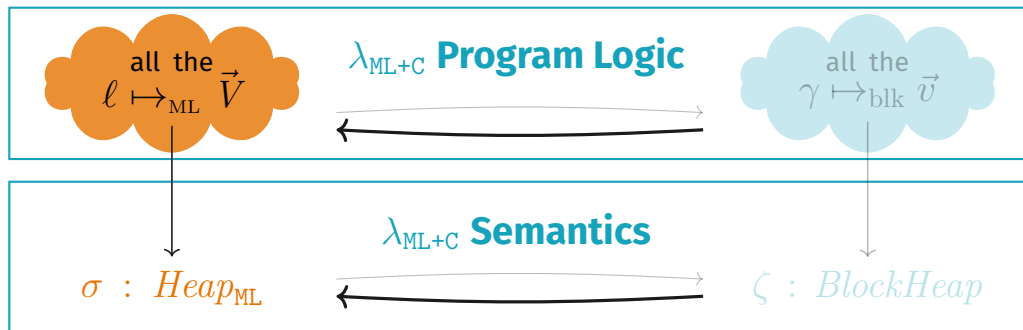
Language Interaction: Program Logic, Take 1



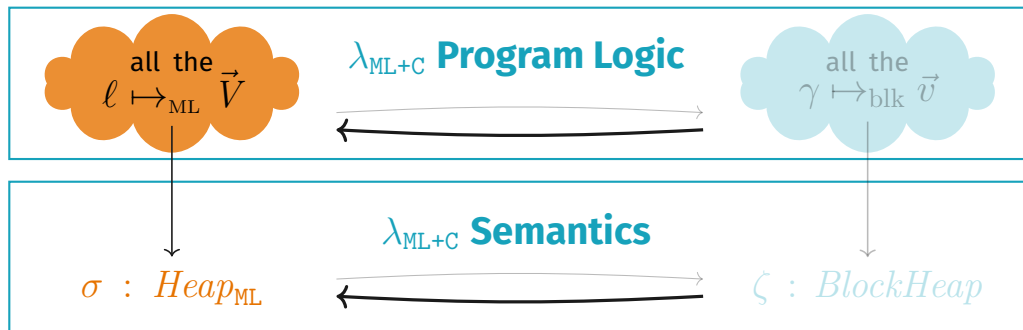
Language Interaction: Program Logic, Take 1



Language Interaction: Program Logic, Take 1



Language Interaction: Program Logic, Take 1

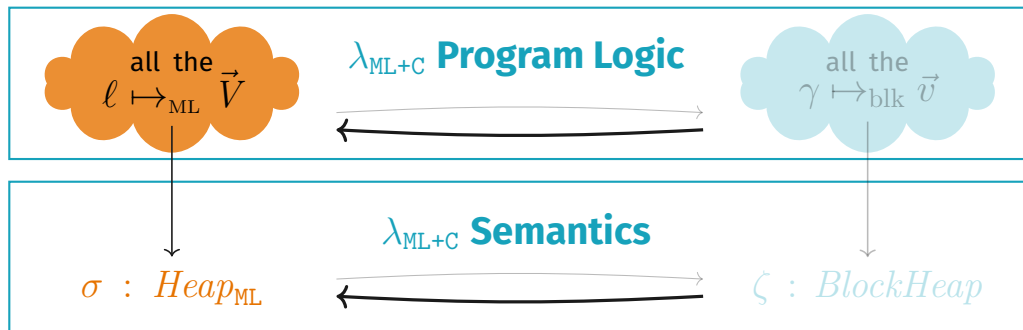


EXTCALL

$\{\text{all}\} \text{ C function body } \{\text{all}\}$

 $\{\text{all}\} \text{ call into C } \{\text{all}\}$

Language Interaction: Program Logic, Take 1



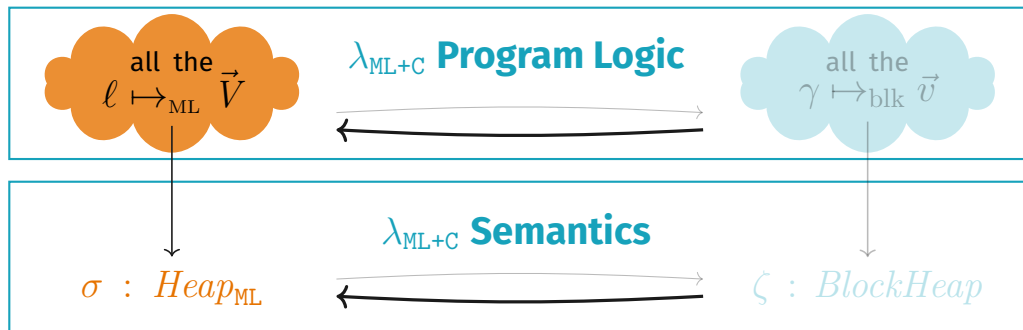
EXTCALL

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

FRAME

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

Language Interaction: Program Logic, Take 1



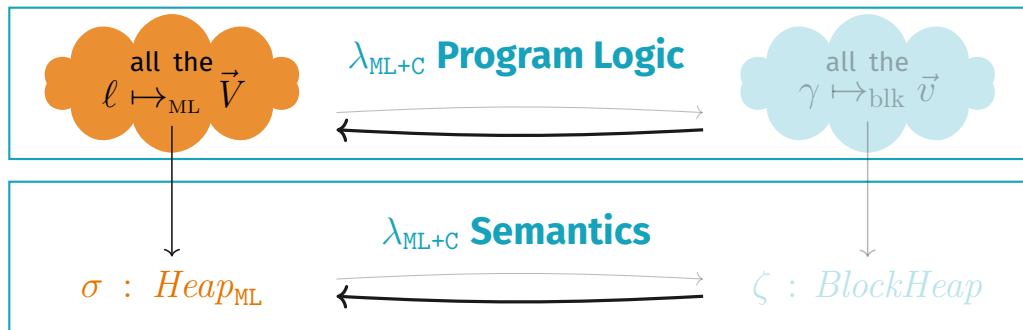
EXTCALL

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

FRAME

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

Language Interaction: Program Logic, Take 1



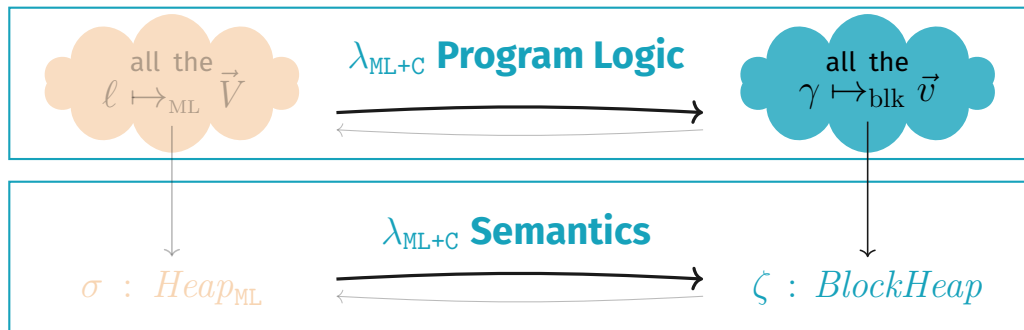
EXTCALL

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

FRAME

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

Language Interaction: Program Logic, Take 1



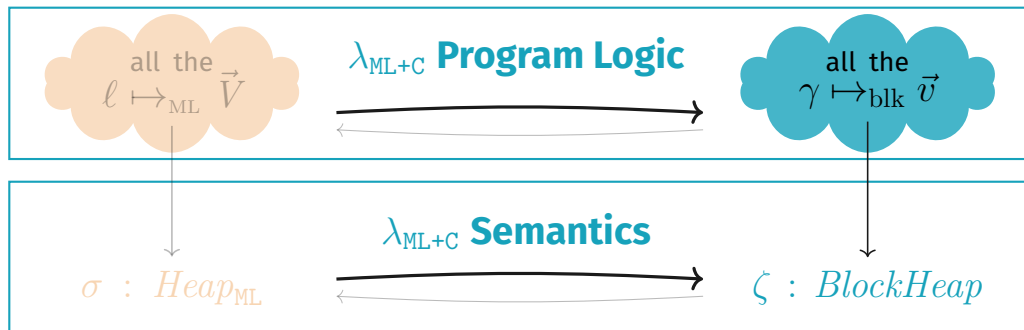
EXTCALL

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

FRAME

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

Language Interaction: Program Logic, Take 1



EXTCALL

$\{\text{all}\} \mathbf{C}$ function body $\{\text{all}\}$

RAME

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

$\{\text{all}\}$ call into \mathbf{C} $\{\text{all}\}$

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

Language Interaction: Program Logic, Take 1

λ_{ML+C} Program Logic

The λ_{ML+C} Semantics operate **globally** on the state



The λ_{ML+C} Program Logic needs **local** reasoning rules

EXTCALL

$\{\text{all}\} \text{ C function body } \{\text{all}\}$

RAME

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

$\{\text{all}\} \text{ call into C } \{\text{all}\}$

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

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**!

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

Language Interaction: More Gradual Rules

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


$$\ell \mapsto_{\text{ML}} \vec{V} \quad \ell_1 \mapsto_{\text{ML}} \vec{V}_1$$

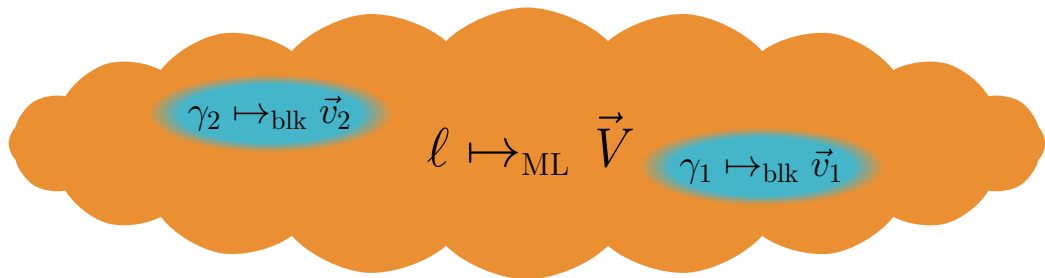
Language Interaction: More Gradual Rules

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

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

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**!

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

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

Language Interaction: More Gradual Rules

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

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

Language Interaction: More Gradual Rules

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

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

View Reconciliation Rules for Converting On-Demand:

$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$

Language Interaction: View Reconciliation

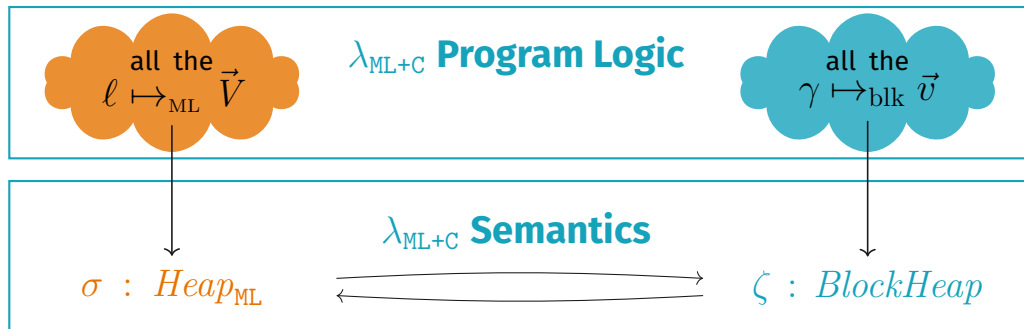
View Reconciliation Rules

$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$

Language Interaction: View Reconciliation

View Reconciliation Rules

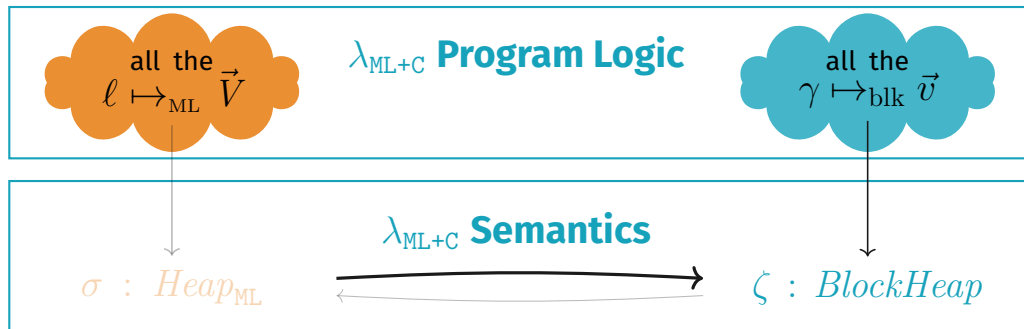
$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$



Language Interaction: View Reconciliation

View Reconciliation Rules

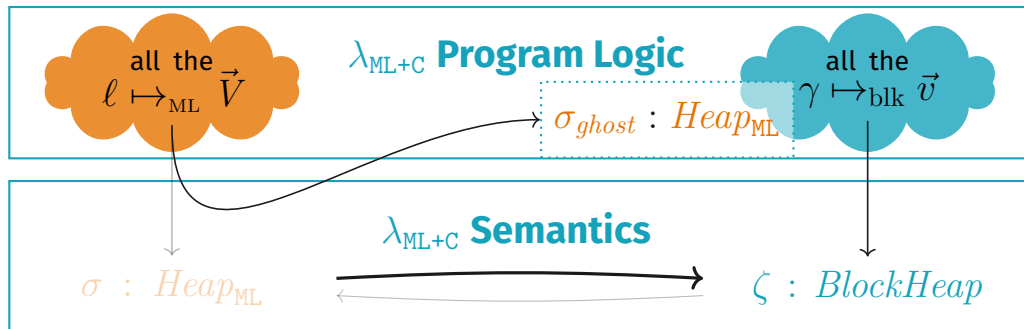
$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$



Language Interaction: View Reconciliation

View Reconciliation Rules

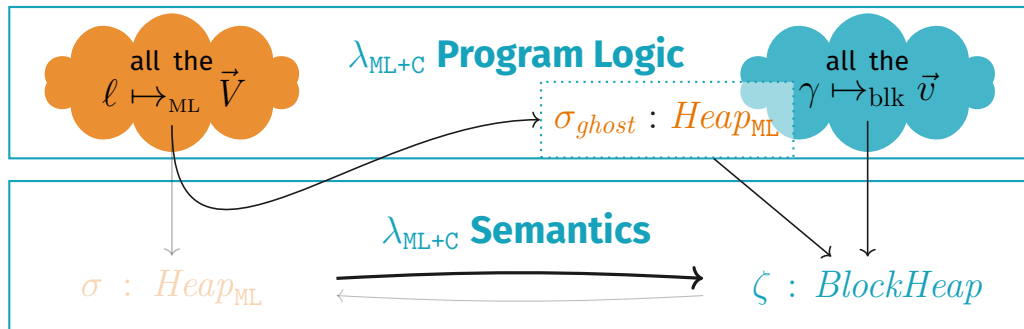
$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$



Language Interaction: View Reconciliation

View Reconciliation Rules

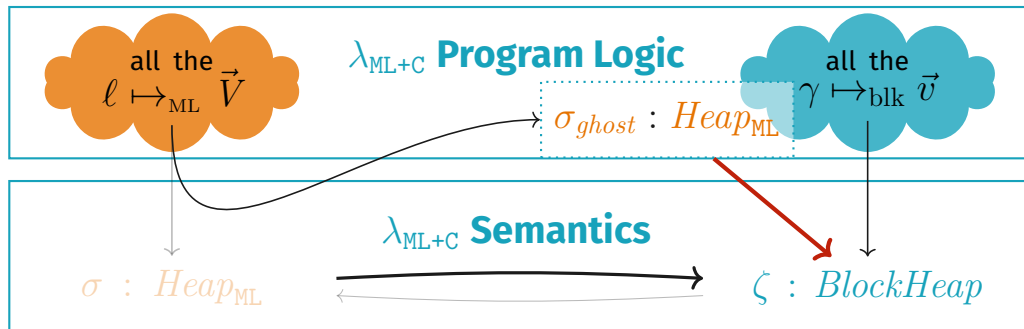
$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$



Language Interaction: View Reconciliation

View Reconciliation Rules

$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\equiv * \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\equiv * \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$



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"
```

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

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"
```

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

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

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

Verifying hash_ref with Melocoton

OCaml glue code

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

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

C glue code

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

EXTCALL

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

Verifying hash_ref with Melocoton

OCaml glue code

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

{r ↦ML n}
  hash_ref(r)
{∃m. r ↦ML m}
```

C glue code

```
value caml_hash_ref(value v) {
  {r ↦ML 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. r ↦ML m * () ∼ML Val_unit}
}
```

EXTCALL

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

Verifying hash_ref with Melocoton

OCaml glue code

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

{r ↦ML n}
  hash_ref(r)
{∃m. r ↦ML m}
```

C glue code

```
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. r ↦ML m * ()} ~ML Val_unit
}
```

VIEW RECONCILIATION (1)

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

Verifying hash_ref with Melocoton

OCaml glue code

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

{r ↦ML n}
  hash_ref(r)
{∃m. r ↦ML m}
```

C glue code

```
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)

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

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  $\Lambda$  := (gmap string  $\Lambda$ .(func)).
```



```
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"]) ...);  
]}.
```

```
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

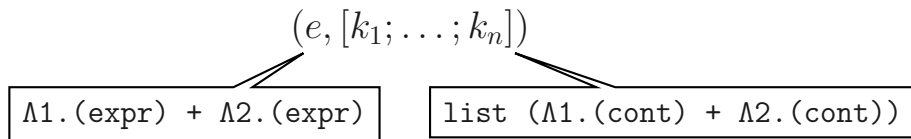
```
Definition p : prog (*???) := link_prog p1 p2.
```

Cross-language linking!

Definition `link_lang {val} (Λ_1 Λ_2 : language val) :`
`language val.`

Definition `link_prog {val} { Λ_1 Λ_2 : language val} :`
`prog Λ_1 \rightarrow prog Λ_2 \rightarrow prog (link_lang Λ_1 Λ_2).`

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



(Omitted: how we can exchange state between Λ_1 and Λ_2)

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 := ...
```

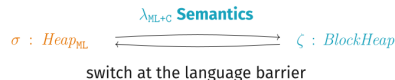
```
Check (link_lang ML_lang C_lang).
```

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

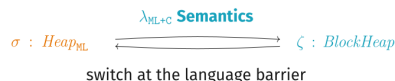


(embeds ML_lang + adds FFI semantics *)*

Definition wrap_lang : language C_val.

Definition wrap_prog : ML_lang.(expr) \rightarrow 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

OCaml* Semantics

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

C* 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

$$\Psi \models p : \Pi$$

“**assuming** interface Ψ , program p **implements** interface Π ”

Program logic Building Blocks

$$\Psi \models p : \Pi$$

“assuming interface Ψ , program p **implements** interface Π ”

```
Lemma link_correct p1 p2  $\Psi_1 \Psi_2$  :  
  dom p1 ## dom p2  $\rightarrow$   
   $\Pi \models p1 :: \Psi \rightarrow$   
   $\Psi \models p2 :: \Pi \rightarrow$   
   $\emptyset \models \text{link\_prog } p1 \ p2 :: \Psi \sqcup \Pi.$ 
```

Program logic Building Blocks

$$\Psi \models p : \Pi$$

“**assuming** interface Ψ , program p **implements** interface Π ”

```
Lemma link_correct p1 p2  $\Psi_1 \Psi_2$  :  
  dom p1 ## dom p2  $\rightarrow$   
   $\Pi \models p1 :: \Psi \rightarrow$   
   $\Psi \models p2 :: \Pi \rightarrow$   
   $\emptyset \models \text{link\_prog } p1 \ p2 :: \Psi \sqcup \Pi.$ 
```

```
Lemma wrap_correct e  $\Psi$  :  
   $\Psi$  on prim_names  $\sqsubseteq \perp \rightarrow$   
  { True } e @  $\Psi$  { True }  $\rightarrow$   
  wrap_intf  $\Psi \models \text{wrap\_prog } e :: \text{prims\_intf } \Psi \sqcup \text{main\_intf}.$ 
```

Adequacy Theorem

```
Lemma adequacy p :  
   $\emptyset \models p :: \text{main\_intf} \rightarrow$   
  is\_safe p (call "main" (),  $\sigma_{init}$ )
```

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

Conclusion:

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

How to Draw an Owl

A fun and creative guide for beginners



Fig 1. Draw two circles



Fig 2. Draw the rest of the owl

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

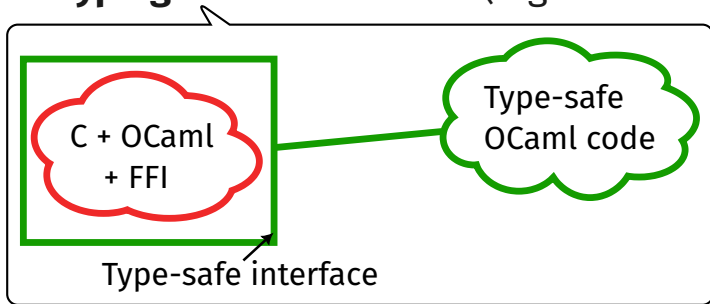
- Abstract over “the other side” using **interfaces and external calls**
- Formalize the **semantics of the FFI**
- Bridge between memory models using **view reconciliation**

Also in Melocoton...

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



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?

Language Locality: Embed Existing Languages

OCaml Program Logic

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

C Program Logic

OCaml Semantics

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

C Semantics

Language Interaction: View Reconciliation Rules

$$\begin{aligned} \ell \mapsto_{\text{ML}} \vec{V} &\Rightarrow \exists \gamma \vec{v}. \gamma \mapsto_{\text{blk}} \vec{v} * \ell \sim_{\text{ML}} \gamma * \vec{V} \sim_{\text{ML}} \vec{v} \\ \vec{V} \sim_{\text{ML}} \vec{v} * \gamma \mapsto_{\text{blk}} \vec{v} &\Rightarrow \exists \ell. \ell \mapsto_{\text{ML}} \vec{V} * \ell \sim_{\text{ML}} \gamma \end{aligned}$$

<https://melocoton-project.github.io>