Melocoton: A Program Logic for Verified Interoperability Between OCaml and C

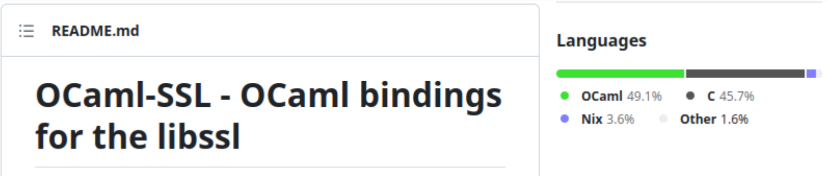
Armaël Guéneau Johannes Hostert Simon Spies Michael Sammler
Lars Birkedal Derek Dreyer

Sept 11, 2023

Many Real Programs Are Multi-Language

Consider the *ocaml-ssl* library:

- Exposes `OpenSSL` (a C library) as an OCaml library
- To do so, it is implemented using a mix of *both* OCaml *and* C code:



Mind the gap!

OCaml **C**

Mind the gap!

OCaml

Structured values

$$\boxed{\lambda_{\texttt{ML}}} \quad V \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \mathit{Loc}) \mid \textsf{true} \mid \textsf{false} \mid \langle\rangle \mid \langle V, V\rangle \mid \lambda x.\, e \;\cdots$$

Garbage collection

C

Integers and pointers

$$\boxed{\lambda_{\texttt{C}}} \quad w \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (a \in \mathit{Addr})$$

Manual memory management

Mind the gap!

OCaml ⟷ **OCaml FFI** ⟷ **C**

Structured values

$\boxed{\lambda_{\text{ML}}}$ $V \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (\ell \in \mathit{Loc})$
$\mid \textsf{true} \mid \textsf{false}$
$\mid \langle\rangle \mid \langle V, V \rangle \mid \lambda x.\, e \; \cdots$

Garbage collection

Integers and pointers

$\boxed{\lambda_{\text{C}}}$ $w \in \mathit{Val} ::= (n \in \mathbb{Z}) \mid (a \in \mathit{Addr})$

Manual memory management

Key Challenge (as an OCaml hacker)

Write “glue code” using the OCaml FFI is **tricky and unsafe**.

mistake ⇒ *memory corruption* (often silent and hard to debug)

Which **rules** should **I** follow to safely use the OCaml FFI?

distinguished
OCaml hacker

Key Challenge (as an expert in program logics)

We already have powerful *program logics* for OCaml and C

but those are for programs written in a *single* language

How do we **formally reason** about such multi-language code?

program logics
expert

Key challenge (in this work)

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

λ_{ML} Semantics

Iris$_{\mathrm{ML}}$
Program Logic

given as black box

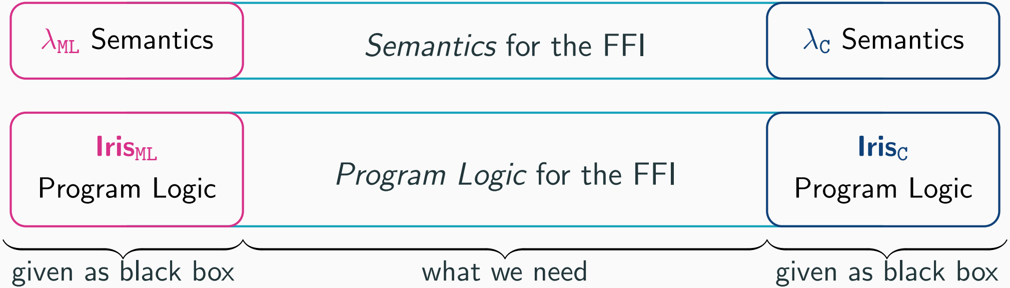
λ_{C} Semantics

Iris$_{\mathrm{C}}$
Program Logic

given as black box

Key challenge (in this work)

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



Design choice: **reuse** most of existing semantics/program logics;
do not drop down to a lowest-common denominator (assembly)!

Contributions

Melocoton:

- Two instantiations of Iris for a ML-like and C-like language with *external calls*
- An *operational semantics* for the OCaml FFI, bridging between the two languages.
- A *separation logic* for the OCaml FFI, bridging between the two language logics.
- A number of interesting *case studies*

Contributions

Melocoton:

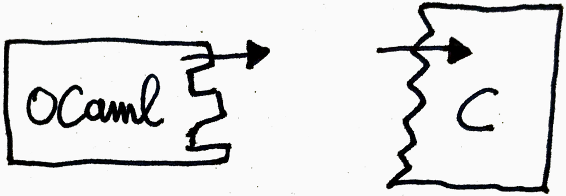
- Two instantiations of Iris for a ML-like and C-like language with *external calls*
- An *operational semantics* for the OCaml FFI, bridging between the two languages.
- A *separation logic* for the OCaml FFI, bridging between the two language logics.
- A number of interesting *case studies*

Language-locality: Verification of mixed OCaml/C programs can be done *almost entirely* in logics for OCaml and C!

In Iris: the logic is proved sound and all proofs are checked in Coq

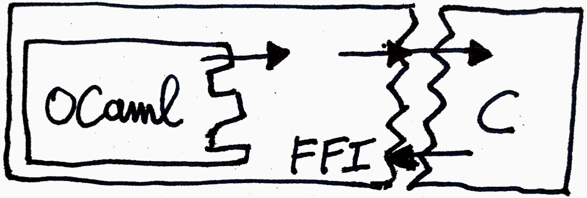
Outline

1. Language-local program logics with external calls



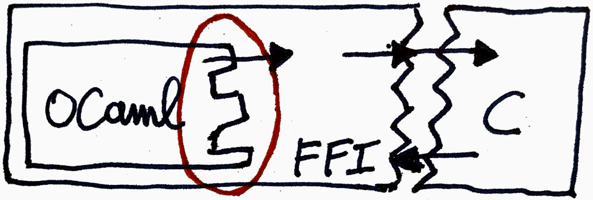
Outline

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



Outline

1. Language-local program logics with external calls
2. Program logic for FFI
3. Focus: the language boundary



Using the OCaml FFI: examples

The OCaml FFI deals with two core challenges:

- mediating between the different views of the OCaml memory
- interacting with the OCaml GC

Using the OCaml FFI: examples

The OCaml FFI deals with two core challenges:

- **mediating between the different views of the OCaml memory**
- interacting with the OCaml GC

Example: updating an OCaml reference from C code

OCaml code:

```ocaml
let main () =
  let r = ref 0 in
  update_ref r; (* TODO call C code and use rand() *)
  print_int !r
```

C code:

```c
int rand(int x) { ... }
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Glue code:

```
value caml_update_ref(value r) {
  /* TODO */
  int y = rand(x);
  /* TODO */
}
```

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

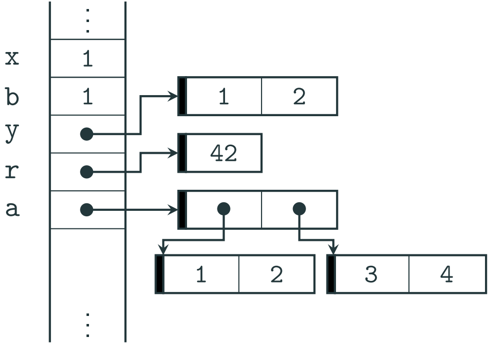
Glue code:

```
value caml_update_ref(value r) {
  /* TODO */
  int y = rand(x);
  /* TODO */
}
```

The runtime representation of OCaml values

At runtime, an OCaml value is either an integer or a pointer to a block:

```
let x = 1
let b = true
let y = (1, 2)
let r = ref 42
let a = [| (1, 2); (3, 4) |]
```



Glue code has access to this *low-level* representation of OCaml values.

Example: updating an OCaml reference from C code

OCaml code:

```
external update_ref : int ref -> unit = "caml_update_ref"
let main () =
  let r = ref 0 in
  update_ref r;
  print_int !r
```

C code:

```
int rand(int x) { ... }
```

Glue code:

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

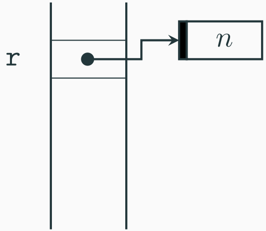
Glue code bridges between OCaml and C values by using powerful **FFI primitives**...

Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);                   /* get a random integer */
  Store_field(r, 0, Val_int(y));     /* store the value in the block */
  return Val_int(0);                 /* return () */
}
```

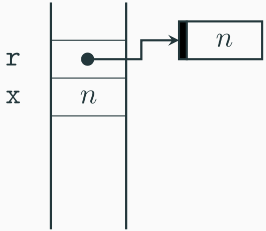
Writing glue code

```
value caml_update_ref(value r) {<--
  int x = Int_val(Field(r, 0));     /* read the first field of the input block */
  int y = rand(x);                  /* get a random integer */
  Store_field(r, 0, Val_int(y));    /* store the value in the block */
  return Val_int(0);                /* return () */
}
```



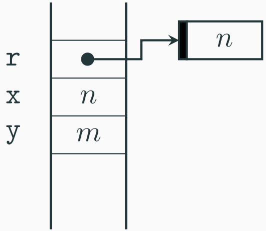
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0)); <-- /* read the first field of the input block */
  int y = rand(x);                  /* get a random integer */
  Store_field(r, 0, Val_int(y));    /* store the value in the block */
  return Val_int(0);                /* return () */
}
```



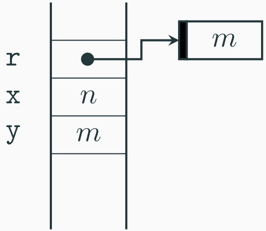
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);               <-- /* get a random integer */
  Store_field(r, 0, Val_int(y));     /* store the value in the block */
  return Val_int(0);                 /* return () */
}
```



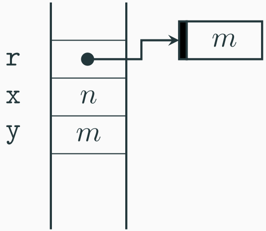
Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));      /* read the first field of the input block */
  int y = rand(x);                   /* get a random integer */
  Store_field(r, 0, Val_int(y));<-- /* store the value in the block */
  return Val_int(0);                 /* return () */
}
```



Writing glue code

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));     /* read the first field of the input block */
  int y = rand(x);                  /* get a random integer */
  Store_field(r, 0, Val_int(y));    /* store the value in the block */
  return Val_int(0);             <-- /* return () */
}
```



Using the OCaml FFI: examples

The OCaml FFI deals with two core challenges:

- mediating between the different views of the OCaml memory
- **interacting with the OCaml GC**

Example: swapping an OCaml pair

OCaml code:

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

Example: swapping an OCaml pair

OCaml code:

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

Glue code:
(first attempt)

```
value caml_swap_pair(value p)
{
  value r = caml_alloc(0, 2); /* allocate a block for the result */
  value x = Field(p, 0);      /* read the input pair */
  value y = Field(p, 1);
  Store_field(r, 0, y);       /* initialize the output pair */
  Store_field(r, 1, x);
  return r;                   /* return it */
}
```

Example: swapping an OCaml pair

OCaml code:

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

Glue code:

(first attempt)

```
value caml_swap_pair(value p)
{
  value r = caml_alloc(0, 2); /* allocate a block for the result */
  value x = Field(p, 0);      /* read the input pair */
  value y = Field(p, 1);
  Store_field(r, 0, y);       /* initialize the output pair */
  Store_field(r, 1, x);
  return r;                   /* return it */
}
```

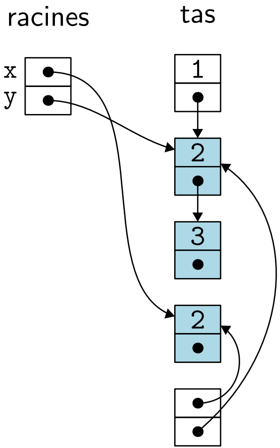
This implementation is unfortunately **incorrect** and will silently corrupt memory!

`caml_alloc` may run the GC which does not know about C variables and arguments...

OCaml has a “tracing” garbage collector.

Starts from *roots*; collects unreachable blocks; may also *move* blocks in memory.

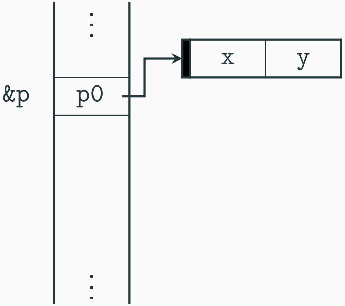
```
let x, y =
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...
```



Swapping pairs in the presence of a garbage collector

This implementation is unfortunately **incorrect!**

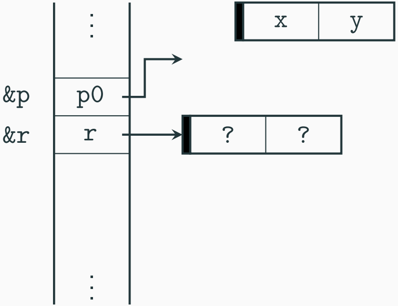
```
value caml_swap_pair(value p)
{                                   <--
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```



Swapping pairs in the presence of a garbage collector

This implementation is unfortunately **incorrect!**

```
value caml_swap_pair(value p)
{
  value r = caml_alloc(0, 2); <--
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

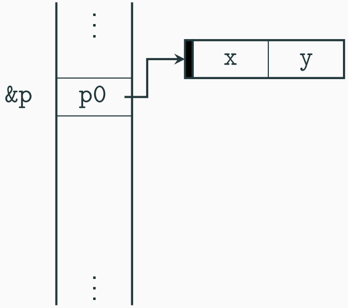


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{                                   <--
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

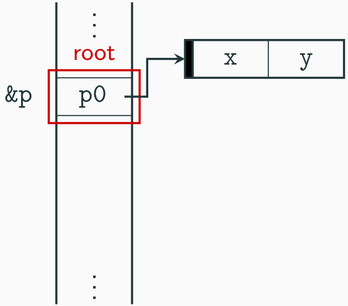


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);              <--
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

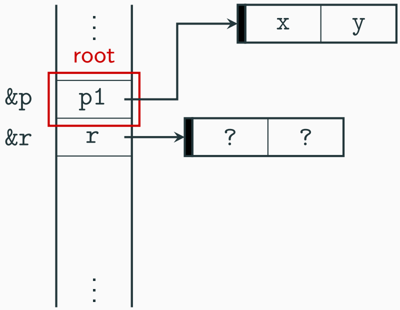


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);  <--
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

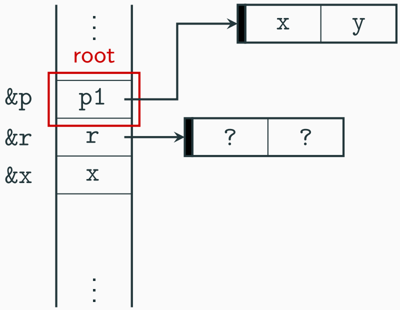


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);      <--
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

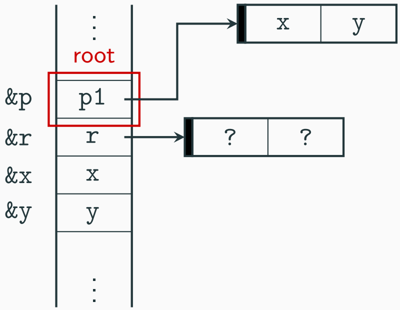


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);        <--
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

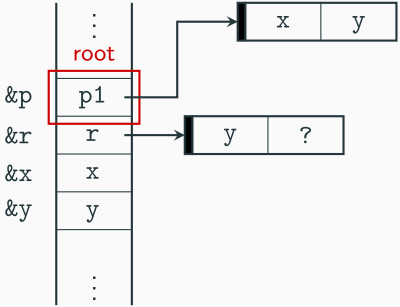


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);        <--
  Store_field(r, 1, x);
  return r;
}
```

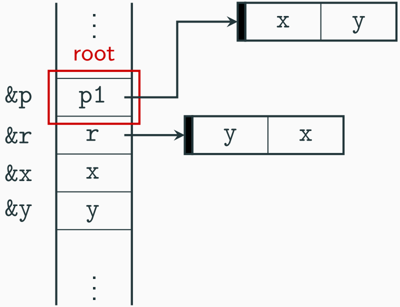


Registering roots

`CAMLparam1(p)` registers &p as a GC root.

The GC will avoid collecting the block, and will *update* p if the block moves.

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);        <--
  return r;
}
```



Unregistering roots

One subtle **bug** remains!

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

Unregistering roots

One subtle **bug** remains!

```c
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  return r;
}
```

The GC will *continue to update* &p after the function returns, corrupting the stack...!

We must use `CAMLreturn()` to unregister local roots when returning.

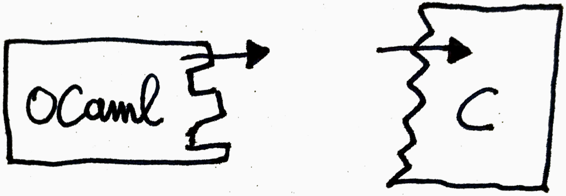
Our final implementation for swap_pair

```
external swap_pair : 'a * 'b -> 'b * 'a = "caml_swap_pair"
```

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```

Outline: Language-local reasoning

1. Language-local program logics with external calls



Language-local reasoning

We reuse:

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

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

Iris$_{\mathtt{ML}}$
Program Logic

Iris$_{\mathtt{C}}$
Program Logic

The one change: a minimal extension allowing **external calls**.

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

We model external calls as a new syntactic construct (inlining the declaration):

$$e \in \mathit{Expr} ::= \cdots \mid \mathsf{call}\ \mathit{fn}\ \vec{e}$$

Modeling External Calls

```
external update_ref : int ref -> unit = "caml_update_ref"

let main () :=
  let r = ref 0 in
  update_ref r;
  print_int !r
```

We model external calls as a new syntactic construct (inlining the declaration):

$$e \in \mathit{Expr} ::= \cdots \mid \mathsf{call}\ \mathit{fn}\ \vec{e}$$

We assign **no semantics** to external calls: they are simply stuck!

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \textsf{call}\ \texttt{caml\_update\_ref}\,[\ell]\,\{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \mathsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

"$\ell \mapsto_{\mathrm{ML}} V$" is a *Separation Logic assertion*

- asserts that the memory location ℓ stores the value V
- grants the **permission** to access the location (read/write)

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \mathsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

To do so, we introduce **interfaces** Ψ and Hoare triples $\{P\}\, e \,@\, \Psi \{v.\, Q\}$ that verify programs against them. For example, for `caml_update_ref`, we assume:

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n\rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\rangle \quad \sqsubseteq \Psi$$

Interface Specifications

We still want to *reason* about calls to `caml_update_ref`, as if it had the specification:

$$\forall \ell\, n.\ \{\ell \mapsto_{\mathrm{ML}} n\}\ \textsf{call}\ \texttt{caml\_update\_ref}\ [\ell]\ \{V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\}_{\mathrm{ML}}$$

To do so, we introduce **interfaces** Ψ and Hoare triples $\{P\}\, e \mathbin{@} \Psi \{v.\, Q\}$ that verify programs against them. For example, for `caml_update_ref`, we assume:

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n\rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\, \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m\rangle \quad \sqsubseteq \Psi$$

⚠ **This is an assumption, not a Hoare triple** ⚠

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

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

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

$$\Psi : \mathit{FnName} \to \vec{\mathit{Val}} \to (\mathit{Val} \to \mathit{iProp}) \to \quad \mathit{iProp}$$

"*iProp*" is the type of *Iris propositions*, which includes:

- quantifiers $\forall, \exists, \ldots$ and pure propositions
- Separation Logic modalities
- memory assertions *of both languages* ($\ell \mapsto_{\mathrm{ML}} V$, $a \mapsto_{\mathrm{C}} w$)
- specifications $\{P\}\, e \,@\, \Psi \,\{Q\}$ *of both languages*

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

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

We desugar

$$\forall \ell\, n.\ \langle \ell \mapsto_{\text{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]$$
$$\langle V'.\ \exists m.\ V' = \langle\rangle * \ell \mapsto_{\text{ML}} m \rangle$$

Desugaring To Predicate Transformers

Implement interface triples as a predicate transformer Ψ:

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

We desugar

$$\forall \ell\, n.\ \langle \ell \mapsto_{\mathrm{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]$$
$$\langle V'.\ \exists m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \rangle$$

as follows:

$$\Psi_{upd}\ fn\ \vec{V}\ \Phi := \exists \ell n.\ \ell \mapsto_{\mathrm{ML}} n * fn = \texttt{caml\_update\_ref} * \vec{V} = [\ell]$$
$$* (\forall V' m.\ V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \mathrel{-\!\!*} \Phi(V'))$$

Implementing Interface Triples

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

We parameterize Hoare triples by Ψ (inspired by de Vilhena and Pottier [2021]):

“$\{P\}\, e \,@\, \Psi\, \{Q\}$” means:

“Starting from a state satisfying P, e reduces to a value arriving in a state satisfying Q — either by normal reductions, *or by making external calls that satisfy* Ψ”

Implementing Interface Triples

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

We parameterize Hoare triples by Ψ (inspired by de Vilhena and Pottier [2021]):

"$\{P\}\, e \,@\, \Psi\, \{Q\}$" means:

"Starting from a state satisfying P, e reduces to a value arriving in a state satisfying Q — either by normal reductions, *or by making external calls that satisfy* Ψ"

Note: In a OCaml-and-C program (after linking), adequacy holds for $\Psi\ \mathit{fn}\ \vec{V}\ \Phi := \bot$

Implementing Interface Triples

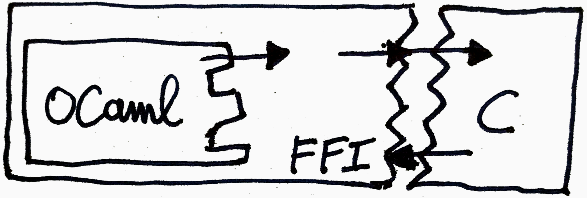
In **Iris**, we then **define** Hoare triples in terms of the operational semantics:

$$\{P\}\, e \,@\, \Psi\, \{Q\} \quad := \quad \Box\,(P \twoheadrightarrow \mathsf{wp}\; e \,@\, \Psi\, \{Q\})$$

$$\mathsf{wp}\; e \,@\, \Psi\, \{Q\} \quad := \quad \begin{cases} Q(v) & e = v \\ \forall e', (e \to e') \Rightarrow \mathsf{wp}\; e' \,@\, \Psi\, \{Q\} & e \text{ reducible} \\ \Psi\;\; \mathit{fn}\;\; \vec{V}\;\; \underbrace{\big(\lambda V'.\, \mathsf{wp}\; K[\,V'\,] \,@\, \Psi\, \{Q\}\big)}_{\text{Postcondition}} & e = K[\mathsf{call}\; \mathit{fn}\; \vec{V}] \end{cases}$$

Outline: The OCaml FFI

1. Language-local program logics with external calls
2. Glue code and program logic for FFI



External Calls in Glue Code

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

```
value caml_update_ref(value r) {
  int x = Int_val (Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

External Calls in Glue Code

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

```
value caml_update_ref(value r) {
  int x = Int_val (Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

$$\begin{array}{ll} \langle \mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \gamma \sim_{\mathrm{C}}^{\theta} w * v' \sim_{\mathrm{C}}^{\theta} w' \rangle & \\ \quad \mathsf{Store\_field}(w, i, w') & \sqsubseteq \Psi_{\mathrm{FFI}} \\ \langle \mathrm{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v}[i := v'] \rangle & \end{array}$$

External Calls in Glue Code

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

```
value caml_update_ref(value r) {
  int x = Int_val(Field(r, 0));
  int y = rand(x);
  Store_field(r, 0, Val_int(y));
  return Val_int(0);
}
```

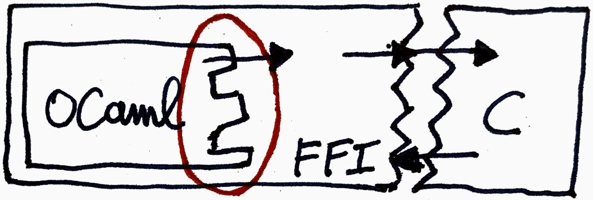
$$\{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \gamma \sim^{\theta}_{\mathrm{C}} w\}$$
$$\texttt{call caml\_update\_ref}\ [w]\ @\ \Psi_{\mathrm{FFI}}$$
$$\{w'.\, \exists m.\ \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m] * w' \sim^{\theta}_{\mathrm{C}} 0\}$$

Glue code is verified using the program logic for C, but additionally **assuming an interface** Ψ_{FFI} for the OCaml FFI primitives, with resources e.g. $\gamma \mapsto_{\mathrm{blk}[t|m]} \vec{v}$.

$$\begin{array}{l} \langle \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \gamma \sim^{\theta}_{\mathrm{C}} w * v' \sim^{\theta}_{\mathrm{C}} w' \rangle \\ \quad \mathsf{Store\_field}(w, i, w') \\ \langle \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v}[i := v'] \rangle \end{array} \sqsubseteq \Psi_{\mathrm{FFI}}$$

Outline: The OCaml-FFI boundary

1. Language-local program logics with external calls
2. Glue code and program logic for FFI
3. Focus: the OCaml-FFI boundary



View Reconciliation

We assumed an interface for `caml_update_ref` that uses ML points-tos:

$$\forall \ell n.\, \langle \ell \mapsto_{\mathrm{ML}} n \rangle\ \texttt{caml\_update\_ref}\ [\ell]\ \langle V'.\, \exists m.\, V' = \langle\rangle * \ell \mapsto_{\mathrm{ML}} m \rangle$$

Meanwhile, we proved the following specification for `caml_update_ref` using Ψ_{FFI}:

$$\begin{array}{l} \{\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [n] * \gamma \sim^{\theta}_{\mathrm{C}} w\} \\ \quad \texttt{call caml\_update\_ref}\ [w]\ @\ \Psi_{\mathrm{FFI}} \\ \{w'.\, \exists m.\, \mathsf{GC}(\theta) * w' \sim^{\theta}_{\mathrm{C}} 0 * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} [m]\} \end{array}$$

These express two different views about the **same piece of state**!

View Reconciliation: Update Rules

Idea:

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

$$\mathsf{GC}(\theta) * \ell \mapsto_{\mathrm{ML}} \vec{V} \equiv\!\!* \; \exists \vec{v}, \gamma.\, \mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \ell \sim_{\mathrm{ML}} \gamma * \vec{V} \sim_{\mathrm{ML}} \vec{v} \quad \text{(ML-TO-FFI)}$$

$$\mathsf{GC}(\theta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{mut}]} \vec{v} * \vec{V} \sim_{\mathrm{ML}} \vec{v} \equiv\!\!* \; \exists \ell.\, \mathsf{GC}(\theta) * \ell \mapsto_{\mathrm{ML}} \vec{V} * \ell \sim_{\mathrm{ML}} \gamma \quad \text{(FFI-TO-ML)}$$

View Reconciliation: Challenge

Challenge: **proving** that the view reconciliation rules are sound is hard!

View Reconciliation: Challenge

Challenge: **proving** that the view reconciliation rules are sound is hard!

The standard workflow in Iris:

- have Separation Logic memory assertions ($\ell \mapsto_{\mathrm{ML}} V$)
- have the state of the operational semantics (finite map: Location $\rightharpoonup$ Value)
- *relate the two* ("state interpretation"). *Often* straightforward...

View Reconciliation: Challenge

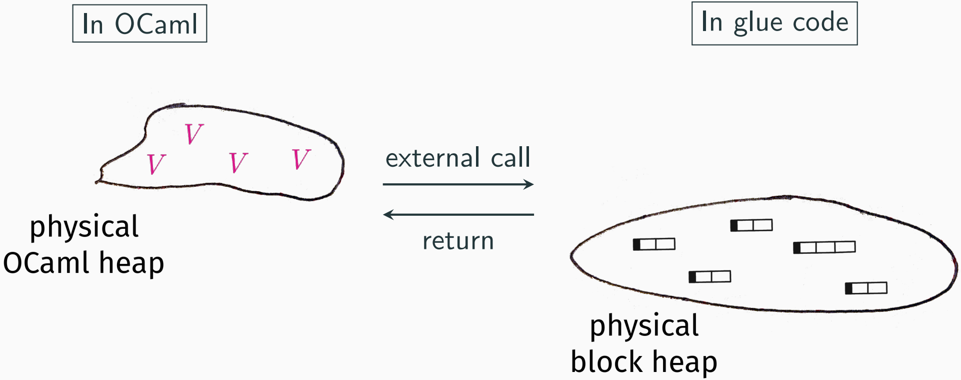
Challenge: **proving** that the view reconciliation rules are sound is hard!

- in the **program logic**, we can hold a mix of $\ell \mapsto_{\mathrm{ML}} \vec{V}$ and $\gamma \mapsto_{\mathrm{blk}[0|\textbf{mut}]} \vec{v}$
- the **operational semantics** has *only one simultaneous view* of the OCaml state

How can we relate the assertions and the operational semantics state?

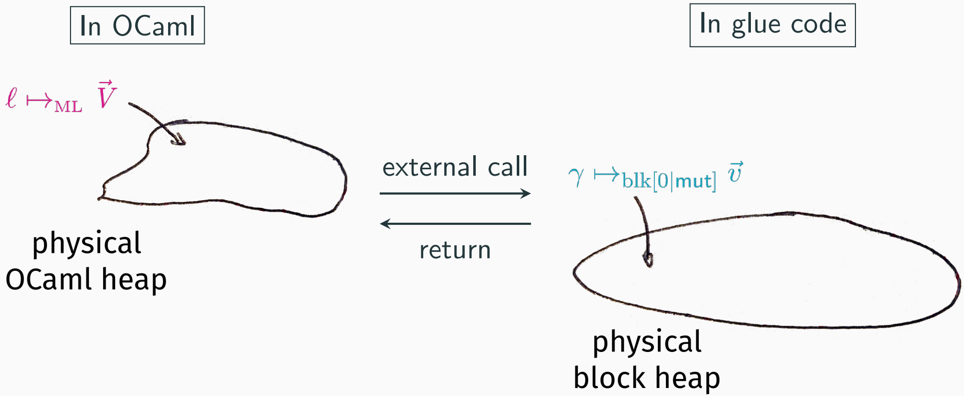
View Reconciliation: Challenge (2)

In the **operational semantics**, there is *only one simultaneous view* of the OCaml state.



View Reconciliation: Challenge (2) and Solution

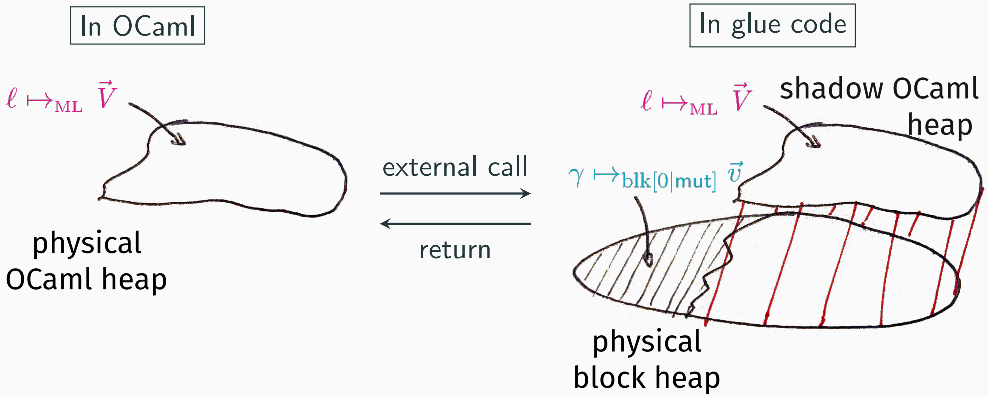
In the **program logic**: what happens to OCaml points-to?



View Reconciliation: Challenge (2) and Solution

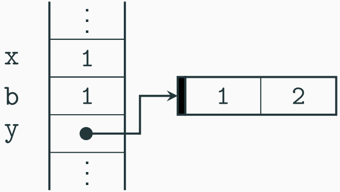
In the **program logic**: what happens to OCaml points-to?

Solution: track *both* views of the state in the program logic



Changing The Representation: Making Difficult Choices

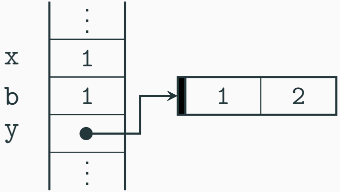
```
let x = ?
let b = ?
let y = (?, ?)
```



Quiz Time: What are the OCaml values of `x`, `b`, and `y`?

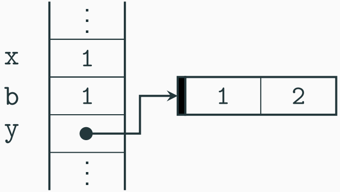
Changing The Representation: Making Difficult Choices

```
let x = 1
let b = true
let y = (1, 2)
```



Changing The Representation: Making Difficult Choices

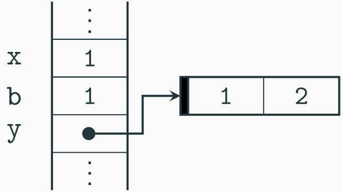
```
let x = 1
let b = true
let y = (1, 2)
```



High-level representation is **not unique**!

Changing The Representation: Making Difficult Choices

```
let x = 1
let b = true
let y = (1, 2)
```



High-level representation is **not unique**!

How does Operational Semantics choose the right value when switching to ML values?

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

$$\text{wp}\ e\ \{\Phi\} :\hat{=} \cdots \lor \big(e \text{ reducible} * \forall e'.\, e \to e' \mathrel{-\!\!*} \text{wp}\ e'\ \{\Phi\}\big) \qquad \text{usual Iris}$$

$$\text{wp}\ e\ \{\Phi\} :\hat{=} \cdots \lor \big(\exists X.\, e \twoheadrightarrow X * \forall e'.\, e' \in X \mathrel{-\!\!*} \text{wp}\ e'\ \{\Phi\}\big) \qquad \text{multi-relations}$$

Regular C and ML, not having angelic non-determinism, retain usual SOS

Angelic Non-Determinism And The Weakest Pre

We use angelic nondeterminism, based on multi-relations (see DimSum, CCR)!

$$\mathsf{wp}\ e\ \{\Phi\} :\hat{=} \cdots \vee \big(e\ \text{reducible} * \forall e'.\, e \to e' \twoheadrightarrow \mathsf{wp}\ e'\ \{\Phi\}\big) \qquad \text{usual Iris}$$

$$\mathsf{wp}\ e\ \{\Phi\} :\hat{=} \cdots \vee \big(\exists X.\, e \twoheadrightarrow X * \forall e'.\, e' \in X \twoheadrightarrow \mathsf{wp}\ e'\ \{\Phi\}\big) \qquad \text{multi-relations}$$

Regular C and ML, not having angelic non-determinism, retain usual SOS

For adequacy, existential needs to be extracted $\Rightarrow$ *transfinite Iris*

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

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

1. Abstract over "the other side" using **interfaces and external calls**
2. Formalize the **semantics of the FFI** (memory model and primitives)
3. Bridge between memory models using **view reconciliation**

Conclusion

Contribution: An Iris for toy C+ML+FFI, emphasizing **language-local reasoning**.

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

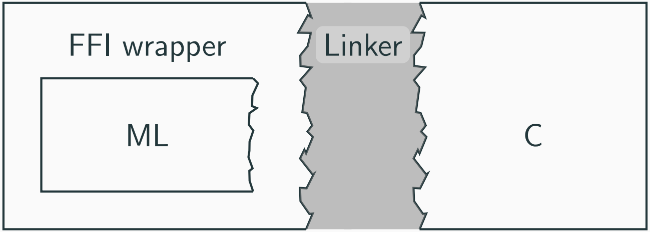
1. Abstract over "the other side" using **interfaces and external calls**
2. Formalize the **semantics of the FFI** (memory model and primitives)
3. Bridge between memory models using **view reconciliation**

More in the paper: `https://melocoton-project.github.io`

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

bonus slides

The semantics



The FFI wrapper

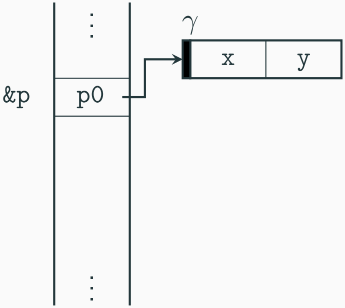
- Convert ML values to block-level
- Provide FFI: a C calling convention for ML

The Linker

- Link programs using the same calling convention
- Resolve external calls

Checking swap_pair

```
value caml_swap_pair(value p)
{                                          <--
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$GC(\alpha)$

$\gamma \mapsto_{\text{blk}[0|\text{imm}]} [x; y]$

$\&p \mapsto_C p0$

Facts:

$\text{blkaddr}(\alpha, \gamma) = p0$

Checking `swap_pair`

A *permission* describes the right to access some resources or memory:

$\boxed{\mathsf{GC}(\alpha)}$: permission to use C functions of the FFI

$\rightarrow$ α: an abstract name that identifies **a specific layout** of the GC memory.

(α changes when the GC moves or deallocates block)

$\boxed{\gamma \mapsto_{\text{blk}[0|\text{imm}]} [x; y; \ldots]}$: permission to access a block in the GC memory

$\rightarrow$ γ: abstract **label** of the block

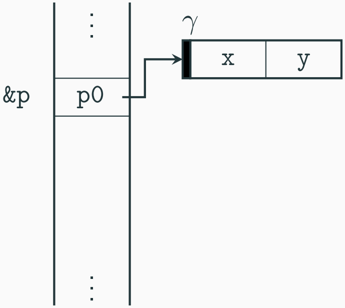
$\rightarrow$ [x; y; ...]: contents of the block

$\boxed{\&p \mapsto_{C} \mathrm{p0}}$: permission to access the C variable p

$\rightarrow$ p0: current value of the variable

Checking swap_pair

```
value caml_swap_pair(value p)
{                                   <--
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\alpha)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathrm{x}; \mathrm{y}]$

$\&p \mapsto_{\mathrm{C}} \mathtt{p0}$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

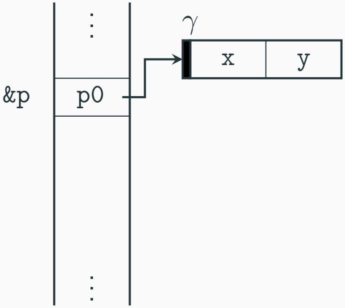
We also collect "facts" (mathematical equalities) of the form:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

means: the block with label γ has concrete address p0, *when the GC memory is in layout* α

Checking swap_pair

```
value caml_swap_pair(value p)
{                                    <--
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$GC(\alpha)$

$\gamma \mapsto_{blk[0|imm]} [x; y]$

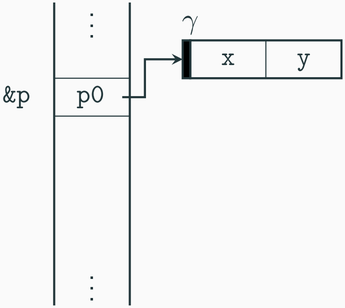
$\&p \mapsto_{C} p0$

Facts:

$blkaddr(\alpha, \gamma) = p0$

Checking swap_pair

```
value caml_swap_pair(value p)
{                                  <--
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\alpha)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathrm{x}; \mathrm{y}]$

$\&p \mapsto_{\mathrm{C}} \mathtt{p0}$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

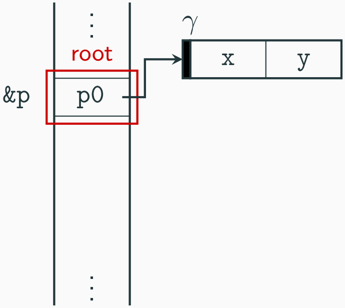
$$\langle \mathsf{GC}(\alpha) * \&p \mapsto_{\mathrm{C}} \mathtt{p0} * \mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0} \rangle$$

$$\mathsf{CAMLparam1}(p)$$

$$\langle \mathsf{GC}(\alpha) * \&p \mapsto_{\mathrm{root}} \gamma \rangle$$

Checking swap_pair

```c
value caml_swap_pair(value p)
{
  CAMLparam1(p);                        <--
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\alpha)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathrm{x}; \mathrm{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

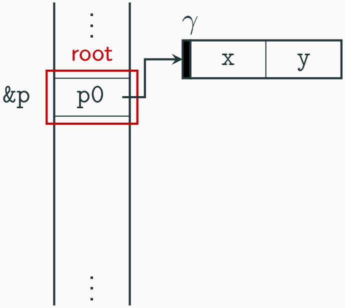
Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathrm{p0}$

$$\langle \mathsf{GC}(\alpha) * \&p \mapsto_{\mathrm{C}} \mathrm{p0} * \mathrm{blkaddr}(\alpha, \gamma) = \mathrm{p0} \rangle$$
$$\mathsf{CAMLparam1}(p)$$
$$\langle \mathsf{GC}(\alpha) * \&p \mapsto_{\mathrm{root}} \gamma \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);                <--
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\alpha)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathrm{x}; \mathrm{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathrm{p0}$

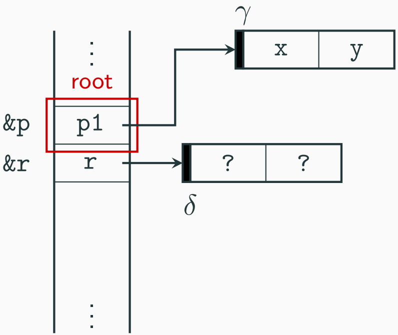
$$\langle \mathsf{GC}(\alpha) \rangle$$

$$\mathsf{caml\_alloc}(0, n)$$

$$\langle \mathtt{r}.\ \exists \beta.\ \mathrm{blkaddr}(\beta, \delta) = \mathtt{r} * \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [?; \ldots; ?] \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);<--
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [?; ?]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

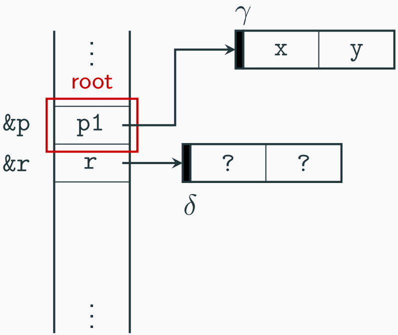
$$\langle \mathsf{GC}(\alpha) \rangle$$

$$\quad \mathsf{caml\_alloc}(0, n)$$

$$\langle \mathtt{r}.\ \exists \beta.\ \mathrm{blkaddr}(\beta, \delta) = \mathtt{r} * \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [?; \ldots; ?] \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);<--
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [?; ?]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

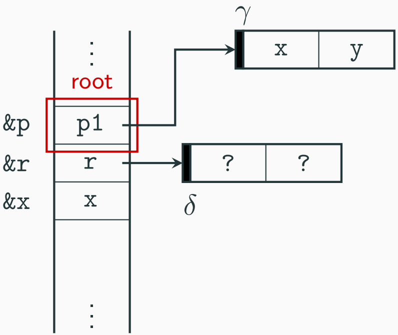
$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

Rule for reading a root &p

$$\langle \mathsf{GC}(\beta) * \&p \mapsto_{\mathrm{root}} \gamma \rangle \quad \mathtt{*(\&p)}$$
$$\langle \mathtt{p1}.\ \mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1} * \mathsf{GC}(\beta), \&p \mapsto_{\mathrm{root}} \gamma \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);     <--
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathsf{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathsf{imm}]} [?; ?]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

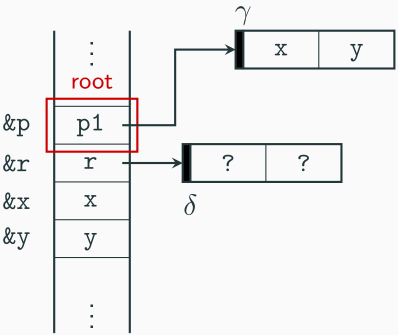
$$\langle \mathsf{GC}(\beta) * \gamma \mapsto_{\mathrm{blk}[0|\mathsf{imm}]} [..; v_i; \ldots] * \mathrm{blkaddr}(\beta, \gamma) = \mathtt{p} \rangle$$

$$\mathsf{Field}(\mathtt{p}, i)$$

$$\langle v_i.\ \mathsf{GC}(\beta), \gamma \mapsto_{\mathrm{blk}[0|\mathsf{imm}]} [..; v_i; \ldots] \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);      <--
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [?; ?]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

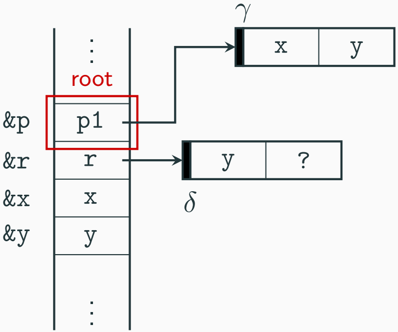
$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

$$\langle \mathsf{GC}(\beta) * \gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v_i; \ldots] * \mathrm{blkaddr}(\beta, \gamma) = \mathtt{p} \rangle$$
$$\mathsf{Field}(\mathtt{p}, i)$$
$$\langle v_i.\ \mathsf{GC}(\beta), \gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v_i; \ldots] \rangle$$

Checking `swap_pair`

```c
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);      <--
  Store_field(r, 1, x);
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{y}; ?]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

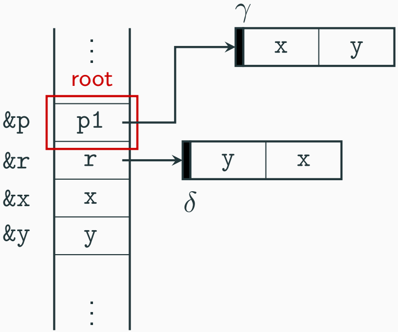
$$\langle \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v_i; \ldots] * \mathrm{blkaddr}(\beta, \delta) = \mathtt{r} \rangle$$

$$\mathsf{Store\_field}(\mathtt{r}, i, v)$$

$$\langle \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v; \ldots] \rangle$$

Checking `swap_pair`

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);    <--
  CAMLreturn(r);
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{root}} \gamma$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{y}; \mathtt{x}]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

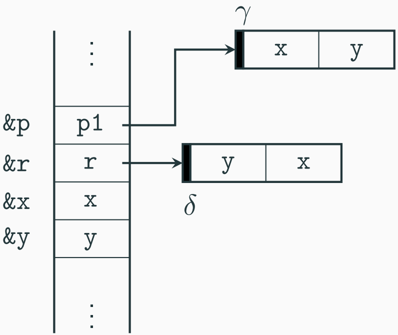
$$\langle \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v_i; \ldots] * \mathrm{blkaddr}(\beta, \delta) = \mathtt{r} \rangle$$

$$\mathsf{Store\_field}(\mathtt{r}, i, v)$$

$$\langle \mathsf{GC}(\beta) * \delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [..; v; \ldots] \rangle$$

Checking swap_pair

```
value caml_swap_pair(value p)
{
  CAMLparam1(p);
  value r = caml_alloc(0, 2);
  value x = Field(p, 0);
  value y = Field(p, 1);
  Store_field(r, 0, y);
  Store_field(r, 1, x);
  CAMLreturn(r);                <--
}
```



Permissions:

$\mathsf{GC}(\beta)$

$\gamma \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{x}; \mathtt{y}]$

$\&p \mapsto_{\mathrm{C}} \mathtt{p1}$

$\delta \mapsto_{\mathrm{blk}[0|\mathrm{imm}]} [\mathtt{y}; \mathtt{x}]$

Facts:

$\mathrm{blkaddr}(\alpha, \gamma) = \mathtt{p0}$

$\mathrm{blkaddr}(\beta, \delta) = \mathtt{r}$

$\mathrm{blkaddr}(\beta, \gamma) = \mathtt{p1}$

$$\langle \mathsf{GC}(\beta) * \&p \mapsto_{\mathrm{root}} \gamma \rangle$$

$$\mathsf{CAMLreturn}(\mathtt{r})$$

$$\langle \mathsf{GC}(\beta) * \&p \mapsto_{\mathrm{C}} \mathrm{blkaddr}(\beta, \gamma) \rangle$$