Melocoton: Formal Reasoning Rules for the OCaml FFI

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

May 11, 2023

FFI?

FFI: “Foreign Function Interface”

In this project, we look at the FFI OCaml↔C.

The mechanism used in OCaml to call to external code using the C ABI.

Useful to write “bindings” to C libraries, to call system primitives, etc.

Requires writing “glue code” to bridge between OCaml and C

Motivation

Writing code using the OCaml FFI is tricky and **very unsafe**.
The programmer needs to obey many FFI-specific rules *documented informally*.

Some existing approaches make this task **safer**: Ctypes, ocaml-rust.
Encode *some* of the FFI rules in the type system of OCaml or Rust.

Melocoton: formalize rules that make using the OCaml FFI **completely safe**.
These rules are formulated using an expressive program logic (based on Coq+Iris).
Future work: a tool that checks FFI code using these rules!

FFI example: calling system APIs

in `unix.ml` from the OCaml standard library:

```
external openfile :
  string -> open_flag list -> file_perm -> file_descr
  = "caml_unix_open"

external close : file_descr -> unit = "caml_unix_close"

type stats = { st_perm : file_perm; st_uid : int; st_size : int; (* ... *) }
external fstat : file_descr -> stats = "caml_unix_fstat"
```

external functions receive and return OCaml values

they can be called from OCaml code like ordinary functions

Anatomy of C code using the OCaml FFI

```c
#include <caml/mlvalues.h>
/* ... */
value caml_unix_open(value path, value flags, value perm)
{
  CAMLparam3(path, flags, perm);
  int fd, cv_flags, clo_flags, cloexec;
  char * p;
  /* ... */
  p = caml_stat_strdup(String_val(path));
  caml_enter_blocking_section();
  fd = open(p, cv_flags, Int_val(perm));
  caml_leave_blocking_section();
  caml_stat_free(p);
  /* ... */
  CAMLreturn (Val_int(fd));
}
```

Anatomy of C code using the OCaml FFI

```c
#include <caml/mlvalues.h>
/* ... */
value caml_unix_open(value path, value flags, value perm)
{
  CAMLparam3(path, flags, perm);
  int fd, cv_flags, clo_flags, cloexec;
  char * p;
  /* ... */
  p = caml_stat_strdup(String_val(path));
  caml_enter_blocking_section();
  fd = open(p, cv_flags, Int_val(perm));
  caml_leave_blocking_section();
  caml_stat_free(p);
  /* ... */
  CAMLreturn (Val_int(fd));
}
```

system call

Anatomy of C code using the OCaml FFI

```c
#include <caml/mlvalues.h>
/* ... */
value caml_unix_open(value path, value flags, value perm)
{
  CAMLparam3(path, flags, perm);
  int fd, cv_flags, clo_flags, cloexec;
  char * p;
  /* ... */
  p = caml_stat_strdup(String_val(path));
  caml_enter_blocking_section();
  fd = open(p, cv_flags, Int_val(perm));
  caml_leave_blocking_section();
  caml_stat_free(p);
  /* ... */
  CAMLreturn (Val_int(fd));
}
```

ocaml values

system call

Anatomy of C code using the OCaml FFI

```
#include <caml/mlvalues.h>
/* ... */
value caml_unix_open(value path, value flags, value perm)
{
  CAMLparam3(path, flags, perm);
  int fd, cv_flags, clo_flags, cloexec;
  char * p;
  /* ... */
  p = caml_stat_strdup(String_val(path));
  caml_enter_blocking_section();
  fd = open(p, cv_flags, Int_val(perm));
  caml_leave_blocking_section();
  caml_stat_free(p);
  /* ... */
  CAMLreturn (Val_int(fd));
}
```

ocaml values

glue code

system call

glue code

Anatomy of C code using the OCaml FFI

```
#include <caml/mlvalues.h>
/* ... */
value caml_unix_open(value path, value flags, value perm)
{
  CAMLparam3(path, flags, perm);
  int fd, cv_flags, clo_flags, cloexec;
  char * p;
  /* ... */
  p = caml_stat_strdup(String_val(path));
  caml_enter_blocking_section();
  fd = open(p, cv_flags, Int_val(perm));
  caml_leave_blocking_section();
  caml_stat_free(p);
  /* ... */
  CAMLreturn (Val_int(fd));
}
```

ocaml FFI types and functions

ocaml values

glue code

system call

glue code

Challenges

Writing C "glue code" that uses the OCaml FFI is tricky:

- operates on a low-level representation of OCaml values;
- needs to account for the action of the OCaml garbage collector;
- embedding C data in OCaml values is subject to extra restrictions;
- reentrancy issues when using callbacks

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

Challenges

Writing C "glue code" that uses the OCaml FFI is tricky:

- operates on a low-level representation of OCaml values;
- needs to account for the action of the OCaml garbage collector;
- embedding C data in OCaml values is subject to extra restrictions;
- reentrancy issues when using callbacks

(in this talk)

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

The plan for today

Let's consider a *minimal example*, and:

1. show how to write glue code and what the *possible pitfalls* are;
2. give a set of *formal rules* that guarantee a *bug-free* use of the OCaml FFI;
3. use these rules to *check the safety* of our example glue code.

A minimal example

As an illustrative example, let's consider a simple `swap_pair` function.

In OCaml: `let swap_pair (x, y) = (y, x)`

Goal: implement an equivalent function in C using the OCaml FFI:

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

Sounds easy!

Swapping pairs

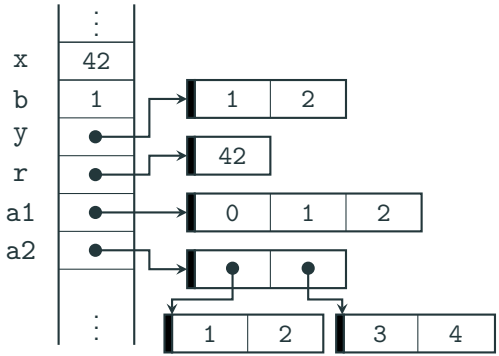
```
value caml_swap_pair(value p)
{
  /* TODO */


}
```

`value` is the C type for OCaml values. What does it represent in memory?

Swapping pairs

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

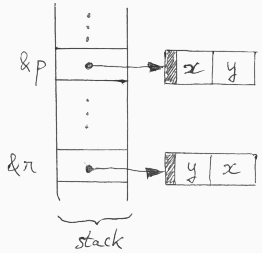
```
let x = 42
let b = true
let y = (1, 2)
let r = ref 42
let a1 = [| 0; 1; 2 |]
let a2 =
  [| (1, 2); (3, 4) |]
```



(integers and pointers are distinguished using their least significant bit)

Swapping pairs

```
value caml_swap_pair(value p)
{
  /* TODO */

  return r;
}
```

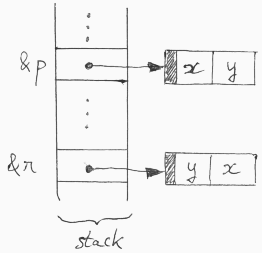


`value` is the C type for OCaml values. What does it represent in memory?

Answer: a `value` holds the low-level runtime representation of an OCaml value.

Swapping pairs (2)

```
value caml_swap_pair(value p)
{
  /* TODO */

  return r;
}
```



So, how do we inspect or construct a `value` in C code?

Swapping pairs (2)

C functions of the OCaml FFI (provided by `<caml/mlvalues.h>` & co):

- `Field(v, i)`, `Store_field(v1, i, v2)` : read and write to a block-shaped value
- `caml_alloc(tag, sz)` : allocate a new block of size sz and tag tag
- `Int_val(v)`, `Val_int(i)` : convert between C integers and OCaml integer values
- and more... (operations on other kinds of blocks, etc.)

Swapping pairs (2)

A 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 components of 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 */
}
```

Swapping pairs, in the presence of a garbage collector (3)

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

```
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;
}
```

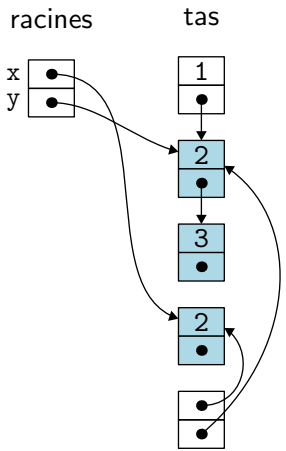
Reason: interaction with the OCaml garbage collector

`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.

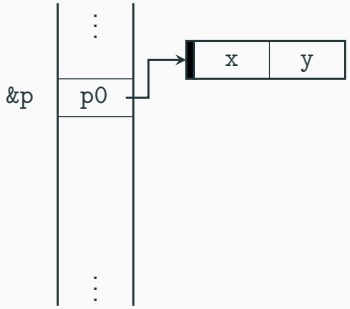
```
let x, y =
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...
```



Swapping pairs, in the presence of a garbage collector (3)

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

```
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;
}
```



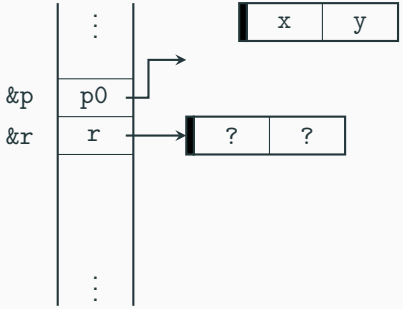
Reason: interaction with the OCaml garbage collector

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

Swapping pairs, in the presence of a garbage collector (3)

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

```
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;
}
```



Reason: interaction with the OCaml garbage collector

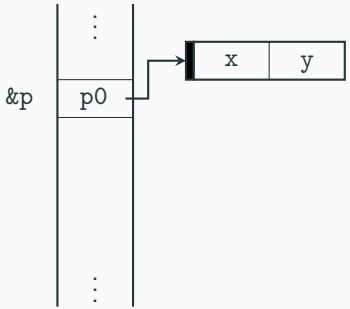
`caml_alloc` may run the GC, which does not know about C variables and arguments!

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;
}
```

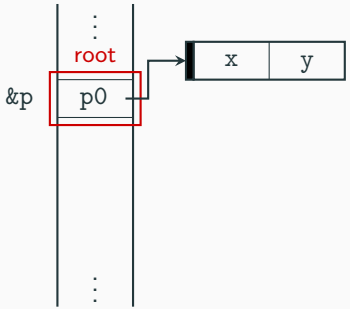


Registering roots

`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;
}
```

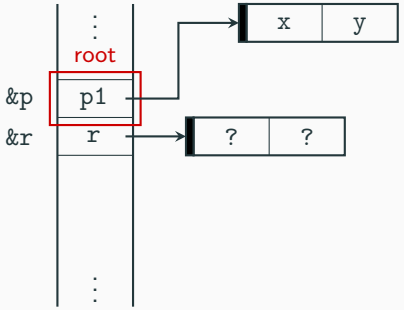


Registering roots

`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;
}
```

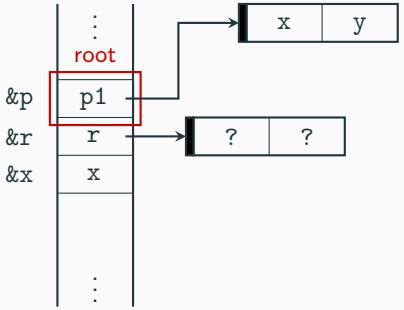


Registering roots

`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;
}
```

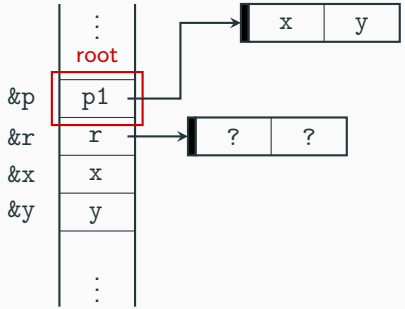


Registering roots

`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;
}
```

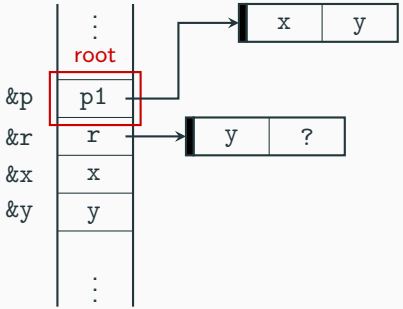


Registering roots

`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;
}
```

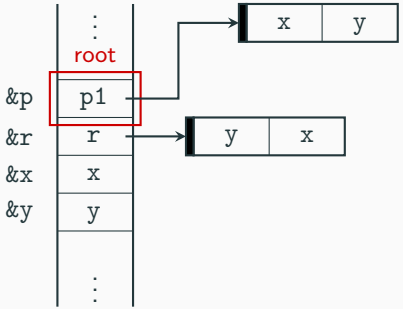


Registering roots

`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!

```
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);
}
```

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);
}
```

...how do we know that it is really correct this time?

Formally understanding the OCaml FFI

How do we know if we have obeyed all the OCaml FFI rules?

→ the OCaml manual (Chapter 22) provides informal guidelines

→ **This work:** *formal* reasoning rules for the OCaml FFI (based on Separation Logic) that capture *all* the constraints of the FFI

Currently: – rules that cover a subset of the available functions in OCaml 4.14
(including callbacks and restricted custom blocks!)
– formal model in Coq+Iris

End goal: – rules that cover all functions of the OCaml FFI
– a tool that automatically checks that C "glue code" follows the rules

Reasoning with permissions

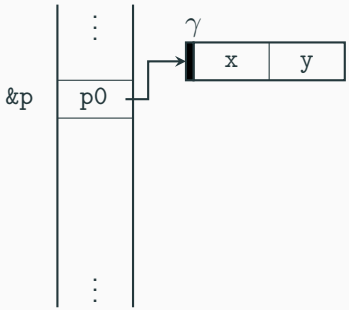
(insiders: "permission" = Separation Logic assertion)

Checking code using our formal rules means:

- Analyze glue code by tracking available "permissions" at each point of the code; (without running the code! think: like an advanced form of typechecking)
- Functions of the FFI require some permissions and can produce new permissions;
- If the required permissions are available at each point, then the program is *correct* (no segfault, no memory corruption)

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<`α`>`

γ `@ blk [x; y]`

`&p @ local(p0)`

Facts:

`blkaddr<`α`>(`γ`)` $=$ `p0`

Checking `swap_pair`

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

`GC<`α`>`: 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)

γ `@ blk [x; y; ...]`: permission to access a block in the GC memory

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

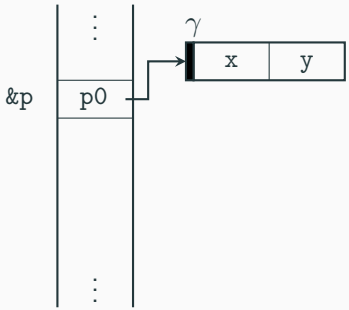
$\rightarrow$ `[x; y; ...]`: contents of the block

`&p @ local(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:

`GC<`α`>`

γ `@ blk [x; y]`

`&p @ local(p0)`

Facts:

`blkaddr<`α`>(`γ`)` $=$ `p0`

Checking `swap_pair`

```
value caml_swap_pair(value p)
{                                <--
  CAMLp
  value
  value
  value
  Store
  Store
  CAMLr
}
```

Permissions:

GC<α>

0)

= p0

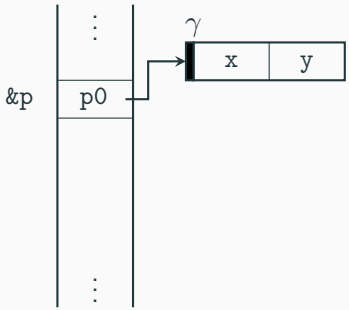
We also collect "facts" (mathematical equalities) of the form:

blkaddr<α>(γ) = 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<`α`>`

γ `@ blk [x; y]`

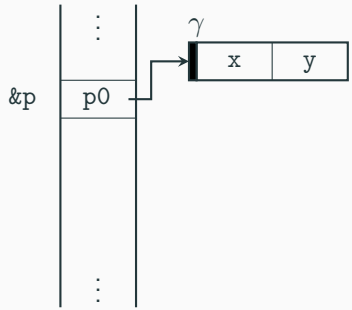
`&p @ local(p0)`

Facts:

`blkaddr<`α`>(`γ`)` $=$ `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:

GC<α>

γ @ blk [x; y]

&p @ local(p0)

Facts:

blkaddr<α>(γ) = p0

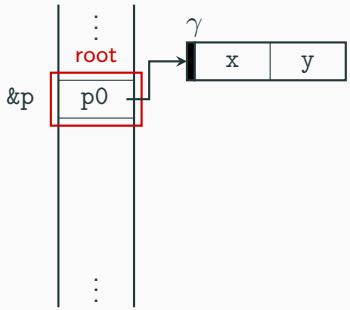
Rule for `CAMLparam1(p)`

consumes: GC<α>, &p @ local(p0), if blkaddr<α>(γ) = p0

produces: GC<α>, &p @ root(γ)

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<α>

γ @ blk [x; y]

&p @ root(γ)

Facts:

blkaddr<α>(γ) = p0

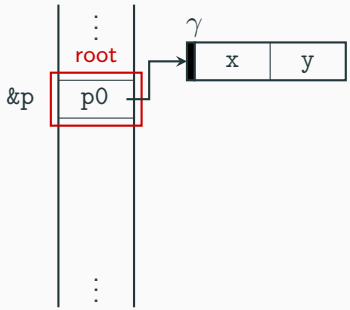
Rule for `CAMLparam1(p)`

consumes: GC<α>, &p @ local(p0), if blkaddr<α>(γ) = p0

produces: GC<α>, &p @ root(γ)

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<α>

γ @ blk [x; y]

&p @ root(γ)

Facts:

blkaddr<α>(γ) = p0

Rule for `caml_alloc(0, n)`

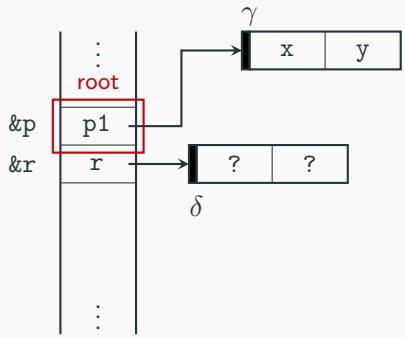
consumes: GC<α>

produces: GC<β>, δ @ blk [?;...;?]

returns: blkaddr<β>(δ)

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [?; ?]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

Rule for `caml_alloc(0, n)`

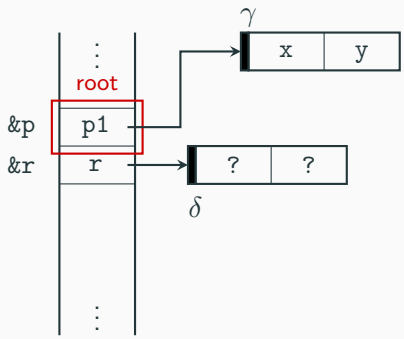
consumes: GC<α>

produces: GC<β>, δ @ blk [?;...;?]

returns: blkaddr<β>(δ)

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [?; ?]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

blkaddr<β>(γ) = p1

Rule for reading a root `p`

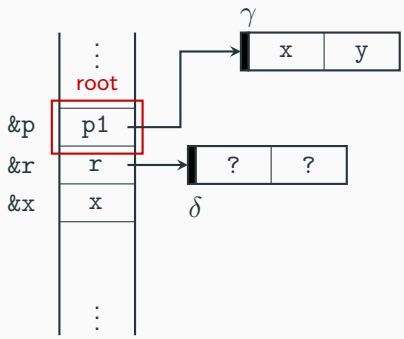
consumes: GC<β>, &p @ root(γ)

produces: GC<β>, &p @ root(γ)

returns: blkaddr<β>(γ)

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [?; ?]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

blkaddr<β>(γ) = p1

Rule for `Field(p, i)`

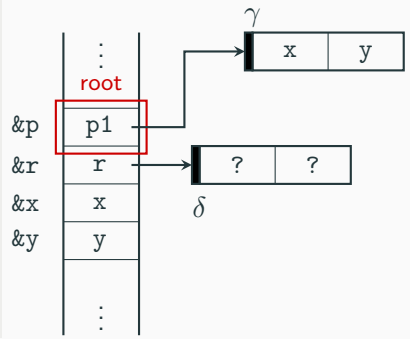
consumes: GC<β>, γ @ blk [..; vi; ...], if blkaddr<β>(γ) = p

produces: GC<β>, γ @ blk [..; vi; ...]

returns: vi

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [?; ?]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

blkaddr<β>(γ) = p1

Rule for `Field(p, i)`

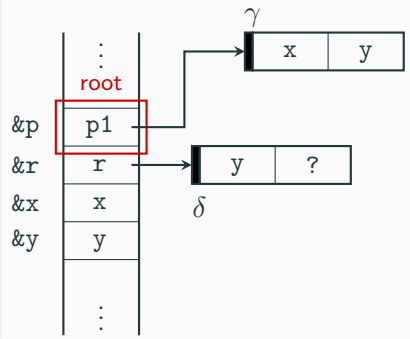
consumes: GC<β>, γ @ blk [..; vi; ...], if blkaddr<β>(γ) = p

produces: GC<β>, γ @ blk [..; vi; ...]

returns: vi

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [y; ?]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

blkaddr<β>(γ) = p1

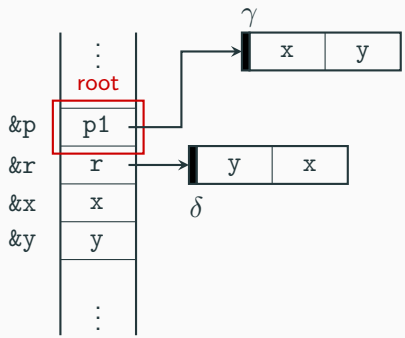
Rule for `Store_field(r, i, v)`

consumes: GC<β>, δ @ blk [..; vi; ...], if blkaddr<β>(δ) = r

produces: GC<β>, δ @ blk [..; v; ...]

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<β>

γ @ blk [x; y]

&p @ root(γ)

δ @ blk [y; x]

Facts:

blkaddr<α>(γ) = p0

blkaddr<β>(δ) = r

blkaddr<β>(γ) = p1

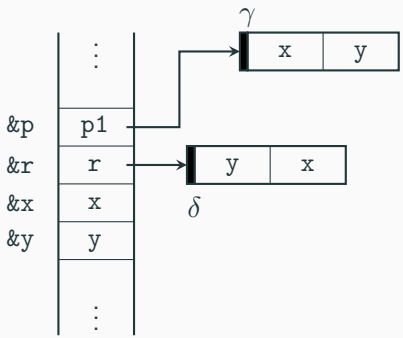
Rule for `Store_field(r, i, v)`

consumes: GC<β>, δ @ blk [..; vi; ...], if blkaddr<β>(δ) = r

produces: GC<β>, δ @ blk [..; v; ...]

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<`β`>`

γ `@ blk [x; y]`

`&p @ local(p1)`

δ `@ blk [y; x]`

Facts:

`blkaddr<`α`>(`γ`) = p0`

`blkaddr<`β`>(`δ`) = r`

`blkaddr<`β`>(`γ`) = p1`

Rule for `CAMLreturn(r)`

consumes: `GC<`β`>`, `&p @ root(`γ`)`

produces: `GC<`β`>`, `&p @ local(blkaddr<`β`>(`γ`))`

Beyond `swap_pair`

- We saw permission-based rules for checking the use of some FFI functions
- *Theorem*: successfully checking the rules $\Rightarrow$ program is safe

Beyond this simple example, we also provide:

- rules to translate between **high-level OCaml values** and low-level block permissions
- an interpretation of (a subset of) **OCaml types** in terms of permissions
- rules for "advanced" FFI constructs: **callbacks**, **custom blocks**

Sample: reentrancy and callbacks

Things become tricky in presence of callbacks, but our logic is quite expressive.

For instance we can prove in Coq that the following implements a fixed-point combinator:

```
let knot (f: ('a -> 'b) -> ('a -> 'b)) =
  let l = ref (fun _ -> assert false) in
  l := (fun x -> f (fun y -> callk l y) x);
  (fun x -> callk l x)
```

```
value callk(value l, value x) {
  value f = Field(l, 0);
  return caml_callback(f, x);
}
```

Melocoton: a logical foundation for verified FFIs

Our rules and permissions are defined in a *well-studied framework*: Separation Logic.
In principle very expressive.

So far, mainly focused on designing this logic for FFI in a nice and **modular** way,
which interoperates smoothly with existing semantics/Separation Logics for OCaml/C

Now that the foundations are done, we can move to the fun stuff!

Future plans & ideas

- more rules covering more of the OCaml FFI API
- an automated tool to check if C code satisfies the FFI rules
- a DSL to write FFI "glue code" with built-in checking of the rules
- using verification tools (Why3, Creusot, Viper?) to enable fully-safe Ctypes/ocaml-rust FFI

Happy to discuss internships / collaborations / etc!

(research paper under submission, ask me if you want a link!)