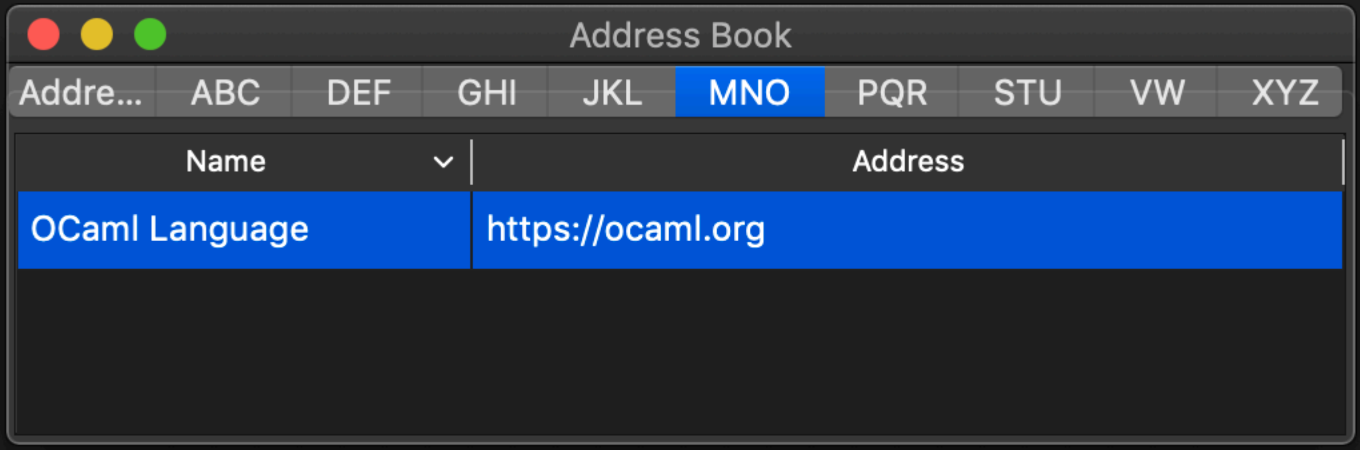
A safer FFI for OCaml?

CAMLroot

PLAN

- Context: OCaml & Qt
- The OCaml FFI
- 1st contribution: roots
- 2nd contribution: regions
- Conclusion

OCaml & Qt


Address Book
Addre...
ABC
DEF
GHI
JKL
MNO
PQR
STU
VW
XYZ
Name
Address
OCaml Language
https://ocaml.org

OCaml

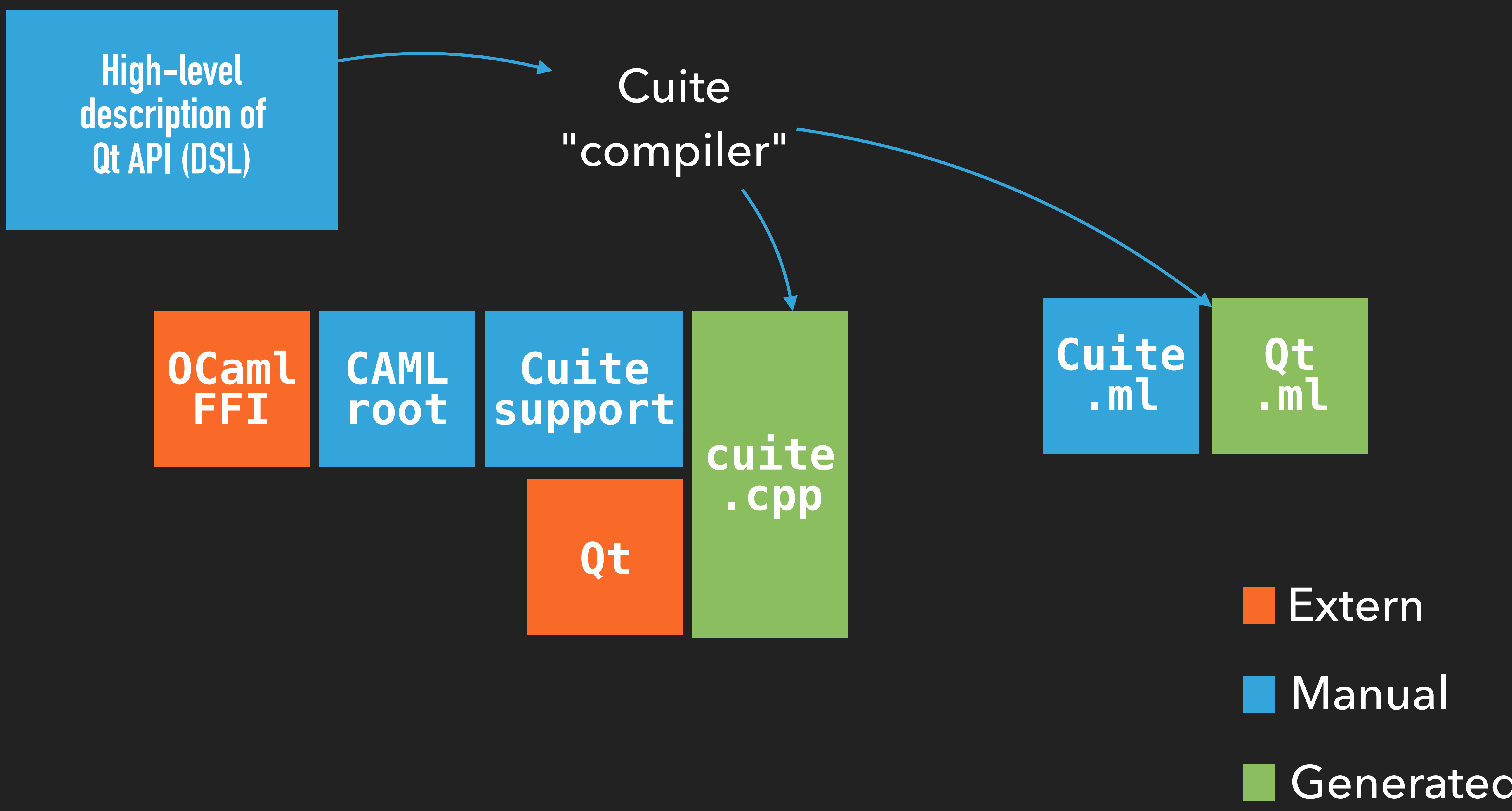
- Functional programming language
- Automatic memory management with a GC

Qt (Gui/Widgets)

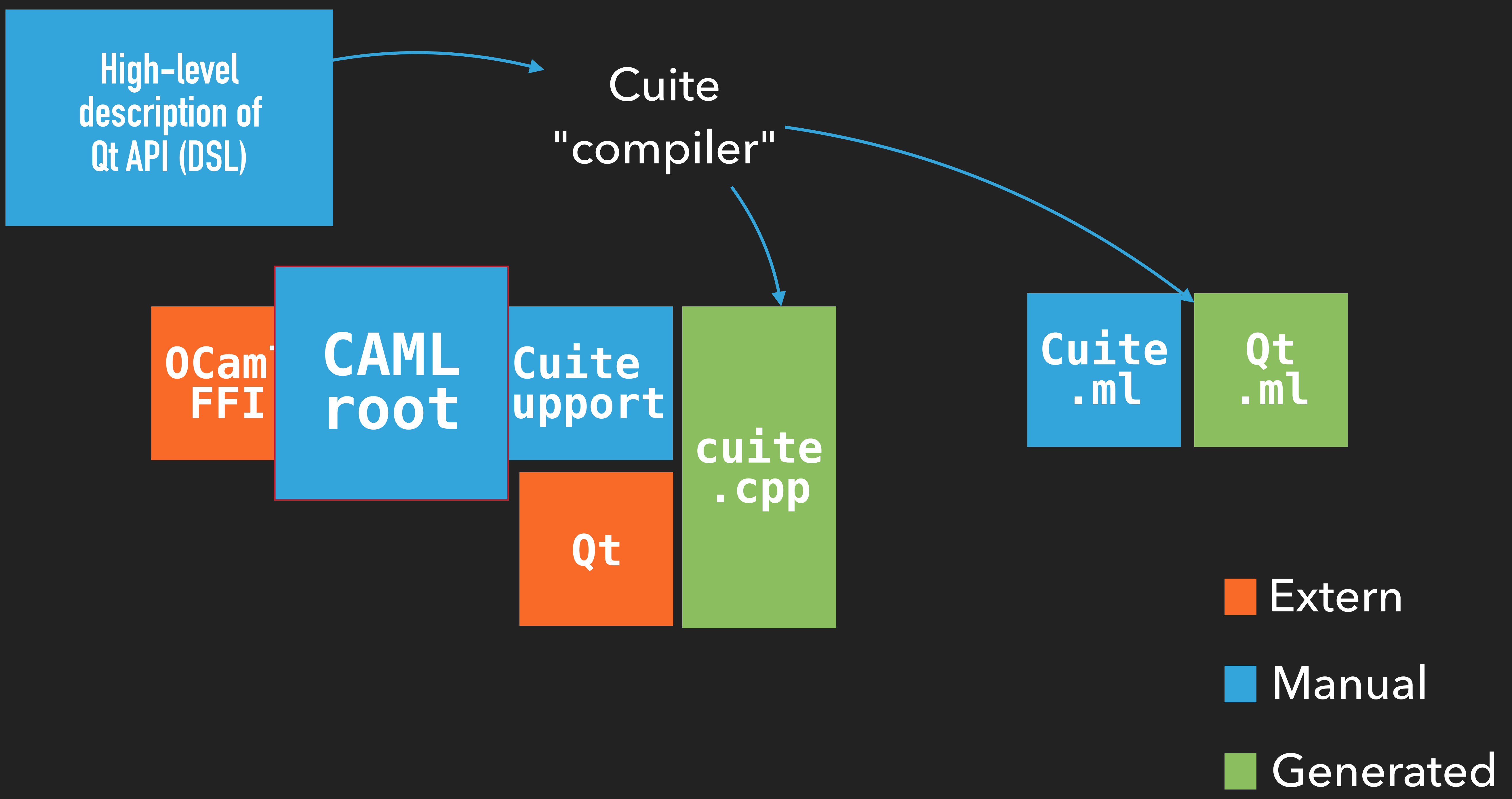
- C++ framework (OOP)
- Dynamic graph of objects
- Complex lifetimes
- Higher-order control flow
- Concurrency
- Very large API surface (thousands of methods), multiple versions

```
[def@miso ~]$ utop
```

CUITE ARCHITECTURE



CUITE ARCHITECTURE



The OCaml FFI

- 👍 Efficient (Sundials/ML)
- 👍 Expressive ((de)constructing values, higher-order control flow, exception management, ...)
- 👎 Low-level, hard to use properly
- 👎 Risky (heap corruption, "heisenbug")

VALUE REPRESENTATION

Two concepts:

- the words
- the blocks

A WORD

An integer as wide as a pointer (32 or 64 bits depending on the platform).

- if the least significant bit is 0,
  the word is interpreted as the address of a block
- if the least significant bit is 1,
  the remaining bits (31 or 63) are interpreted as a signed integer (the word is then said to be "tagged integer")

```
typedef intptr_t value;
```

A BLOCK

A block is a chunk of memory managed by the GC. Its size, expressed as a number of words, is stored in a header that precedes the block.
The header also determines whether the block should be scanned or not:

- if yes, the **block** is made of **words** (that are themselves interpreted as immediate integers or as block addresses)
- if not, the content is opaque to the GC; it does not affect the graph of OCaml heap

ALLOCATION, CONSTRUCTION & DE-CONSTRUCTION OF VALUES

For immediate values:

```
value Val_long(long int);
long int Long_val(value);

value Val_bool(bool);
bool Bool_val(value);
```

Allocation, construction & de-construction of Values

For blocks:

```
value caml_alloc(mlsize_t, tag_t);
```

```
value Field(value, int);
void Store_field(value, int, value);
```

GC INTEGRATION

Periodically, the GC has to:

- traverse the heap to determine the set of live values;
  the GC has to know the roots manipulated from C side
- copy and compact the blocks, and update the addresses;
  the GC needs to be able to change the values of C roots

EXAMPLE

```
let mk_pair x y = (x, y)

CAMLprim value c_mk_pair(value x, value y)
{
  CAMLparam2(x, y);
  CAMLlocal1(result);
  result = caml_alloc(2, 0);
  Store_field(result, 0, x);
  Store_field(result, 1, y);
  CAMLreturn(result);
}
```

EXAMPLE

```
let mk_pair x y = (x, y)

CAMLprim value c_mk_pair(value x, value y)
{
  CAMLparam2(x, y);
  CAMLlocal1(result);
  result = caml_alloc(2, 0);
  Store_field(result, 0, x);
  Store_field(result, 1, y);
  CAMLreturn(result);
}
```

Add &x et &y to
the set of roots.

Example

```
let mk_pair x y = (x, y)

CAMLprim value c_mk_pair(value x, value y)
{
  CAMLparam2(x, y);
  CAMLlocal1(result);
  result = caml_alloc(2, 0);
  Store_field(result, 0, x);
  Store_field(result, 1, y);
  CAMLreturn(result);
}
```

Declare a variable result and add &result to the set of roots.

EXAMPLE

```
let mk_pair x y = (x, y)

CAMLprim value c_mk_pair(value x, value y)
{
  CAMLparam2(x, y);
  CAMLlocal1(result);
  result = caml_alloc(2, 0);
  Store_field(result, 0, x);
  Store_field(result, 1, y);
  CAMLreturn(result);
}
```

The allocation can trigger the GC.
x and y could be updated.

EXAMPLE

```
let mk_pair x y = (x, y)

CAMLprim value c_mk_pair(value x, value y)
{
  CAMLparam2(x, y);
  CAMLlocal1(result);
  result = caml_alloc(2, 0);
  Store_field(result, 0, x);
  Store_field(result, 1, y);
  CAMLreturn(result);
}
```

Remove &x, &y and &result from the set of roots. Return result.

BAD EXAMPLE (1/2)

```
let mk_quad x y z w = ((x, y), (z, w))

CAMLprim
value c_mk_quad(value x, value y,
                value z, value w)
{
  CAMLparam4(x, y, z, w);
  CAMLlocal1(result);
  result = c_mk_pair(c_mk_pair(x, y),
                     c_mk_pair(z, w));
  CAMLreturn(result);
}
```

BAD EXAMPLE (1/2)

```
let mk_quad x y z w = ((x, y), (z, w))

CAMLprim
value c_mk_quad(value x, value y,
                value z, value w)
{
  CAMLparam4(x, y, z, w);
  CAMLlocal1(result);
  result = c_mk_pair(c_mk_pair(x, y),
                     c_mk_pair(z, w));
  CAMLreturn(result);
}
```

- The result of the first call is not stored in a root.
- If the second call trigger a GC, the heap will get corrupted.

Bad example (2/2)

```
let mk_triplet x y z = (x, (y, z))

CAMLprim
value c_mk_triplet(value x, value y, value z)
{
  CAMLparam3(x, y, z);
  CAMLlocal1(result);
  result = c_mk_pair(x, c_mk_pair(y, z));
  CAMLreturn(result);
}
```

BAD EXAMPLE (2/2)

```
let mk_triplet x y z = (x, (y, z))

CAMLprim
value c_mk_triplet(value x, value y, value z)
{
  CAMLparam3(x, y, z);
  CAMLlocal1(result);
  result = c_mk_pair(x, c_mk_pair(y, z));
  CAMLreturn(result);
}
```

- x is read
- c_mk_pair can trigger the GC, which might update x.

This is an undefined behavior.

OUR GOALS

- Detect unregistered roots
- Prevent undefined behaviours due to GC interaction
- Simplify management of roots

1ST CONTRIBUTION: A ROOT-CENTRIC API

OCaml "value"s do not behave like real values from the C point of view:

- their address is captured by the GC,
- their value can change between each call, if the GC got triggered.

Passing an argument ("x") is not simply copying a value but results in a *memory load* that sample the actual value at the time of the call.

Passing roots as argument (1/2)

The problem comes from the implicit dereferencing that happens when passing an argument.

- In Rust, Caml-oxide shows that its type system is fine enough to capture this subtlety.
- In C, `CAMLroot` replaces arguments of type "`value`" by arguments of type "`value*`" (that represent roots).

PASSING ROOTS AS ARGUMENT (2/2)

Return values are replaced by an extra argument; another root in which the result will be stored.

```
void mlroot_alloc(value *, mlsize_t, tag_t);
void mlroot_set_field(value *, int, value *);
```

- return type is `void`: it is unlikely that the developer will continue to nest calls.

- arguments are now pointers, there is no risk of mixing both styles:
  `mlroot_set_field(result, 0, &caml_alloc(2, 0));`

Administrative Normal Form (ANF)

```
void mlroot_mk_pair(value *out,
                    value *x, value *y);

CAMLprim
value c_mk_triplet(value x, value y, value z)
{
  CAMLparam3(x, y, z);
  CAMLlocal2(result, tmp);
  mlroot_mk_pair(&tmp, &y, &z);
  mlroot_mk_pair(&result, &x, &tmp);
  CAMLreturn(result);
}
```

DEFENSIVE PROGRAMMING

- The arguments are now pointers.
  The implicit dereferencing is now explicit but will happen in the primitive functions of the FFI and not in user code.
- These pointers are roots, which should be registered to the GC by the time they reached primitive functions.
- A optional "defensive" mode checks that a root has actually been registered before each dereferencing.

WHAT DOES THIS INDIRECTION BUY US?

👍 No more undefined behavior:

- thanks to the ANF-style
- thanks to the use of adresses (which are stable) and not values

👍 An opt-in defensive mode that detects wrong use as early as possible.

👎 A slight increase in verbosity.

- A similar performance profile

2ND CONTRIBUTION: REGION-BASED ALLOCATION OF ROOTS

The root-centric API got rid of incorrect value manipulation.

We still have to take of roots. Can we simplify this too?

Idea: introducing "regions", an array of roots that permits dynamic allocation of roots.

THE REGIONS

When switching from OCaml to C, a region is set up:

```
CAMLprim value c_mk_pair(value x, value y)
{
  CAMLregion(&x, &y);
  value *result = mlregion_alloc(2, 0);
  mlroot_set_field(result, 0, &x);
  mlroot_set_field(result, 1, &y);
  CAMLregion_return(*result);
}
```

THE REGIONS

When switching from OCaml to C, a region is set up:

```
CAMLprim value c_mk_pair(value x, value y)
{
  CAMLregion(&x, &y);
  value *result = mlregion_alloc(2, 0);
  mlroot_set_field(result, 0, &x);
  mlroot_set_field(result, 1, &y);
  CAMLregion_return(*result);
}
```

Region is initialised with the addresses &x and &y

THE REGIONS

When switching from OCaml to C, a region is set up:

```
CAMLprim value c_mk_pair(value x, value y)
{
  CAMLregion(&x, &y);
  value *result = mlregion_alloc(2, 0);
  mlroot_set_field(result, 0, &x);
  mlroot_set_field(result, 1, &y);
  CAMLregion_return(*result);
}
```

A fresh root is returned

THE REGIONS

When switching from OCaml to C, a region is set up:

```
CAMLprim value c_mk_pair(value x, value y)
{
  CAMLregion(&x, &y);
  value *result = mlregion_alloc(2, 0);
  mlroot_set_field(result, 0, &x);
  mlroot_set_field(result, 1, &y);
  CAMLregion_return(*result);
}
```

When leaving the region,
&x, &y and result are dropped.

THE REGIONS

When switching from OCaml to C, a region is set up.

- Populated with arguments
- Dynamically extended when allocating new local roots
- Released when returning to OCaml

THE REGIONS

The dynamic allocation provided by region let us recover and safe and direct style:

```
value *mlregion_pair(value *x, value *y);

CAMLprim
value c_mk_triplet(value x, value y, value z)
{
  CAMLregion(&x, &y, &z);
  value *result =
    mlregion_pair(&x, mlregion_pair(&y, &z));
  CAMLregion_return(*result);
}
```

VARIATIONS (1/2) : SUB-REGIONS

All roots are released when leaving C code.
If a function needs a lot of roots (for instance, because they are allocated in a loop), this introduces a memory leak.

For this we provide **sub-regions**.

These enable a local management roots:

```
typedef region_t;
void mlregion_subenter(region_t *region);
void mlregion_subleave(region_t *region);
```

VARIATIONS (1/2) : REGIONS WITHOUT OCAML

OCaml only allow a single thread to access the runtime at any given time. A lock is used to control the access to runtime state from multiple threads.

A different kind of region lift the management of this lock to the region API:

```
void mlregion_release_runtime_system(void);
void mlregion_acquire_runtime_system(void);
```

This API is also defensive: inside such region, calls to FFI primitives will fail (`mlregion_alloc`, `mlregion_get_field`, …).

IN C++

- References (&) allow to hide the distinction between values and pointers, reducing the syntactic noise
- "Resource Acquisition Is Initialization" idiom (RAII) allow to tie region management to lexical scope.

```
CAMLprim value cpp_external(value f)
{
  CAMLregion r(&f);
  ...
  return result;
}
```

CONCLUSION

Two changes to simplify the OCaml FFI:

- a root-centric API that covers the same cases as normal OCaml FFI while increasing safety
- a region system that can simplify code, especially in C++, but that is not as expressive as normal FFI
- both are in development on https://github.com/let-def/cuite and https://github.com/let-def/camlroot

Related Work

- **O'Saffire**
  A verifier for manual bindings. Not maintained anymore.
- **Ctypes**
  An EDSL for generating bindings.
- **Caml-oxide**
  A proof-of-concept FFI in Rust that uses the type system to enforce GC invariants.

Benchmark

Name	Time/Run	mWd/Run	Percentage
mk_pair_caml	12.62ns	3.00w	57.74%
mk_pair_caml_slow	15.88ns	3.00w	72.66%
mk_pair_root	21.86ns	3.00w	100.00%
mk_pair_root_safe	25.84ns	3.00w	118.20%

mk_pair_caml: FFI OCaml (macros)
mk_pair_caml_slow: FFI OCaml (functions)
mk_pair_root: FFI CAMLroot (checks disabled)
mk_pair_root_safe: FFI CAMLroot (checks enabled)

REFERENCES

- Sundials/ML
  T. Bourke, J. Inoue, M. Pouzet
  https://hal.inria.fr/hal-01408230

- Ctypes
  J. Yallop, A. Madhavapeddy, D. Sheets
  https://github.com/ocamllabs/ocaml-ctypes

- Caml-oxide
  S. Dolan
  https://github.com/stedolan/caml-oxide