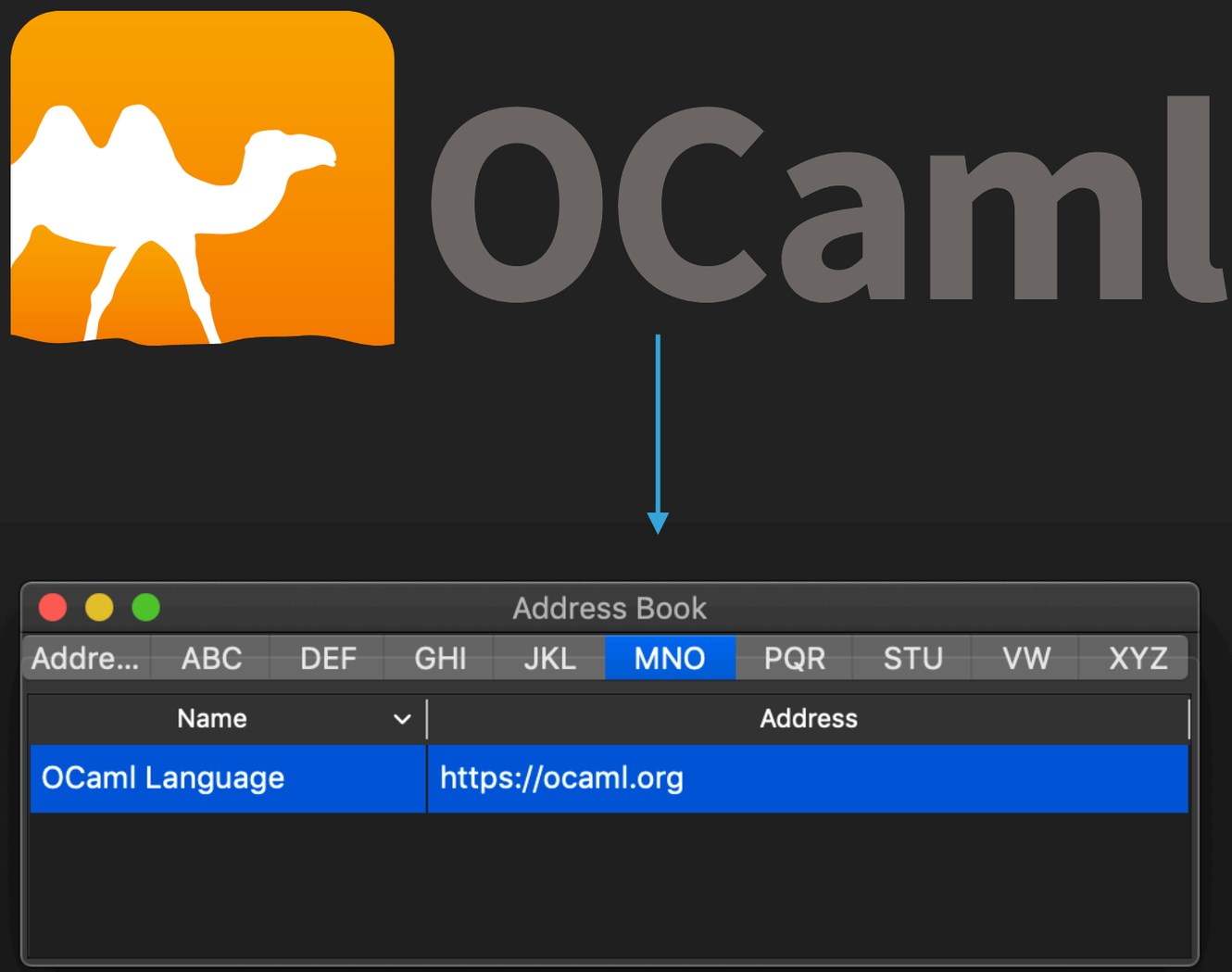
Une "FFI" plus sûre pour OCaml ?

CAMLroot

PLAN

- Contexte : OCaml & Qt
- La "FFI" OCaml
- Contribution 1 : Les racines
- Contribution 2 : Les régions
- Conclusion

OCaml & Qt


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

OCaml

- Programmation fonctionnelle
- Gestion automatique de la mémoire (GC)

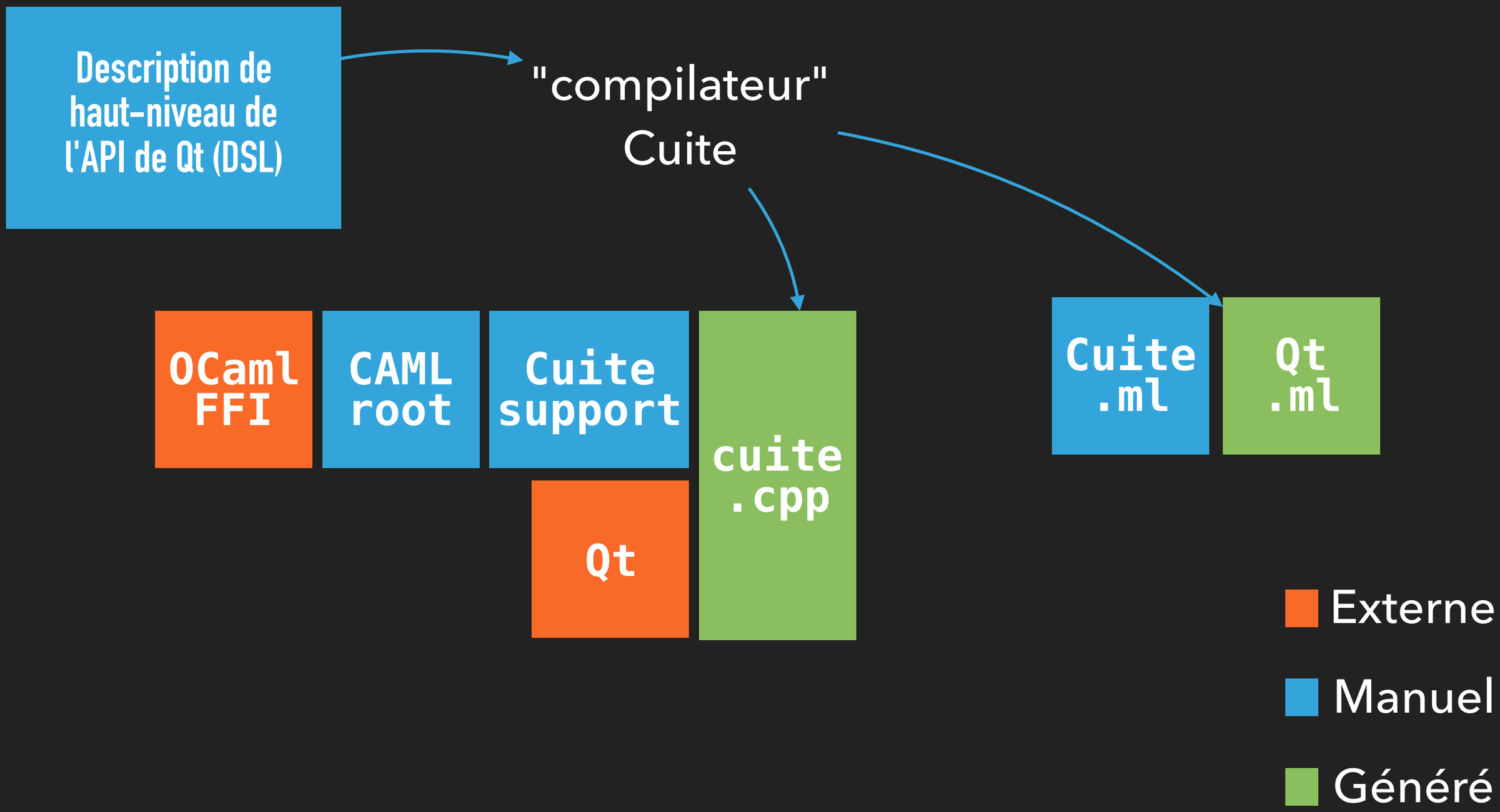
Qt (Gui/Widgets)

- Framework C++ (OOP)
- Graphe d'objets dynamique
- Durées de vie complexes
- Ordre supérieur
- Concurrence
- API très large (milliers de méthodes)

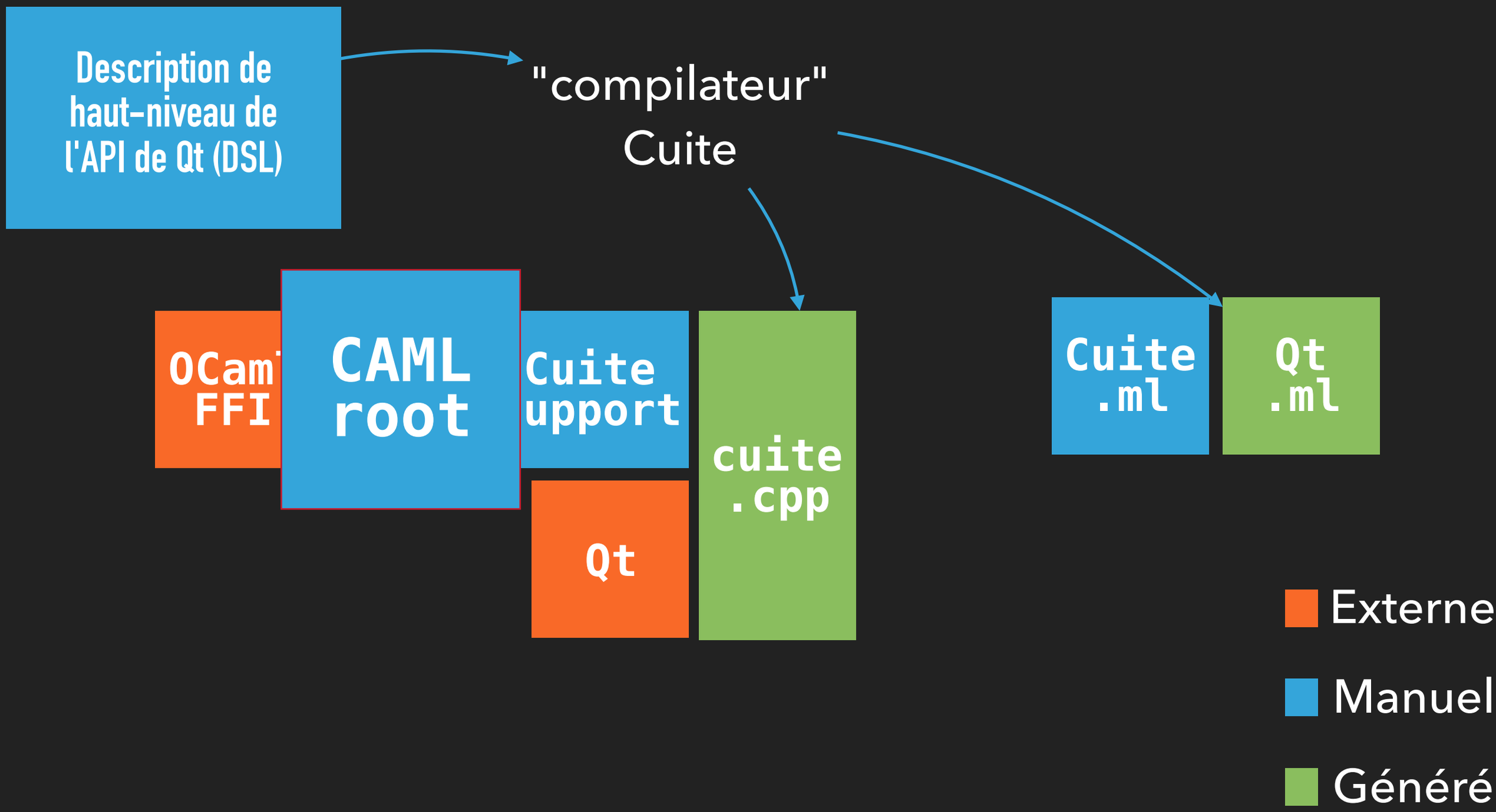
OCaml & Qt

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

ARCHITECTURE DE CUITE



ARCHITECTURE DE CUITE



La FFI OCaml

- 👍 Performante (Sundials/ML)
- 👍 Expressive ((dé)construction de valeurs, flot de contrôle d'ordre supérieur, gestion des exceptions, ...)
- 👎 Bas-niveau, difficile à utiliser correctement
- 👎 Risquée (corruption du tas, "heisenbug")

Représentation des Valeurs

Deux concepts :

- les mots
- les blocs

Un mot

Un entier de la taille d'un pointeur (32 ou 64 bits selon la plateforme).

- si le bit de poids faible est 0,
  il est interprété comme l'adresse d'un bloc
- si le bit de poids faible est 1,
  les bits restants (31 ou 63) sont interprétés comme un entier signé, dit "tagged integer"

```
typedef intptr_t value;
```

Un bloc

Un bloc est une zone mémoire gérée par le GC. Sa taille, en nombre de mots, est stockée dans une entête qui précède le bloc.
L'entête détermine également si le contenu du bloc est scanné ou pas :

- si oui, le **bloc** est composé de **mots (**eux-même interprétés comme des entiers ou des adresses de blocs)
- si non, le contenu est opaque pour le GC; il n'affecte pas la graphe du tas OCaml

Allocation, construction et déconstruction des Valeurs

Pour les valeurs immédiates :

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

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

Allocation, construction et déconstruction des Valeurs

Pour les blocs :

```
value caml_alloc(mlsize_t, tag_t);
value Field(value, int);
void Store_field(value, int, value);
```

Intégration au GC

Périodiquement, le GC doit :

- traverser le tas pour déterminer les valeurs en vie; le GC a besoin de connaître les racines côté C
- copier et compacter les blocs, et mettre à jour leurs adresses; le GC doit pouvoir mettre à jour une valeur du code C

Exemple

```
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);
}
```

Exemple

```
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);
}
```

Ajoute &x et &y à l'ensemble des racines.

Exemple

```
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);
}
```

Déclare une variable result et ajoute &result aux racines.

Exemple

```
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);
}
```

L'allocation peut déclencher le GC.
x et y pourraient être mis à jour.

Exemple

```
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);
}
```

Enlève &x, &y et &result des racines.
Retourne result.

Mauvais exemple (1/2)

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

```
CAMLprim
value c_mk_triplet(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);
}
```

Mauvais exemple (1/2)

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

```
CAMLprim
value c_mk_triplet(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);
}
```

- Le résultat du premier appel n'est pas enregistré comme racine.
- Si le second appel déclenche un GC, le tas sera corrompu

Mauvais exemple (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);
}
```

Mauvais exemple (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 est lu
- c_mk_pair peut déclencher le GC, qui peut réécrire x.

C'est un comportement indéfini.

Nos objectifs

- Détecter les racines non enregistrées
- Prévenir les comportements indéfinis dûs au GC
- Simplifier la gestion des racines

Contribution 1 : une API centrée sur les racines

Les "value" OCaml ne se comportent pas comme des valeurs du point de vue de C :

- leur adresse est capturée par le GC,
- leur valeur peut changer entre tout appel, si le GC est déclenché.

Passer un argument ("x") revient donc à *charger* depuis la mémoire la valeur à l'instant présent.

Passer des racines en argument (1/2)

Le problème vient du dé-référencement implicite lors du passage de l'argument.

- En Rust, Caml-oxide montre que le système de type est assez fin pour capturer cette distinction.
- En C, `CAMLroot` remplace les arguments de type "`value`" par des arguments de type "`value*`" (représentant des racines).

Passer des racines en argument (2/2)

Les valeurs de retour sont remplacées par un argument supplémentaire, une racine dans lequel le résultat sera stocké.

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

- le type de retour est `void` : il est peu probable d'imbriquer les appels.
- les arguments sont des pointeurs, il n'y a pas de risque de mélanger les deux styles : `mlroot_set_field(result, 0, &caml_alloc(2, 0));`

Forme normale administrative (ANF)

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

CAMLprim value c_mk_triplet(value x, value y,
value z)
{
  CAMLparam2(x, y);
  CAMLlocal2(result, tmp);
  mlroot_mk_pair(&tmp, &y, &z);
  mlroot_mk_pair(&result, &x, &tmp);
  CAMLreturn(result);
}
```

Programmation défensive

- Les arguments sont maintenant des pointeurs.
  Le dé-référencement implicite est devenu explicite mais n'a lieu que dans les primitives de `CAMLroot`.
- Ces pointeurs doivent être des racines, enregistrées auprès du GC.
- Un mode "défensif" optionnel permet de vérifier qu'une racine est bien enregistrée avant chaque dé-référencement.

Où en sommes-nous ? Avec un niveau d'indirection :

👍 Plus de comportements indéfinis :

- grâce à la "forme normale"
- grâce au passage par adresse, et non par valeur

👍 Un mode défensif optionnel pour détecter très tôt les erreurs de manipulation

👎 Légèrement plus verbeux

- Profil de performance similaire

Contribution 2 : allocation par région

L'API centrée sur les racines a permis d'écarter les problèmes de manipulation des valeurs.

Il reste à gérer les racines. Peut-on simplifier cet aspect ?

Idée : permettre l'allocation dynamique de racines dans une région.

Les régions

Lors du passage d'OCaml à C, une région est mise en place :

```
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, &x);
  CAMLregion_return(*result);
}
```

Les régions

Lors du passage d'OCaml à C, une région est mise en place.

- Peuplée avec les arguments
- Dynamiquement étendue lors de l'allocation de variables locales
- Libérée au retour à OCaml

Les régions

On peut retrouver un style direct et sûr avec l'allocation dynamique :

```
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);
}
```

Variantes (1/2) : les sous-régions

Toutes les racines sont libérées à la sortie du code C.
Si une fonction nécessite beaucoup de variables locales (boucle), cela peut introduire une fuite de mémoire.

Une variante : les sous-régions.

Celles-ci permettent de gérer localement des racines :

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

Variantes (2/2) : régions sans OCaml

OCaml n'autorise qu'un seul cœur à accéder au runtime à un instant donné. Un verrou permet de contrôler l'accès au runtime depuis plusieurs threads.

Un type de région permet de gérer l'accès à ce verrou :

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

L'API est défensive : à l'intérieur de cette région, les appels aux primitives (`mlregion_alloc`, `mlregion_get_field`, ...) échoueront.

Et en C++

- Utilisation des références pour supprimer la distinction entre valeurs et pointeurs de valeur
- "Resource Acquisition Is Initialization" (RAII) pour lier une région au scope lexical

Conclusion

Deux changements pour faciliter l'utilisation de la FFI d'OCaml :

- l'API centrée sur les racines couvre les mêmes besoins que celle fournie par OCaml en rajoutant de la sûreté
- le système de région peut simplifier le code, en particulier en C++, mais n'est pas aussi générale que celle de base
- en développement, sur https://github.com/let-def/cuite et https://github.com/let-def/camlroot

Travaux existants

- **O'Saffire**
  Vérificateur de bindings écrit manuellement. Plus maintenu.
- **Ctypes**
  Génération de bindings depuis un EDSL.
- **Caml-oxide**
  Preuve de concept en Rust, utilisant le système de type pour enforcer les invariants du GC.

RÉFÉRENCES

- 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

Mesures de performance

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 (fonctions)
mk_pair_root: FFI CAMLroot (tests désactivés)
mk_pair_root_safe: FFI CAMLroot (tests activés)