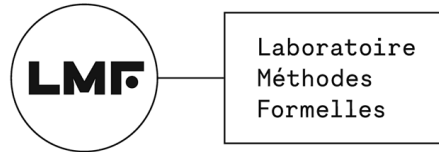


Optimisation du filtrage par motifs sur les types extensibles

JFLA 2026

Maël Coulmance, Vincent Laviron, Pierre Chambart

Inria



Au menu aujourd'hui

1. Types Extensibles
2. Schéma de Compilation
3. Compilation de *switch*
4. Prototype *Dispatch Dynamique*
5. Prototype *Hash Modulo Collisions*
6. Perspectives

Types Extensibles

- Type somme
- Ensemble de constructeurs peut être étendu
- Utilisé pour représenter les exceptions

```
(* Une déclaration de type *)  
type t = .. (* c'est la vraie syntaxe *)
```

```
(* Une extension de type *)  
type t += Andouillette | Boudin of int | Chipon of t
```

```
type t = ..

module Toulouse =
  struct
    type t += S
  end

module Morteau =
  struct
    type t += S
  end
```

```
let miam = function
  | Toulouse.S -> "Toulouse"
  | Morteau.S   -> "Morteau"
  | _           -> "Autre"
```

- Défini comme un alias vers un constructeurs pré-existant
- Invisible dans les signatures

```
type t = ..  
  
type t += A  
  
type t += B=A
```

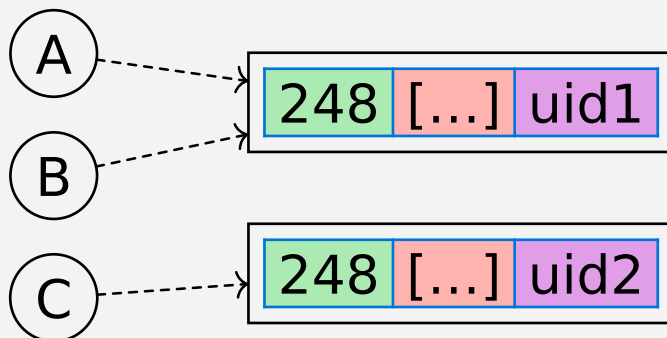
```
let foo = function  
| A -> "A"  
| B -> "B"      (* inaccessible *)  
| _ -> "Autre"  
  
foo A (* "A" *)  
foo B (* "A" *)
```

Schéma de Compilation

À l'initialisation du module:

- Allocation d'un bloc pour chaque nouveau constructeur
- Alias pour les constructeurs rebondés

```
type t = ..  
type t += A | C  
type t += B=A
```



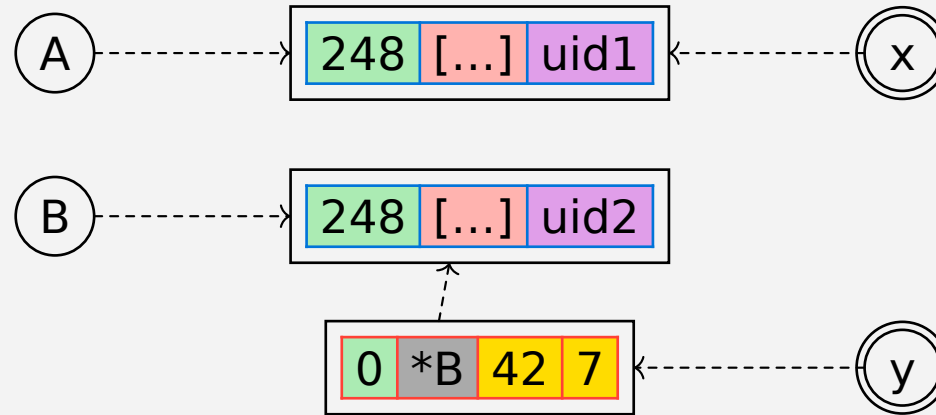
À la compilation, **aucun** moyen de connaître l'*uid*.
Celui-ci est généré au *runtime*.

À la compilation d'un filtrage **aucun** moyen de savoir quels sont les
constructeurs rebindés

- *constants*: alias vers le bloc alloué lors de l'extension
- *non-constants*: allocation d'un nouveau bloc

```
type t = ..  
type t += A | B of int * int
```

```
let x = A  
let y = B (42, 7)
```



- Séparation des constructeurs *constants* et *non-constants*
- Génération d'une chaîne de tests

```
type t = ..  
  
type t += A | B | C | D  
  
let foo =  
  function  
  | A -> "A"  
  | B -> "B"  
  | C -> "C"  
  | D -> "D"  
  | _ -> "unknown"
```

```
let foo x =  
  if x == A then "A"  
  else if x == B then "B"  
  else if x == C then "C"  
  else if x == D then "D"  
  else "unknown"
```

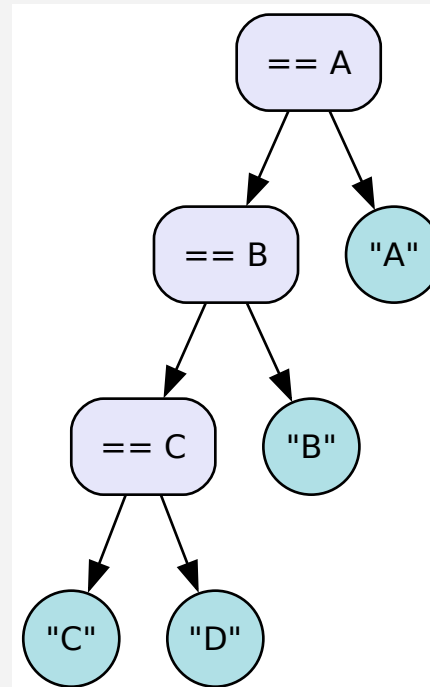
Rappel: On ne **connaît pas** les *uid*.

TEMPS LINÉAIRE !!!

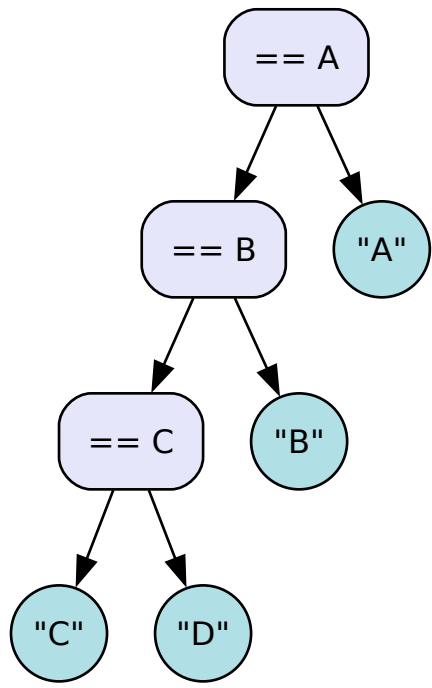
```
let foo =  
  function  
  | C1 -> "C1"  
  | C2 -> "C2"  
  | C3 -> "C3"  
  | C4 -> "C4"  
  (* ... *)  
  | Cn -> "Cn"
```

Compilation de *switch*

```
let foo x =  
  if x == A then "A"  
  else if x == B then "B"  
  else if x == C then "C"  
  else "D"
```



Chaine de tests
 $O(n)$



Arbre de décision
 $O(\log(n))$
ordre total

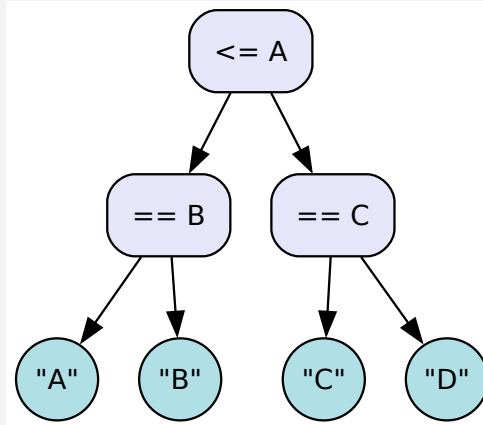
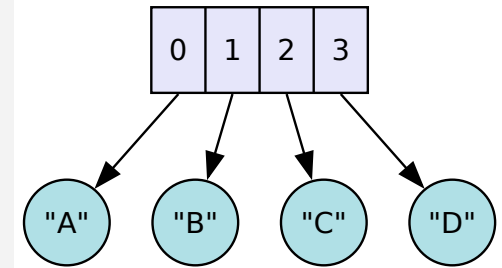


Table de symboles
 $O(1)$
ensemble dense



Prototype *Dispatch* ***Dynamique***

Problèmes:

- On ne connaît pas l'ensemble des constructeurs existants
- On ne sait pas dans quel ordre vont être créés les constructeurs

Solution ?

- Construire (dynamiquement) un arbre de décision sur les identifiants uniques

À la compilation:

- On associe à chaque constructeur filtré un identifiant statique
- On met à jour le filtrage en remplaçant chaque constructeur par son identifiant statique
- On génère une liste associative (constructeur -> identifiant statique) et un appel à une fonction de la bibliothèque standard.

Dans la bibliothèque standard: *camlinternalExtension*

```
val init_table : (extension_constructor * int) list -> (* à l'init *)  
                (uid:int -> (* au match *) int)
```

À l'exécution:

- On initialise la table (une seule fois)
- Pour chaque filtrage on récupère l'indice de la branche à partir de l'identifiant unique du constructeur, puis on switch

```
let miam = function
  Morteau    -> "Mo"
| Strasbourg -> "St"
| Toulouse   -> "To"
| _          -> "Autre"
```

```
(let
  table =
    (apply CamlinternalExtension.init_table [(Morteau, 3), (Strasbourg, 2), (Toulouse, 1)])
  foo =
    (function param
      (switch (apply table param)
        case int 3: "Mo"
        case int 2: "St"
        case int 1: "To"
        default:   "Autre"))))
```

```
let miam s =  
  let module M = struct  
    type t += Morteau  
  end in  
  match s with  
  | M.Morteau   -> "M"  
  | Strasbourg  -> "S"  
  | _           -> "Autre"
```

- *Lifting* du code d'initialisation de la table
- Fusion des tables équivalentes

Pour les cas extrêmes:

- Amélioration asymptotique notable

Pour le code existant:

- Coût d'initialisation des tables trop élevé

Prototype *Hash Modulo* *Collisions*

Problèmes:

- On ne connaît pas l'ensemble des constructeurs existants
- On ne sait pas dans quel ordre vont être créés les constructeurs
- *On ne connaît pas les identifiants uniques des constructeurs*

Solution ?

- Construire un arbre de décision sur les noms des constructeurs
- Inspiré de la manière dont sont compilés les variants polymorphes
- Optimiser la recherche en filtrant sur le hash du nom des constructeurs

Collisions: Deux constructeurs peuvent avoir le même nom et/ou le même hash.

Solution pour les variants polymorphes:

Vérifier au typage et refuser les programmes avec collisions

Solution pour les types extensibles:

Générer une chaîne de tests d'égalité sur les *uid*

- Défini comme un alias vers un constructeurs pré-existant
- Invisible dans les signatures

```
type t = ..
```

```
type t += A
```

```
type t += B=A
```

Rappel: Lors de la compilation d'un filtrage on ne sait pas quels sont les constructeurs rebindés.

Rebinding: Le rebindeur doit avoir le même hash que le rebindé

Première solution:

- Étendre le langage OCaml pour rendre cette information disponible

Problème:

- Solution complexe: les rebindings sont cachés par les signatures

Rebinding: Le rebindeur doit avoir le même hash que le rebindé

Deuxième solution:

- Interdire le rebinding pour les types optimisés

Conséquences:

- Différence entre types extensibles optimisés ou non
- Pas d'optimisation possible pour le code qui utilise le rebinding

Arbitrage:

- Rebinding peu utilisé en pratique

Modifications apportées au typeur:

- Ajout d'une annotation [`@strict`] aux déclarations de type
- Vérifier l'absence de rebinding pour les types *stricts*
- Vérifier la cohérence signature/implémentation

Rétrocompatible !

Schéma de compilation:

- Pour les types extensibles “classiques”: comme avant
- Pour les types extensibles “stricts”:
 - Analyse:
 - Séparer les constructeurs statiquement distinguables
 - Grouper les constructeurs par hash
 - Codegen:
 - Générer un arbre de décision sur les hash
 - Par hash, générer une séquence de tests

Pour les cas extrêmes:

- Amélioration asymptotique notable

Pour le code existant:

- Léger surcoût dans les pires cas:
 - Cas optimaux pour le schéma classique
 - Trop de collisions
- Amélioration en moyenne

Perspectives

Intuition: L'utilisation du hash nous donne un ensemble potentiellement dense.

- Hachage parfait
- Table de symboles (*hash modulo collisions* avec des tables)
- Peut aussi s'appliquer aux variants polymorphes

Intuition: les constructeurs sont ajoutés par scope

- Dans un scope on a accès à plus d'informations statiques
- Recherche dynamique sur les scopes
(réutilisation de nos schémas)

```
type scope = A | B | C
type s += S of scope

match s with
| S x -> match x with
        A -> "A" | B -> "B" | C -> "C"; ;
| _    -> "Autre"
```

Contributions:

- Variants extensibles “strictes”
- Conception de trois schéma de compilation
- Preuves de concept de deux de ces trois schèmes dans une branche du compilateur
- Validation sur du code réel (dont dune et le compilateur lui même)
- Benchmarks

Ouverture:

- Implémenter le troisième schéma et en étudier d'autres

Merci pour votre attention !