Un régime au concentré d'automate

Pierre-Marie Pédrot

PPS/πr^2

6 février 2013

Un problème pratique

- Fichiers objets vo de Coq
  - Représentent l'état du moteur interne
- Sérialisation de données OCaml
  - Structures du premier ordre
  - Fragment purement fonctionnel

Un problème pratique

- Fichiers objets vo de Coq
  - Représentent l'état du moteur interne
- Sérialisation de données OCaml
  - Structures du premier ordre
  - Fragment purement fonctionnel

Problème

Comment compresser ces données ?

1. Intelligemment.
2. Efficacement.

Pour quoi faire ?

- Moins de place sur le disque (sérialisation)
- Moins d'empreinte mémoire au chargement (désérialisation)
- Mais aussi un léger gain en temps
  - $\rightsquigarrow$ Comparaisons optimisées
  - $\rightsquigarrow$ Le GC vous salue bien

Yes, we can

- On utilise un algorithme bien connu
  - $\rightsquigarrow$ adapté de la théorie des automates
- Bizarrement, personne ne semblait y avoir pensé
- Et en plus, c'est généralisable à d'autres besoins
- « Un *hashconsing* » à postériori

Égalité générique vs. égalité physique

Avant de se lancer dans le cœur algorithmique, jetons un œil aux entrailles d'OCaml.

Égalité générique

$$\texttt{val}\ (\sim) : \alpha \rightarrow \alpha \rightarrow \texttt{bool}$$

- Insensible au type
- Analyse structurelle
- Récurse et peut boucler

Égalité physique

$$\texttt{val}\ (\equiv) : \alpha \rightarrow \alpha \rightarrow \texttt{bool}$$

- Insensible au type
- Comparaison de pointeurs
- Temps constant

Quelques théorèmes faux

Bonne définition

La fonction $(\sim)$ induit une relation d'équivalence.

Inclusion

Si $x \equiv y$ alors $x \sim y$.

Quelques théorèmes faux

Bonne définition

La fonction $(\sim)$ induit une relation d'équivalence.

Inclusion

Si $x \equiv y$ alors $x \sim y$.

Contre-exemple

Dans les deux cas, ceci est dû à la potentielle non-terminaison.

$$\texttt{let rec } l = 0 :: l \texttt{ in } l$$

Représentation mémoire OCaml

OCaml ne connaît que deux types de données en mémoire :

- Les entiers ;
- Les blocs, eux-même subdivisés en :
  - Tableaux contenant récursivement des données ;
  - Types de base OCaml (chaînes, flottants, etc.) ;
  - Autre (clôtures, objets, abstraits, etc.).

Représentation mémoire OCaml

OCaml ne connaît que deux types de données en mémoire :

- Les entiers ;
- Les blocs, eux-même subdivisés en :
  - Tableaux contenant récursivement des données ;
  - Types de base OCaml (chaînes, flottants, etc.) ;
  - Autre (clôtures, objets, abstraits, etc.).

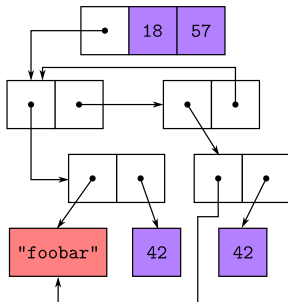
On travaille sur le sous-ensemble représenté par :

$$\begin{array}{rcl} \mathtt{data} & := & \mathtt{Int}\ \mathtt{of}\ \mathbb{N} \mid \mathtt{Ptr}\ \mathtt{of}\ \mathtt{ptr} \\ \mathtt{struct} & := & \mathtt{Arr}\ \mathtt{of}\ \mathtt{tag} \times \mathtt{data}\ \mathtt{array} \mid \mathtt{Str}\ \mathtt{of}\ \mathtt{string} \\ \mathtt{memory} & := & (\mathtt{ptr}, \mathtt{struct})\ \mathtt{map} \end{array}$$

Un exemple

```
let x = Some 42 in
let y = Some 42 in
let s = "foobar" in
let u = (s, x) in
let v = (s, y) in
let rec l =
  u :: v :: l in
(l, 18, 57)
```

$\mapsto$



L'égalité structurelle revisitée

On peut maintenant définir la vraie relation d'équivalence qui se cache derrière l'égalité structurelle, et qui valide les deux précédents théorèmes.

$$\frac{\text{pour tout } i \leq n \qquad x_i \sim y_i}{\texttt{Arr}\,(t, [|x_1 \ldots x_n|]) \sim \texttt{Arr}\,(t, [|y_1 \ldots y_n|])} \qquad \frac{}{\texttt{Str}\ s \sim \texttt{Str}\ s}$$

$$\frac{p \mapsto x \qquad q \mapsto y \qquad x \sim y}{\texttt{Ptr}\ p \sim \texttt{Ptr}\ q} \qquad \frac{}{\texttt{Int}\ n \sim \texttt{Int}\ n}$$

L'égalité structurelle revisitée

On peut maintenant définir la vraie relation d'équivalence qui se cache derrière l'égalité structurelle, et qui valide les deux précédents théorèmes.

$$\frac{\text{pour tout } i \leq n \qquad x_i \sim y_i}{\texttt{Arr}\,(t, [|x_1 \ldots x_n|]) \sim \texttt{Arr}\,(t, [|y_1 \ldots y_n|])} \qquad \frac{}{\texttt{Str}\ s \sim \texttt{Str}\ s}$$

$$\frac{p \mapsto x \qquad q \mapsto y \qquad x \sim y}{\texttt{Ptr}\ p \sim \texttt{Ptr}\ q} \qquad \frac{}{\texttt{Int}\ n \sim \texttt{Int}\ n}$$

Petit détail

Ces inférence sont à prendre **coinductivement** !
Il s'agit d'une **bisimulation**...

La spécification

On cherche à définir une transformation φ :

- qui à toute mémoire $\mathfrak{M}$ associe une mémoire $\varphi(\mathfrak{M})$
- qui à tout pointeur p de $\mathfrak{M}$ associe un pointeur $\varphi(p)$ de $\varphi(\mathfrak{M})$
- telle que pour tout pointeur p, on a $p \sim \varphi(p)$ (correction)
- telle que si $p \sim q$, alors $\varphi(p) = \varphi(q)$ (minimalité)

La transformation φ associe à toute mémoire la mémoire équivalente de **partage maximal**.

Correction sémantique

Question.

En supposant qu'on sait réaliser la spécification, peut-on remplacer ainsi un objet x par un autre y tel que $x \sim y$ tout en préservant le comportement observable du programme ?

Correction sémantique

Question.

En supposant qu'on sait réaliser la spécification, peut-on remplacer ainsi un objet x par un autre y tel que $x \sim y$ tout en préservant le comportement observable du programme ?

En général, non.

- $\rightsquigarrow$ Certaines (rares) fonctions ne respectent pas l'égalité structurelle ;
- $\rightsquigarrow$ Pas en présence d'effets de bord !

Fonctions non-structurelles

En fait, il n'y en a pas beaucoup. Essentiellement :

- L'égalité physique (évidemment) ;
- Les fonctions du module `Marshal` (heureusement).

Assez bizarrement, des fonctions sordides la respectent au contraire :

- L'égalité générique...
- Le module `Obj` en général ;
- La fonction de hash générique.

Effets de bord : la localisation, ça compte

Un contre-exemple assez anodin.

```
let p =
  let x = ref 0 in
  let y = ref 0 in
  (x, y)
```

```
let q =
  let z = ref 0 in
  (z, z)
```

```
let f (x, y) =
  x := 42;
  !x = !y
```

$\leadsto$ On a $\mathrm{p} \sim \mathrm{q}$ mais $\mathrm{f\ p} \not\sim \mathrm{f\ q}$.

Effets de bord : rencontre du troisième type

En fait, on peut faire bien pire : au *runtime*, OCaml a oublié les types, et notre égalité structurelle est insensible au type.

```
let p = Some false          let q = ref None
```

$\rightsquigarrow$ On a hélas p $\sim$ q. Remplacer comme des fous furieux est ici **violemment** incorrect vis-à-vis de la sémantique, et ce type d'exemples est une usine à `segfault`.

```
let f x = x := Some 42 in
let g = function None -> true | Some b -> not b in
let () = f q in g p
```

Domaine d'application

Conjecture

Le fragment purement fonctionnel de ML préserve le comportement observationnel par égalité structurelle.

Domaine d'application

Conjecture

Le fragment purement fonctionnel de ML préserve le comportement observationnel par égalité structurelle.

No proof.

$\rightsquigarrow$ Pas de preuve, mais un argument (pas trop foireux) : il existe des implémentations du lambda-calcul + constructeurs qui n'ont pas de notion de localisation.

Domaine d'application

Conjecture

Le fragment purement fonctionnel de ML préserve le comportement observationnel par égalité structurelle.

No proof.

$\rightsquigarrow$ Pas de preuve, mais un argument (pas trop foireux) : il existe des implémentations du lambda-calcul + constructeurs qui n'ont pas de notion de localisation.

... Et mon implémentation n'a pas fait segfaulter Coq.

Réaliser la spécification

Et maintenant, comment calculer le partage maximal d'une mémoire ?

Réaliser la spécification

Et maintenant, comment calculer le partage maximal d'une mémoire ?

Un indice chez vous

Automata are coalgebras.

Et maintenant, comment calculer le partage maximal d'une mémoire ?

Solution

1. Voir une mémoire comme un automate.
   - $\rightsquigarrow$ Pointeurs $\mapsto$ transitions entre deux états
   - $\rightsquigarrow$ Données internes $\mapsto$ transitions sur soi-même

2. Miniser l'automate

3. Retraduire l'automate en mémoire

Ça marche !

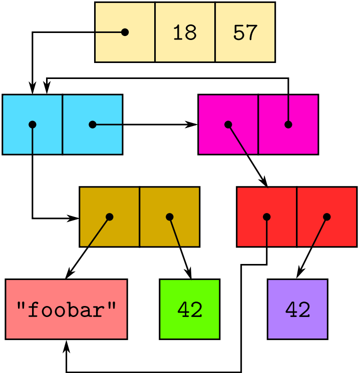
Correction

Le processus décrit au slide précédent génère bien le partage maximal.

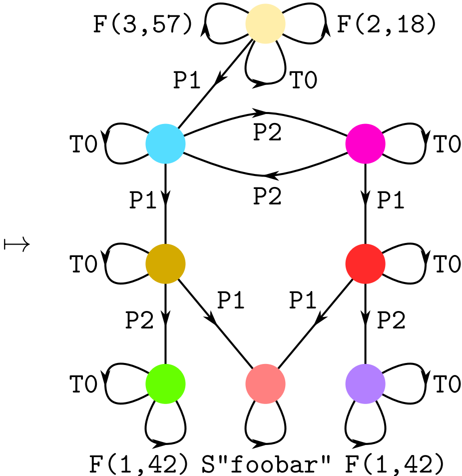
Coût

- Pour minimiser, on utilise l'algorithme de Hopcroft, en $O(n \log n)$.
- Les deux étapes de traduction sont en $O(n)$

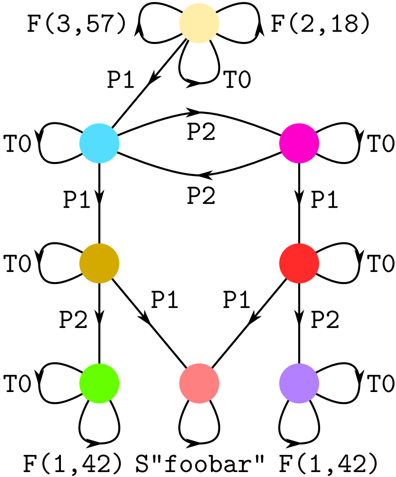
Notre leading example



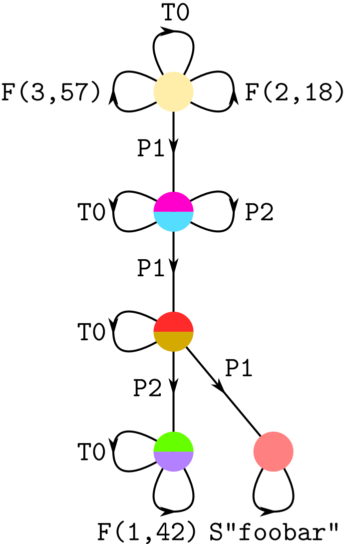
↦



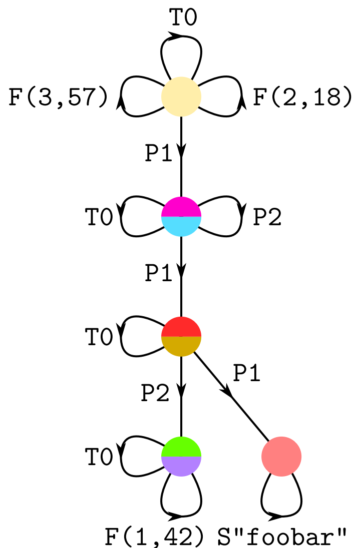
Notre leading example



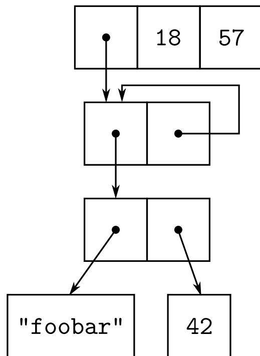
$\mapsto$



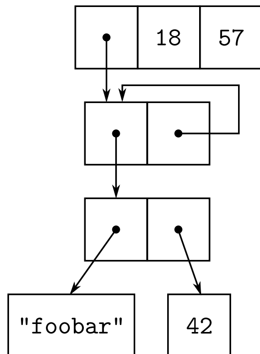
Notre leading example



$\mapsto$



Notre leading example



$\sim$

```
let s = "foobar" in
let r = Some 42 in
let p = (s, r) in
let rec l = p :: l in
(l, 18, 57)
```

En pratique

- Se présente sous la forme d'une fonction

$$\texttt{share} : \alpha \to \alpha$$

- Implémenté en OCaml via une désérialisation ad-hoc
  - $\rightsquigarrow$ Sous-optimal, mériterait une implémentation plus bas-niveau
- Généralisable aux structures de données non-mutables quelconques : il suffit de considérer $p \sim q$ ssi $p \equiv q$ quand p et q abstraits : clôtures, objets, pointeurs hors du tas OCaml...
- Rant collatéral : pourquoi diable l'égalité structurelle d'OCaml[1] boucle alors qu'elle pourrait calculer l'équivalence d'automates déterministes à peu de frais ?

1. Même si elle n'est pas *secure*...

Benchs

Appliqué avec succès sur les structures de données présentes dans les fichiers objets de Coq :

- Environ 30% de gain d'espace, orthogonal avec `bzip`
- Entre 5 et 10% de gain de temps de compilation sur la bibliothèque standard, mais le coût de la compression est très supérieur à ce gain

Conclusion

- Une application bête et méchante d'un algorithme bien connu
- ... mais ça marche
- Cet algorithme peut servir dans des cas plus généraux (seule contrainte : l'utilisation purement fonctionnelle des structures compactées)

Des questions ?