

# Analyse de l'utilisation mémoire des applications OCaml sans changer leur comportement

Groupe de Travail - IRILL

**Çagdas Bozman**<sup>1,2,3</sup>

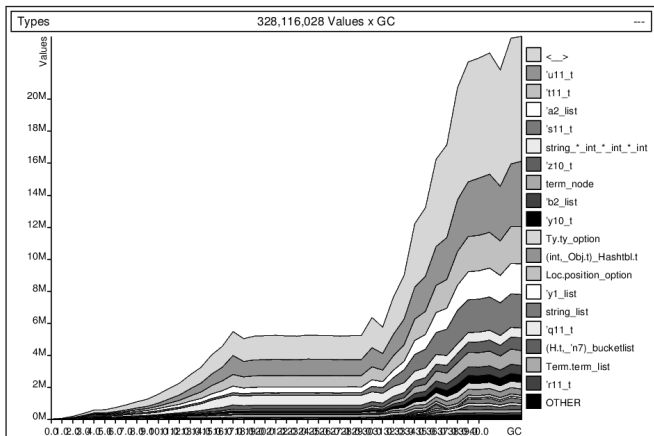
OCamlPro<sup>1</sup> | INRIA<sup>2</sup> | ENSTA ParisTech<sup>3</sup>

13 Février 2014 - Paris

# Problème Mémoire

- Ce que nous voulons ?
  - Étudier le comportement des programmes OCaml
  - Développer des outils de profiling
  
- Pourquoi ?
  - Réduire la quantité de mémoire utilisée
  - Fixer les fuites mémoires (si possible)
  - Passer le moins de temps possible dans la gestion mémoire

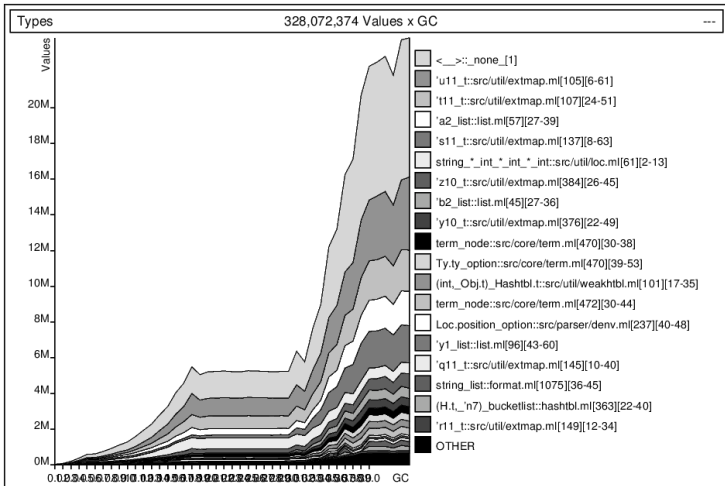
# Exemple Concret – Why3<sup>1</sup> (1/2)



<sup>1</sup>Why3 est une plateforme de vérification de programme  
(<http://why3.lri.fr/>)

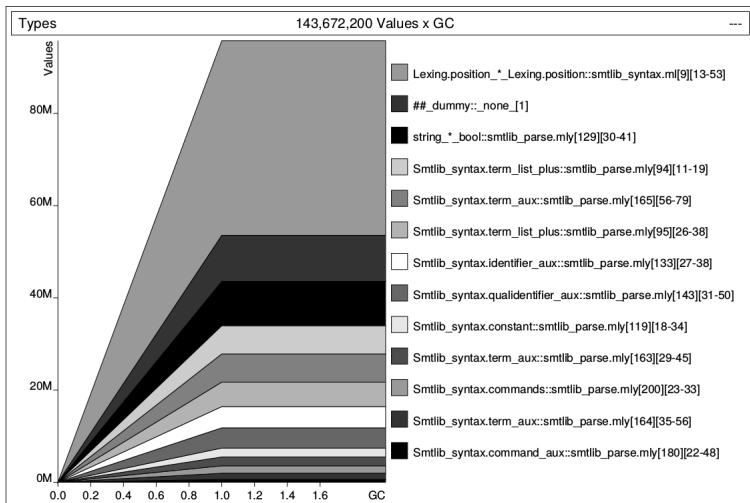
# Example Concret – Why3 (2/2)

Avec plus de précision sur les locations dans le code source



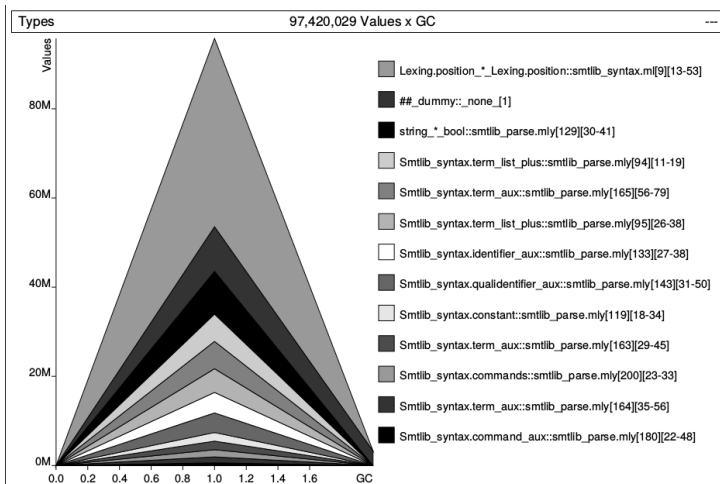
# Fuite mémoire dans Alt-Ergo (1/2)

Avant



# Fuite mémoire dans Alt-Ergo (2/2)

Après



# Comment obtenir ce graphe ?

```
$ opam switch 4.01.0+memprof
```

```
$ opam install why3
```

```
$ OCAMLRUNPARAM=m why3replayer.opt -C why3.conf p9_16
```

cette étape génère plusieurs snapshots du tas

Pas besoin de changer le code source ni les options de compilation.  
Pas d'impact sur le temps d'exécution.

```
$ opam install ocp-memprof
```

```
$ ocp-memprof -loc -sizes PID
```

cette étape analyse tous les snapshots générés

Plus qu'à regarder les graphes.

# Snapshots

Qu'est-ce qu'un *Snapshot* ?

- Version compressée du tas
- Contient les identifiants de location, graphe avec les pointeurs, etc.
- Globales (toplevel modules)

Comment on obtient ces graphes ?

- Scan linéaire de tous les chunks<sup>2</sup> qui contiennent des ensembles de blocs.

---

<sup>2</sup> grand bloc de mémoire



# Générer un snapshot

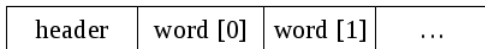
## Deux façons de générer des snapshots

- Utiliser `OCAMLRUNPARAM=m` pour forcer un programme à générer un snapshot après chaque GC majeur (automatique)
- Demander explicitement au programmeur de générer un snapshot (manuel):
  - en lui envoyant un signal HUP (très utile pour des applications comme les serveurs, cf *mldonkey*)
  - en annotant le programme à l'aide du module GC, en utilisant la fonction  

```
val dump_heap : string -> unit
```

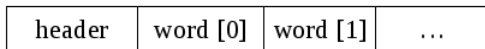
# Patch du compilateur (1/3)

Bloc mémoire OCaml:

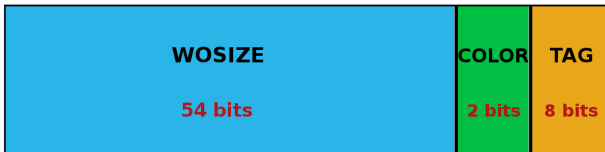


# Patch du compilateur (1/3)

Bloc mémoire OCaml:

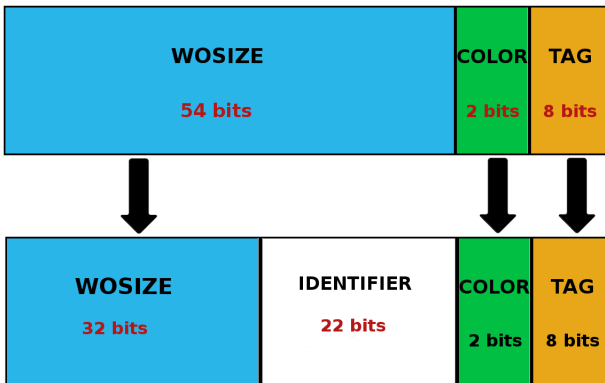


En-tête d'un bloc OCaml (un mot mémoire) sur les machines 64-bit:



## Patch du compilateur (2/3)

L'en-tête après notre modification:



# Patch du compilateur (3/3)



- Impact minimal sur les performances (seulement lors de la génération des snapshots)

# Patch du compilateur (3/3)



- Impact minimal sur les performances (seulement lors de la génération des snapshots)
- Pas de surcoût en espace mémoire

## Patch du compilateur (3/3)



- Impact minimal sur les performances (seulement lors de la génération des snapshots)
- Pas de surcoût en espace mémoire
- Pas d'impact sur le GC (son comportement n'est pas modifié)

## Patch du compilateur (3/3)



- Impact minimal sur les performances (seulement lors de la génération des snapshots)
- Pas de surcoût en espace mémoire
- Pas d'impact sur le GC (son comportement n'est pas modifié)



- **Seulement sur les plateformes 64-bits**



# Patch du compilateur (3/3)



- Impact minimal sur les performances (seulement lors de la génération des snapshots)
- Pas de surcoût en espace mémoire
- Pas d'impact sur le GC (son comportement n'est pas modifié)



- Seulement sur les plateformes 64-bits
- Les identifiants de location sont limités ( $2^{22} \sim 4$  millions d'allocations)

# Patch du compilateur (3/3)

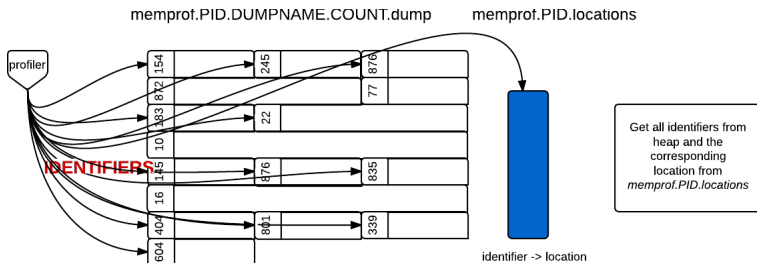


- Impact minimal sur les performances (seulement lors de la génération des snapshots)
- Pas de surcoût en espace mémoire
- Pas d'impact sur le GC (son comportement n'est pas modifié)



- Seulement sur les plateformes 64-bits
- Les identifiants de location sont limités ( $2^{22} \sim 4$  millions d'allocations)
- Taille maximum des blocs est de 32Go

# Un outil basé sur les identifiants



# Un outil basé sur le graphe

Ce que nous faisons avec les pointeurs:

```
val find_block : heap -> pointer -> block
val accessible_blocks : ?depth:int -> heap -> PtrSet.t ->
PtrSet.t
```

```
val words : heap -> PtrSet.t -> int
val types : typed_heap -> PtrSet.t -> TySet.t
val blocks_of_type : typed_heap -> Types.type_expr ->
PtrSet.t
```

```
val arrays : typed_heap -> PtrSet.t
val constr : typed_heap -> PtrSet.t
val objects : typed_heap -> PtrSet.t
val tuple : ?size:int -> typed_heap -> PtrSet.t
```

# Conclusion

La suite ?

- Améliorer le framework avec les pointeurs:
  - Plus de types ? (c.f. thèse G.Henry)
  - Durée de vie des valeurs (nombre de GC par exemple)
- Tester sur des exemples concrets:
  - Why3
  - Alt-ergo
  - OCaml, etc.

Questions ?