Analyse

Bruno Dufour
Université de Montréal
dufour@iro.umontreal.ca

Analyse de programmes

- **Objectif**: analyser ou déduire certaines propriétés d'un programme automatiquement
- Traditionnellement utilisée par les compilateurs pour optimiser le code généré
  - Ex: valeurs constantes, calculs redondants, cibles des appels, etc.
- De nos jours, l'analyse de programmes est souvent utilisée pour construire des outils pour les programmeurs

Analyse et outils

- Exemples d'utilisation de techniques d'analyse:
  - vérification
  - compréhension
  - transformation
  - décompilation
  - obfuscation
  - optimisations
  - etc.
- Pour comprendre les résultats et les limitations des outils, il est nécessaire de comprendre les techniques utilisées

Techniques d'analyse

- 2 techniques de base :
  - **Analyse statique** : basée sur l'analyse du code (source ou compilé)
  - **Analyse dynamique** : basée sur l'observation d'un programme durant son exécution

Analyse dynamique

Analyse dynamique

- L'analyse dynamique permet:
  - d'obtenir des résultats plus précis pour une ou plusieurs exécutions concrètes
  - d'obtenir de l'information de nature temporelle à propos de l'exécution
  - d'obtenir de l'information sur la fréquence ou l'importance de certains événements
    - Ex : une méthode est appelée très souvent, beaucoup de tableaux sont créés, etc.

Utilité de l'analyse dynamique

- Compréhension de programmes
- Détection de phases d'exécution
- Analyse de l'utilisation des ressources
  - Analyse de performance
  - Analyse de l'utilisation de la mémoire
- Optimisations JIT
- Débogage
- etc.

Collection de l'information

Instrumentation

- Le code source ou exécutable est modifié pour produire l'information nécessaire
- Information limitée à celle accessible par l'application
- Implémentation difficile
- Généralement efficace

Interface de profilage

- Le medium d'exécution (CPU, JVM, etc.) permet l'accès à l'information dynamique
- Information limitée à celle supportée par l'interface
- Facile d'utilisation
- Généralement peu efficace

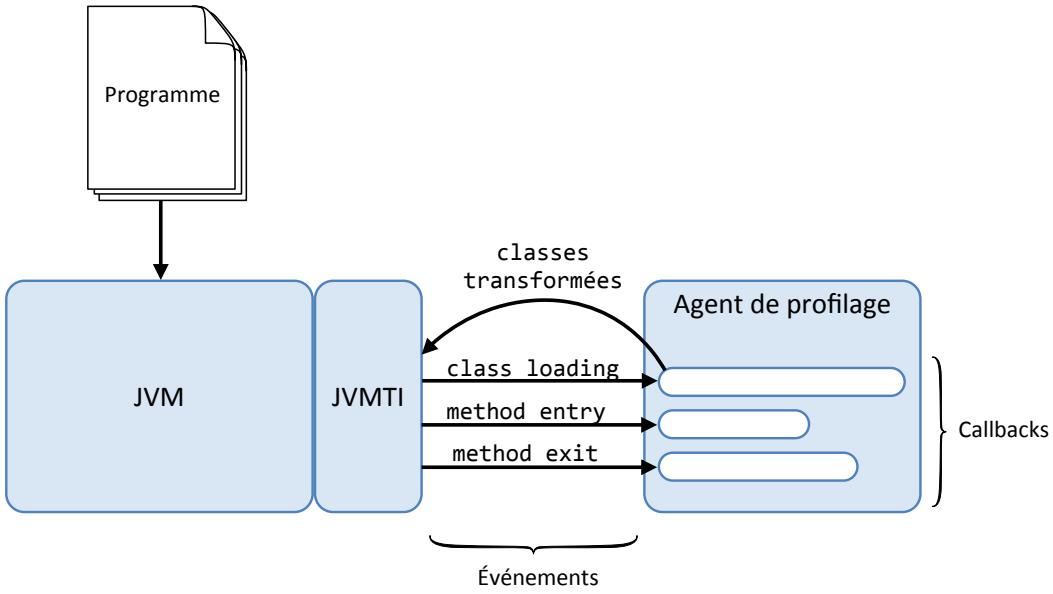
Instrumentation

```
public static int[] append(int i, int[] a) {
    int[] r;
    Analysis.recordEntry("append");

    if (a != null) {
        Analysis.recordAlloc("int[]", a.length + 1);
        r = new int[a.length + 1];
        Analysis.recordCall("System.arraycopy");
        System.arraycopy(a, 0, r, 1, a.length);
    } else {
        Analysis.recordAlloc("int[]", 1);
        r = new int[1];
    }
    r[0] = i;

    Analysis.recordExit("append");
    return r;

}
```


Instrumentation

```
public static int[] append(int i, int[] a) {
    int[] r;
    Analysis.recordEntry("append");

    try {
        if (a != null) {
            Analysis.recordAlloc("int[]", a.length + 1);
            r = new int[a.length + 1];
            Analysis.recordCall("System.arraycopy");
            System.arraycopy(a, 0, r, 1, a.length);
        } else {
            Analysis.recordAlloc("int[]", 1);
            r = new int[1];
        }
        r[0] = i;

        Analysis.recordExit("append");
        return r;
    } finally {
        Analysis.recordExit("append");
    }
}
```

Interface de profilage



Types d'analyse dynamique

- En ligne (*Online*)
  - Le programme est évalué au cours de l'exécution
  - Des calculs complexes peuvent perturber l'exécution
  - Seulement l'information pertinente est enregistrée
- Hors ligne (*Offline*)
  - Le programme est évalué après l'exécution (post-mortem) à l'aide de traces d'exécution
  - L'impact sur l'exécution est diminué
  - La quantité d'information à enregistrer peut être énorme (et proportionnelle au temps d'exécution)

Traces d'exécution

- Enregistre certains **événements** survenus lors de l'exécution
  - Exécution d'instructions / blocs de base / branches
  - Appels de méthodes
  - Lecture / écriture de valeurs en mémoire
  - Etc.
- Une trace peut contenir plusieurs types d'événements
- Peut facilement atteindre plusieurs GB pour des exécutions relativement courtes
  - La nature des traces permet de développer des techniques de compression très efficaces

Analyses JIT

- JIT = Just-in-time (juste à temps)
  - L'interprétation pure du code est trop lente
  - Les interpreteurs modernes effectue de la compilation durant l'exécution
- Les compilateurs JIT utilisent de l'information dynamique pour guider les optimisations (*feedback-directed optimisations*, ou *FDO*) qui permet :
  - de sélectionner les optimisations les plus profitables
  - d'effectuer des optimisations qui ne sont pas possibles statiquement
  - d'invalider certaines optimisations si la situation change

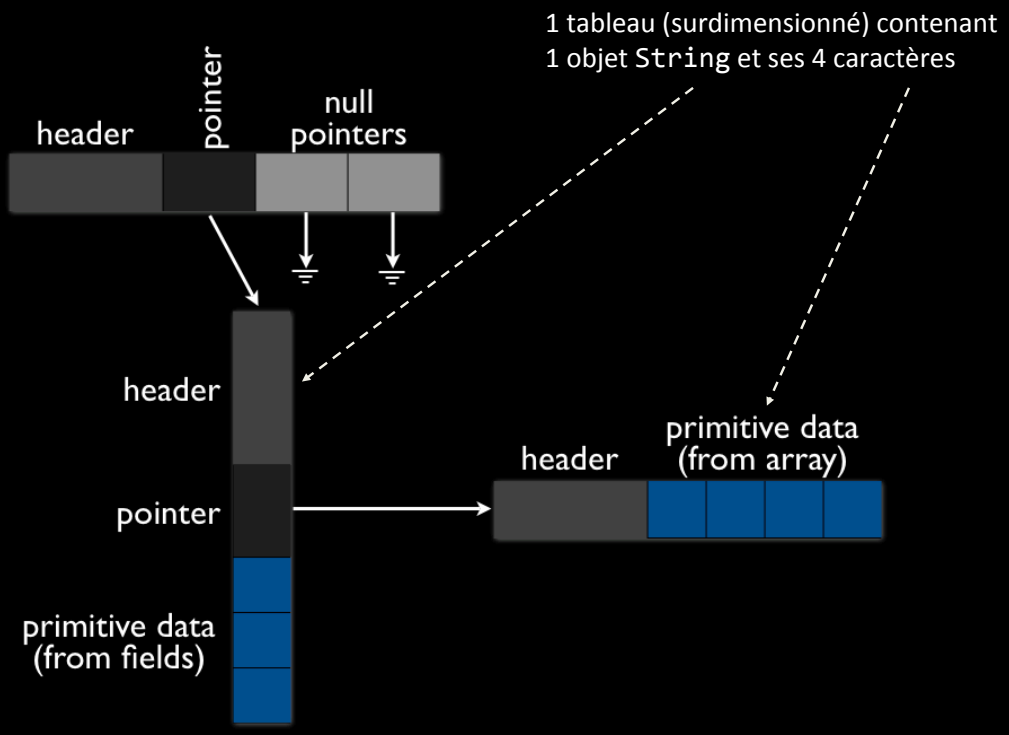
Exemples d'optimisations de JIT

- **Incorporation de méthodes** (*inlining*): permet d'éviter un appel en remplaçant l'appel par le code de la méthode appelée
- **Restructuration du code**: utilise l'information sur la fréquence d'exécution pour maximiser la localité sur le chemin le plus souvent emprunté
- **Versionnement** du code: génère des versions spécialisées d'un même fragment de code à utiliser dans différents contextes
- et bien d'autres…

Analyse de la mémoire

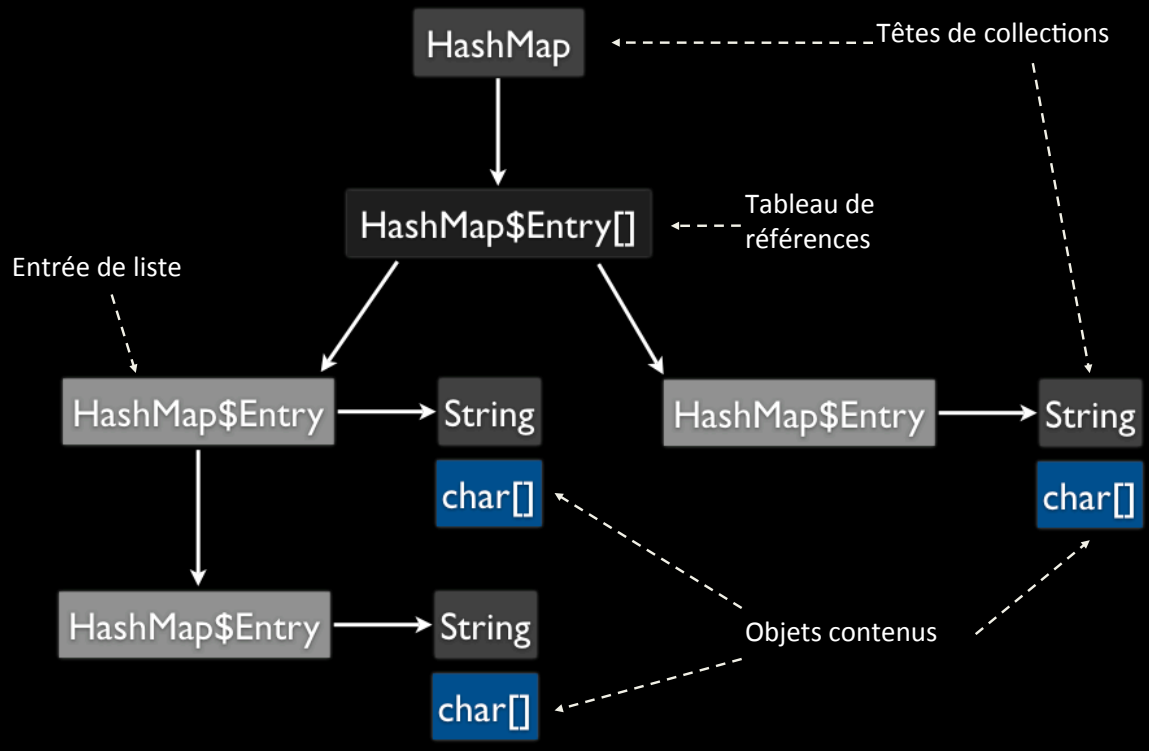
- Identification des fuites de mémoire (*memory leaks*)
- Analyse d'objets de tas (*heap*)
  - Nombre/taille/etc des objets
  - Classification de la mémoire utilisée
  - Connectivité/structure des objets
  - etc.

Utilisation des octets



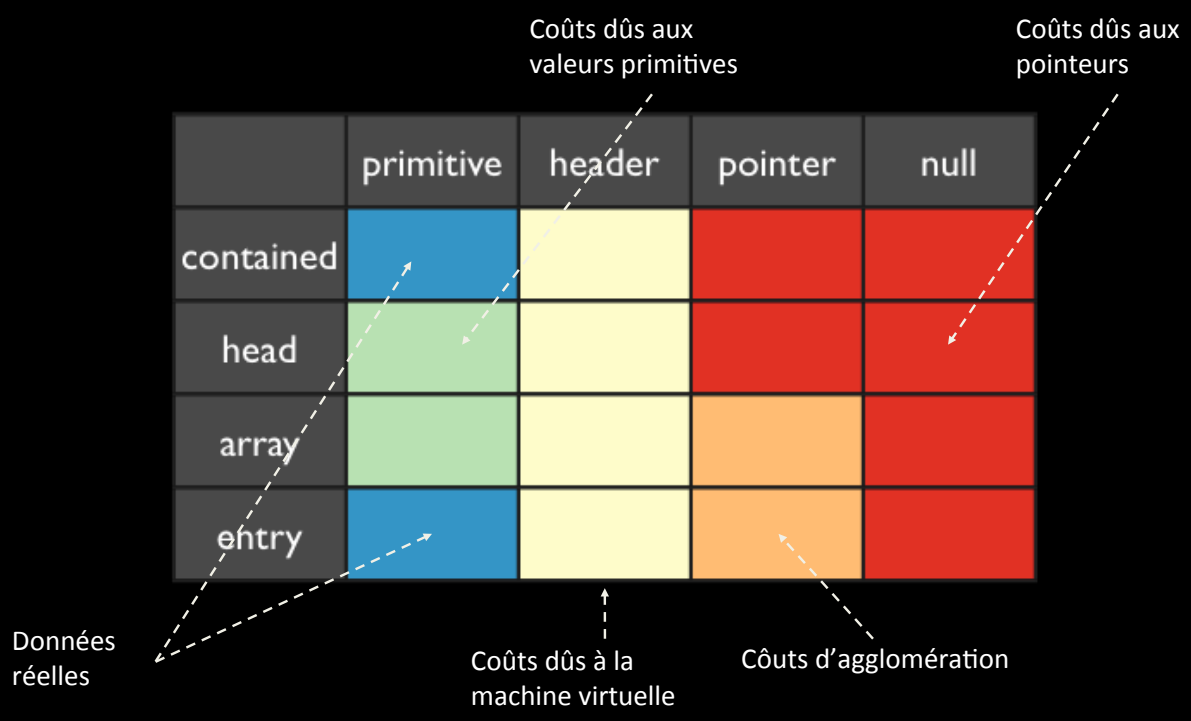
Source: N. Mitchell & G. Sevitsky, "The Causes of Bloat, The Limits of Health", OOPSLA'07

Agglomération d'objets en structures



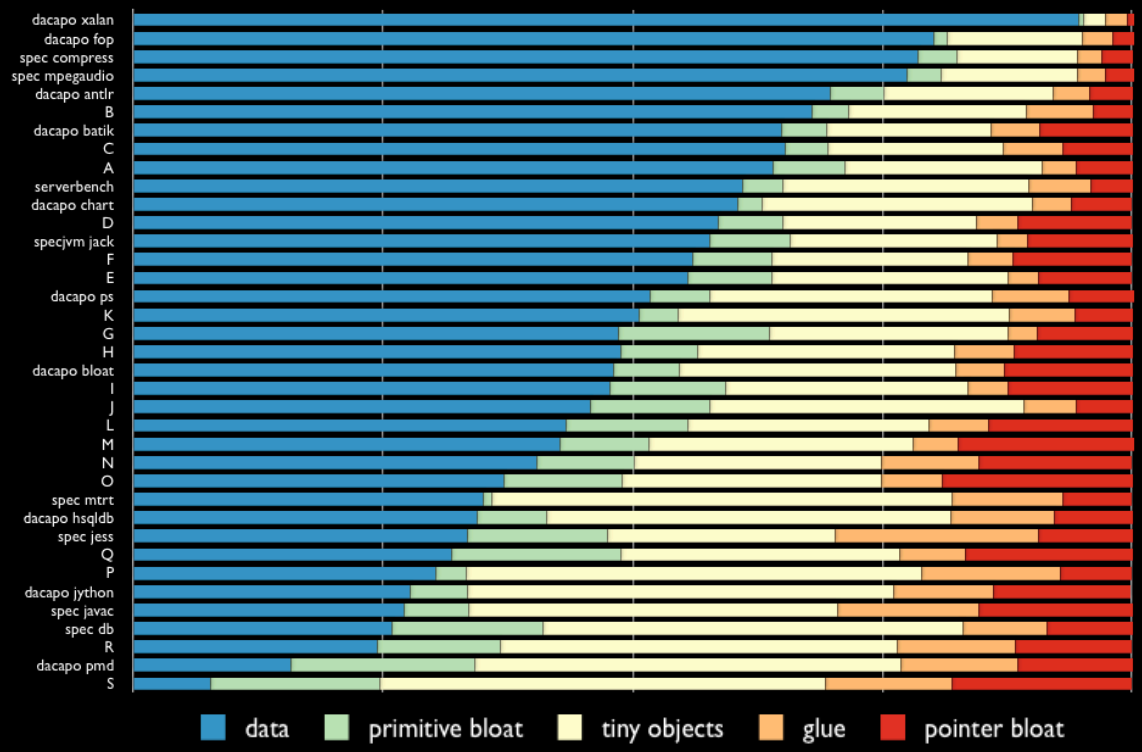
Source: N. Mitchell & G. Sevitsky, "The Causes of Bloat, The Limits of Health", OOPSLA'07

Classfication des octets



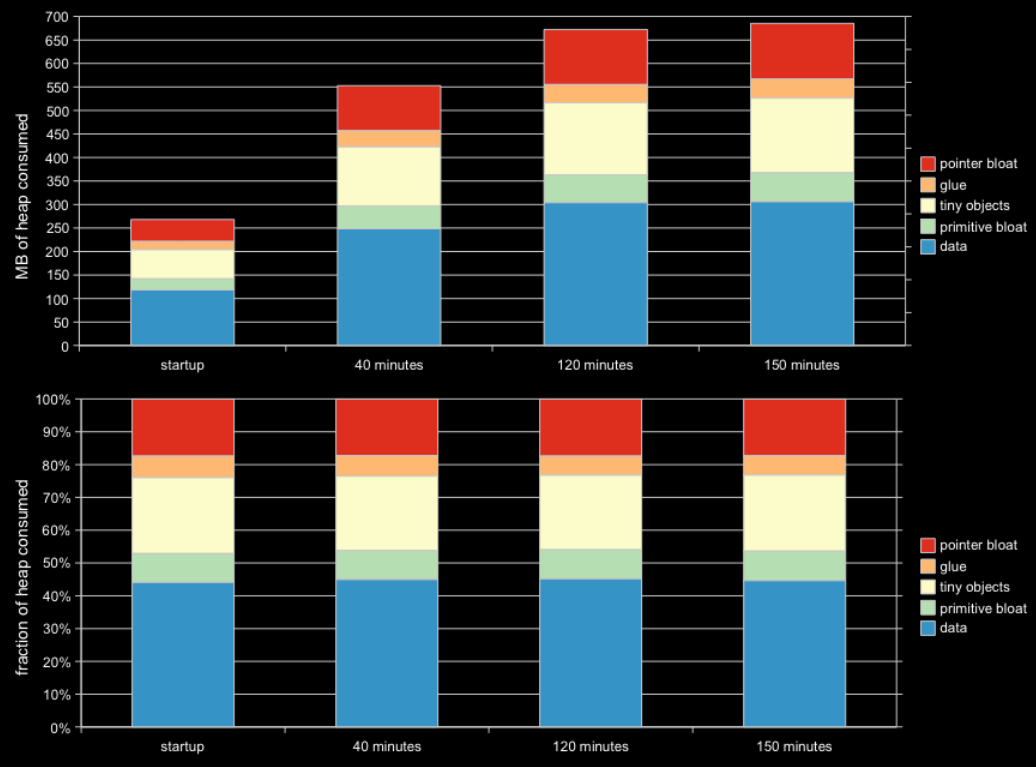
Source: N. Mitchell & G. Sevitsky, "The Causes of Bloat, The Limits of Health", OOPSLA'07

Résultats



Source: N. Mitchell & G. Sevitsky, "The Causes of Bloat, The Limits of Health", OOPSLA'07

Évolution et échelle



Source: N. Mitchell & G. Sevitsky, "The Causes of Bloat, The Limits of Health", OOPSLA'07

Analyse statique

Analyse statique

- L'analyse statique permet:
  - d'obtenir des résultats qui sont vrais pour toutes les exécutions d'un programme
  - de prouver certaines propriétés à propos d'un programme
  - d'analyser du code incomplet (dépend du type d'analyse effectué...)
    - Ex: analyser une bibliothèque sans connaître le programme client

Utilité de l'analyse statique

- Compréhension de programmes
- Optimisations
- Vérification
  - Sécurité
  - Absence de bogues
  - Garanties de performance (*WCET - Worst Case Execution Time*)
  - etc.
- Compréhension de programmes
- Aide au développement (ex: *code completion*)
- etc.

Représentations de programmes

- Code source (haut niveau)
- Code 3-addresses (niveau intermédiaire)
  - resultat ← <opérande1> <opérateur> <opérande2>
  - Par extension, d'autres opérations qui n'ont pas 2 opérandes sont aussi acceptées
- Code compilé (bas niveau)
  - Java: bytecode
  - En Java, il est relativement facile de générer du code 3-addresses à partir de bytecode

Code source

```
public static void factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Code 3-addresses

```
public static void factorial(int) {
        int n, $i1, $i2, $i3;

        n := @parameter0: int;
        if n > 1 goto label0;

        return 1;

     label0:
        $i1 = n - 1;
        $i2 = staticinvoke <fact: int factorial(int)>($i1);
        $i3 = n * $i2;
        return $i3;
}
```


Code compilé (*bytecode*)

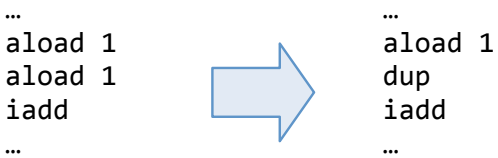
```
 public static void factorial(int) {
    0:   iload_0             // push n
    1:   iconst_1            // push 1
    2:   if_icmpgt 7         // if (n > 1) goto 7
    5:   iconst_1              // push 1
    6:   ireturn               // return 1
                             // else
    7:   iload_0               // push n
    8:   iload_0               // push n
    9:   iconst_1              // push 1
    10:  isub                  // compute (n-1)
    11:  invokestatic #2;      // call factorial(int)
    14:  imul                  // Compute n * factorial(n-1)
    15:  ireturn               // Return computed value
 }
```

Types d'analyse statique

- Locale (*Peephole*): quelques instructions
- Intraprocédurale: une seule méthode
- Interprocédurale: un programme complet ou un fragment de programme

Analyses locales

- Requièrent uniquement la séquence des instructions du programme



Exemple - analyse intraprocédurale

```
int fib(int m) {
    if (m <= 1) {
        return m;
    } else {
        int f0 = 0, f1 = 1, f2, i;

        for (i = 2; i <= m; i++) {
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
        }

        return f2;
    }
}
```

Q: Ce code compile-t-il ?

A: Non.

```
fib.java:12: variable f2 might not have been initialized
        return f2;
               ^
```


Exemple - analyse intraprocédurale

```
int fib(int m) {
    if (m <= 1) {
        return m;
    } else {
        int f0 = 0, f1 = 1, f2, i;

        for (i = 2; i <= m; i++) {
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
        }

        return f2;
    }
}
```

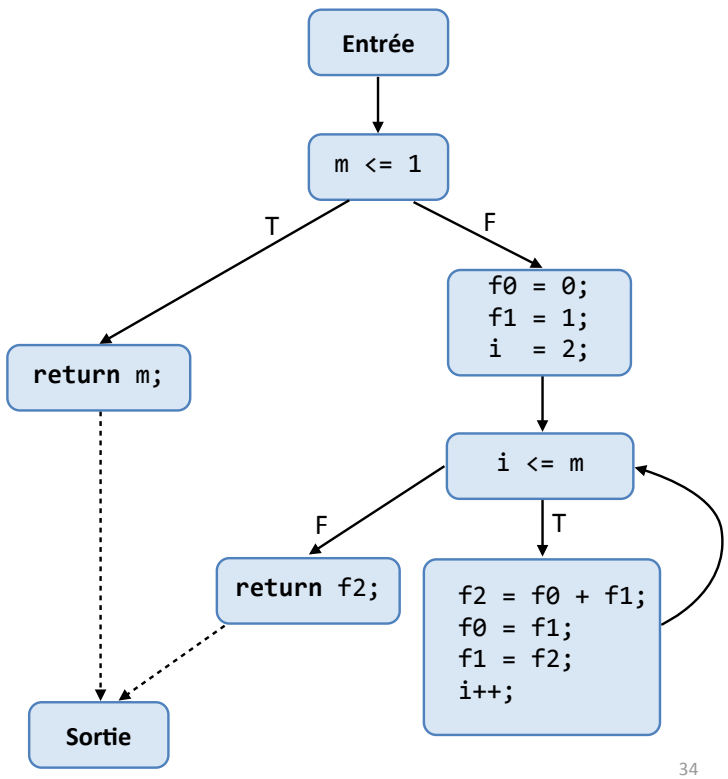
Q: Comment le compilateur détecte-t-il des situations semblables ?

Analyses intraprocédurales

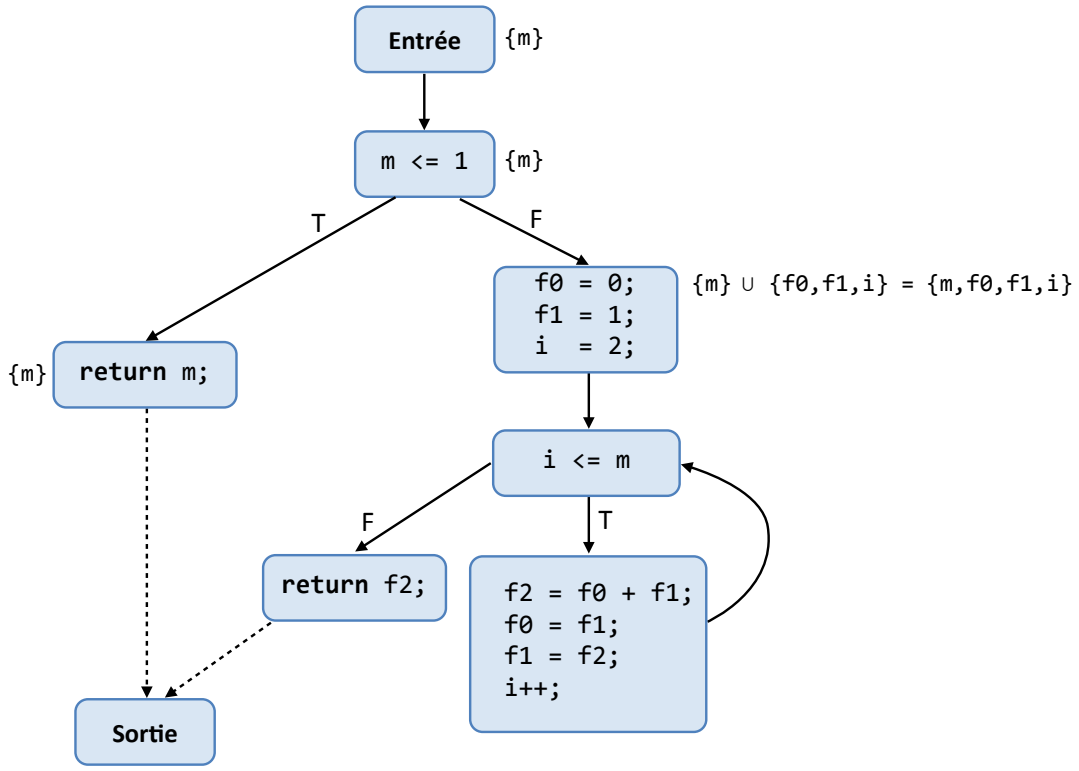
- Requiert un graphe de flot de contrôle pour la méthode/ fonction analysée
  - Décrit les chemins d'execution possibles en utilisant un graphe
  - Produit par une analyse de flot de contrôle
  - Nœuds: blocs de base (basic blocks), représentent des séquences d'instructions avec une entrée et une sortie

Graphe de flot de contrôle

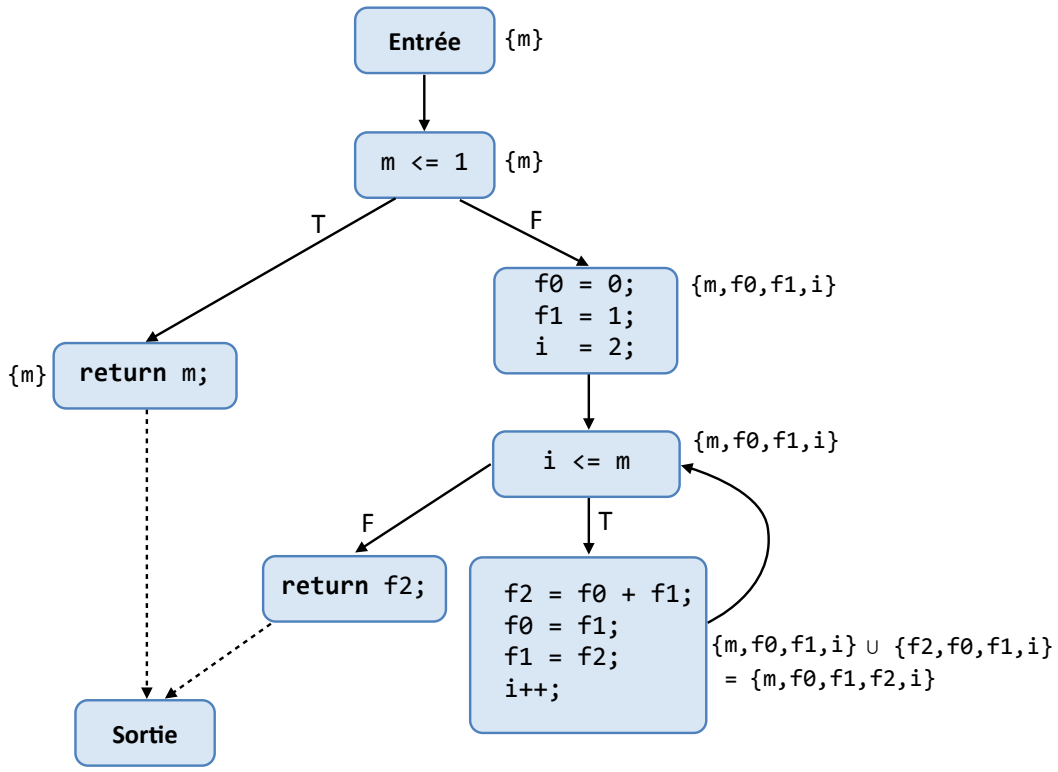
```
int fib(int m) {
  if (m <= 1) {
    return m;
  } else {
    int f0 = 0, f1 = 1, f2, i;

    for (i = 2; i <= m; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }

    return f2;
  }
}
```



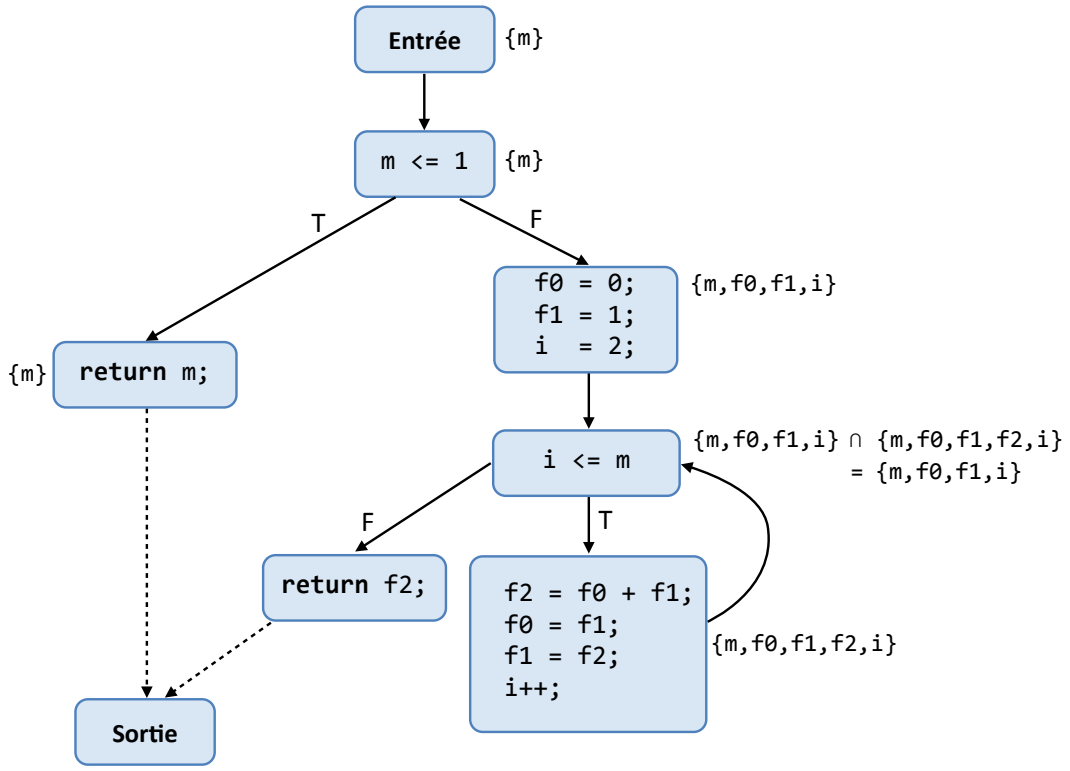
Analyse intraprocédurale



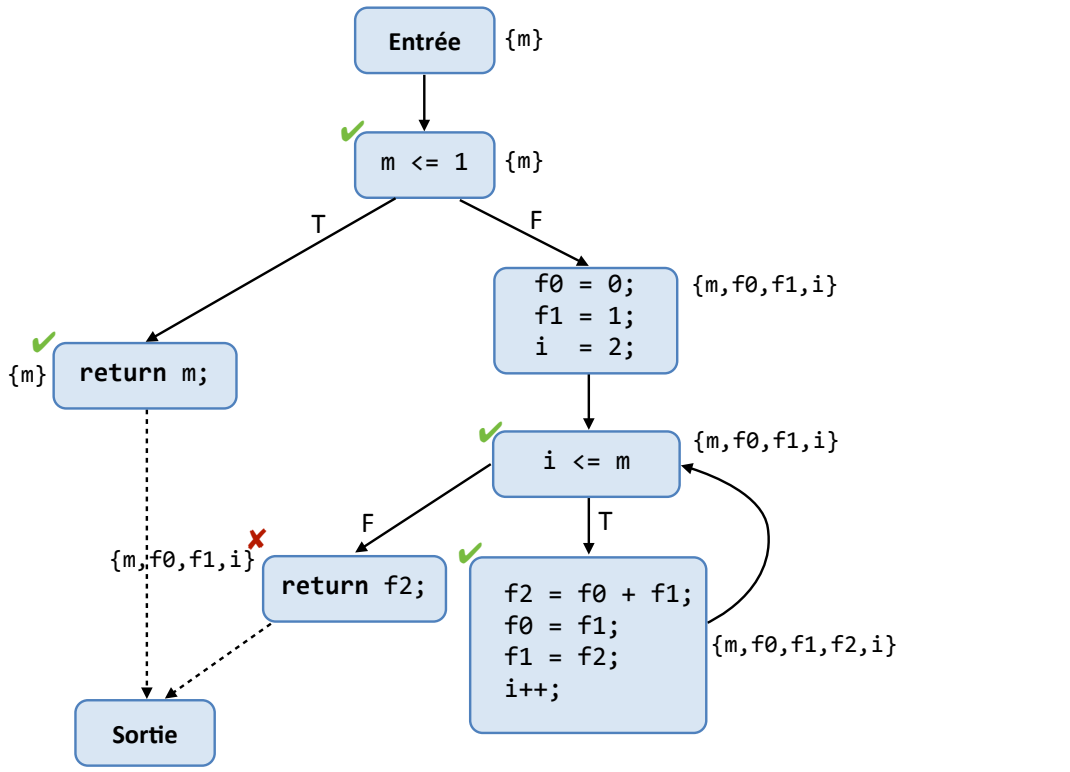
Analyse intraprocédurale



Analyse intraprocédurale



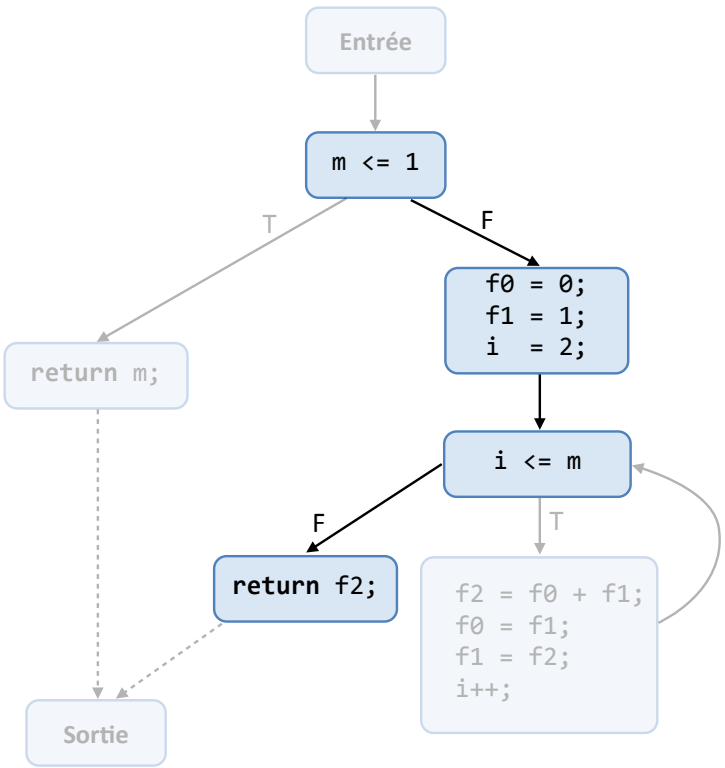
Analyse intraprocédurale



Analyse par flot de données

- L'exemple précédant utilise la technique d'**analyse par flot de données**
  - Les données sont associées à plusieurs points d'un programme (ex : instructions ou au blocs)
  - Chaque instruction peut ajouter ou retirer des valeurs
  - Les valeurs sont propagées d'une instruction à l'autre jusqu'à ce que les valeurs convergent (aucun changement, point fixe)
    - En général, il faut garantir que le point fixe sera atteint en prouvant certains propriétés mathématiques.
  - Autres exemples:
    - Vérification que tous les chemins d'exécution d'une méthode contiennent une instruction 'return' du type approprié
    - Identification du code inaccessible d'une méthode

Analyse conservatrice



Contraintes

m > 1

i = 2

i > m

$\therefore m > 1 \wedge i = 2 \wedge i > m$
$\equiv m > 1 \wedge m < 2$

n'admet aucune solution entière, donc cette exécution est impossible.

Analyses conservatrices

- En général, il est impossible de déterminer le comportement précis d'un programme sans l'exécuter
  - Peut être réduit au problème de l'arrêt de Turing (*halting problem*), qui est indécidable
- L'analyse statique calcule donc une **approximation** du comportement réel
- Elle peut :
  - Surestimer le comportement : on parle d'analyse **sûre** (*safe*)
  - Sous-estimer le comportement : on parle d'analyse **non-sûre** (*unsafe*)

Analyses interprocédurales

- Une analyse intraprocédurale est très limitée.
  - Dans plusieurs cas, il faut pouvoir considérer le programme en entier
  - Les appels de méthodes peuvent avoir des conséquences importantes sur les résultats calculés
- Une analyse interprocédurale, contrairement à l'analyse intraprocédurale, peut propager les données à travers les appels
- L'analyse interprocédurale doit pouvoir déterminer les appels possible à un site d'appel donné
  - Indécidable en général, il faut donc faire une approximation

Problèmes ouverts

- Exceptions
- Réflection / introspection
- Méthodes natives
- Chargement de classes dynamique
- Programmes incomplets

Comparaison des techniques

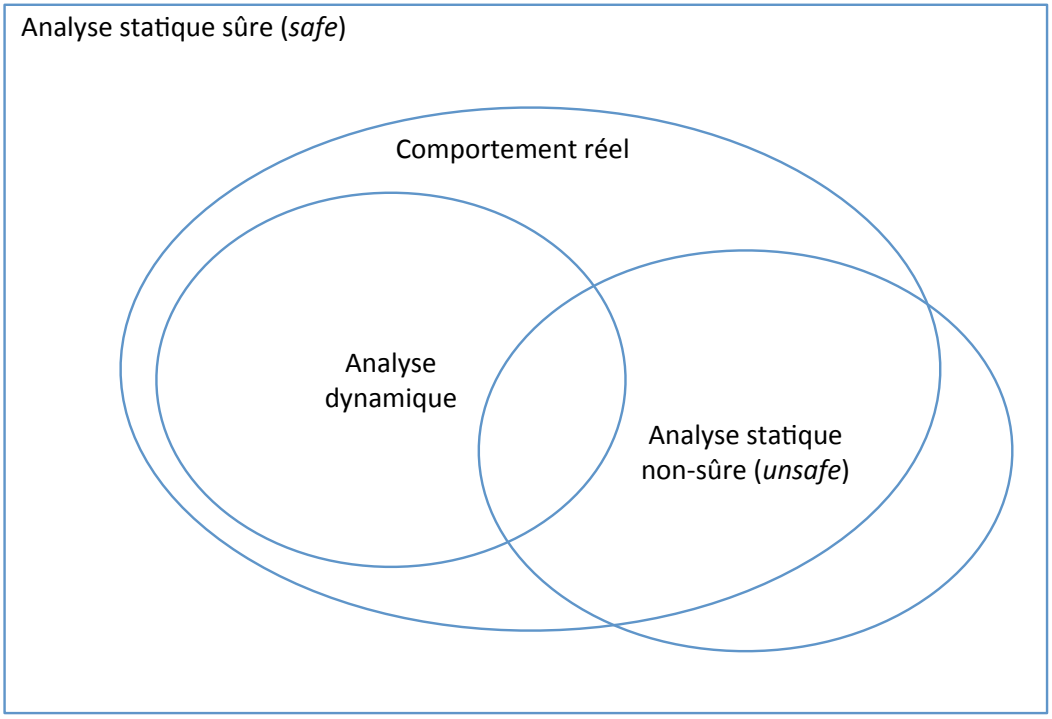
Analyses statiques

- Considèrent toutes les exécutions possibles
- Calculs complexes mais sans impact sur l'exécution

Analyses dynamiques

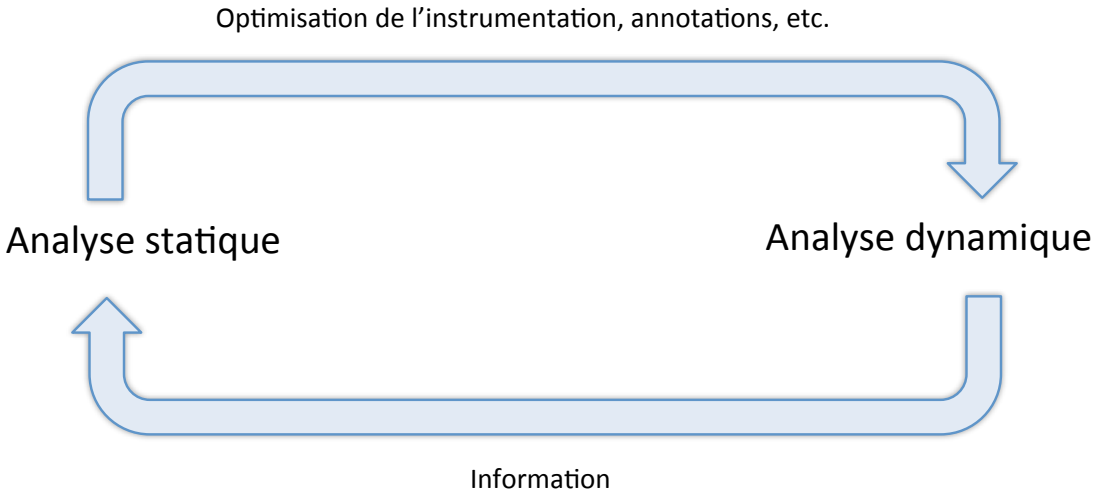
- Considèrent certaines exécutions concrètes, généralisation difficile
- Impact sur l'exécution proportionnel à la quantité d'information recueillie

Approximation du comportement

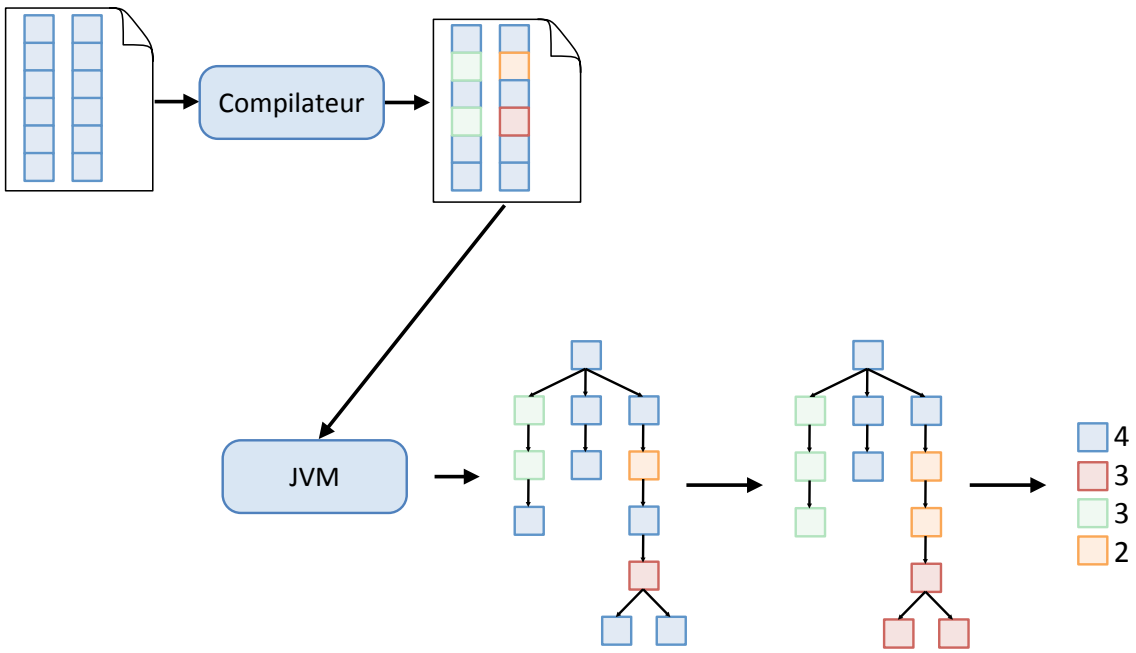


Analyses combinées

Combinaisons d'analyses



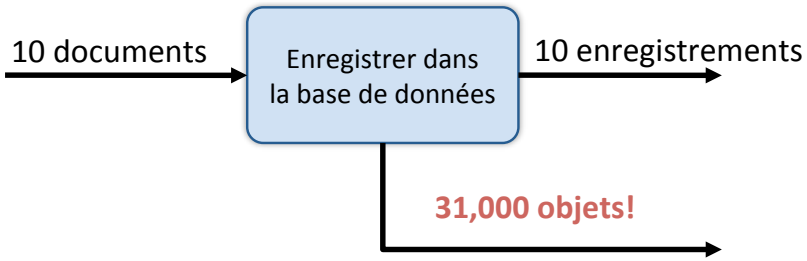
Statique → Dynamique



Dynamique → Statique

- Le sujet principal à partir d'ici
  - Permet d'augmenter l'analyse statique avec de l'information dynamique
    - Réflexion, classes chargées, méthodes natives, etc.
    - Fréquences, types créés, etc.
  - Permet de réduire le coût de l'analyse dynamique à l'exécution

Un problème de performance réel



- Les objets temporaires sont coûteux:
  - Allocation de la mémoire
  - Initialisation
  - Collection par le nettoyeur (ramasse-miettes, *garbage collector*)

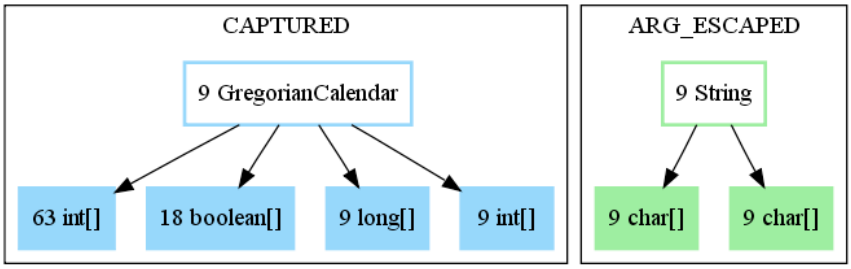
Quels sont ces objets?

Type	Objets créés
java.lang.Integer	859
java.lang.Object[]	678
java.util.AbstractList$Itr	457
Id	420
java.util.ArrayList	401
int[]	378
java.nio.HeapByteBuffer	352
java.util.HashMap$Entry[]	320
java.util.HashMap	299
AccessControlEntry	260
+ 238 autres	...

D'où proviennent-ils?

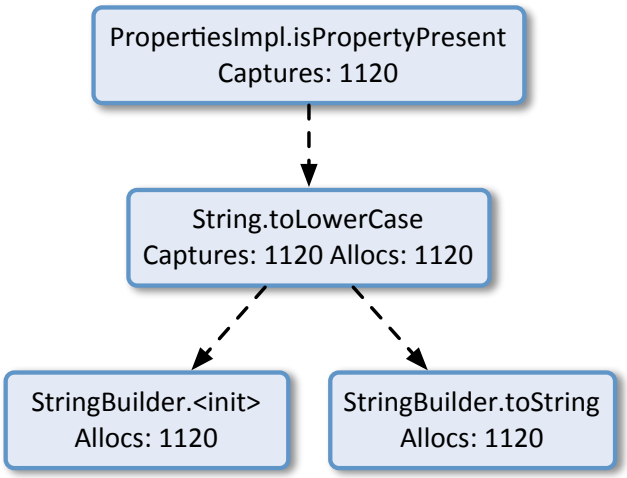
Méthode	Objets créés
a(int, Object, int, int)	1290
Format(String, …)	799
getDBResult()	650
getSecurityDescriptor()	440
applyPattern(String)	369
queueRollFwdRequest()	313
getObjectActualClassName()	160
loadFromBytes(byte[])	150
convertForCreate()	140
setupExecutionContextUsingAppend()	140
+ 394 autres	...

Example de résultat

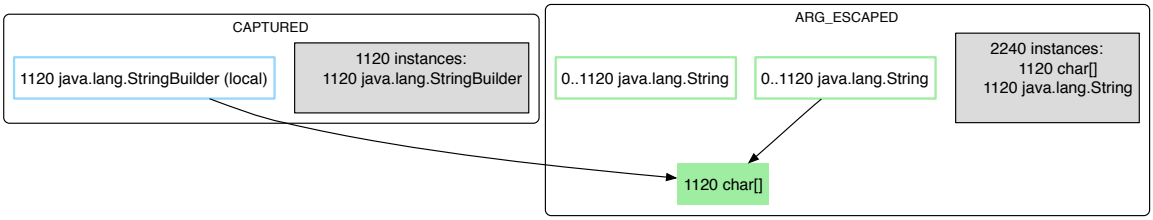


`DateSerializer.getValueAsString()`

isPropertyPresent

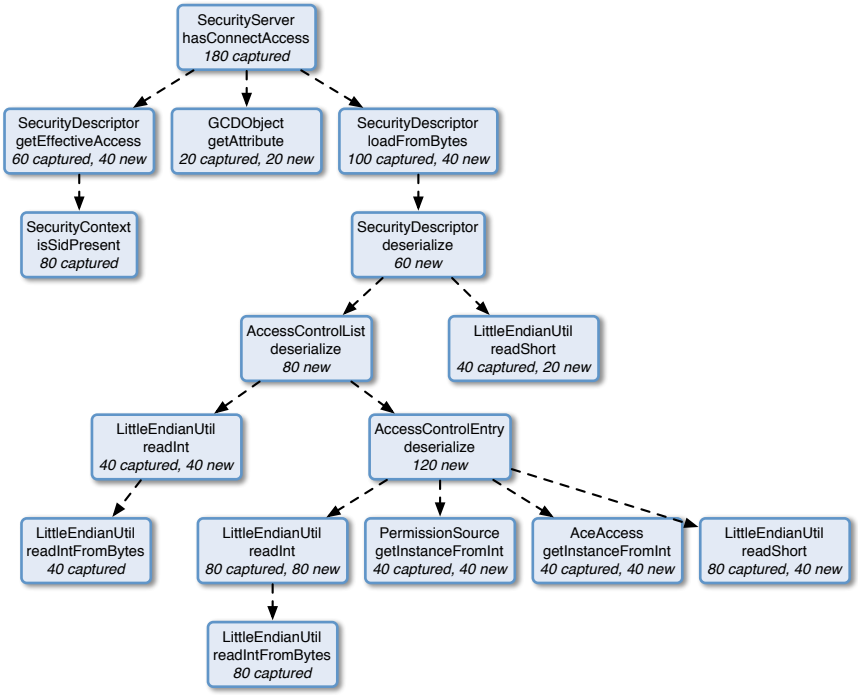


toLowerCase

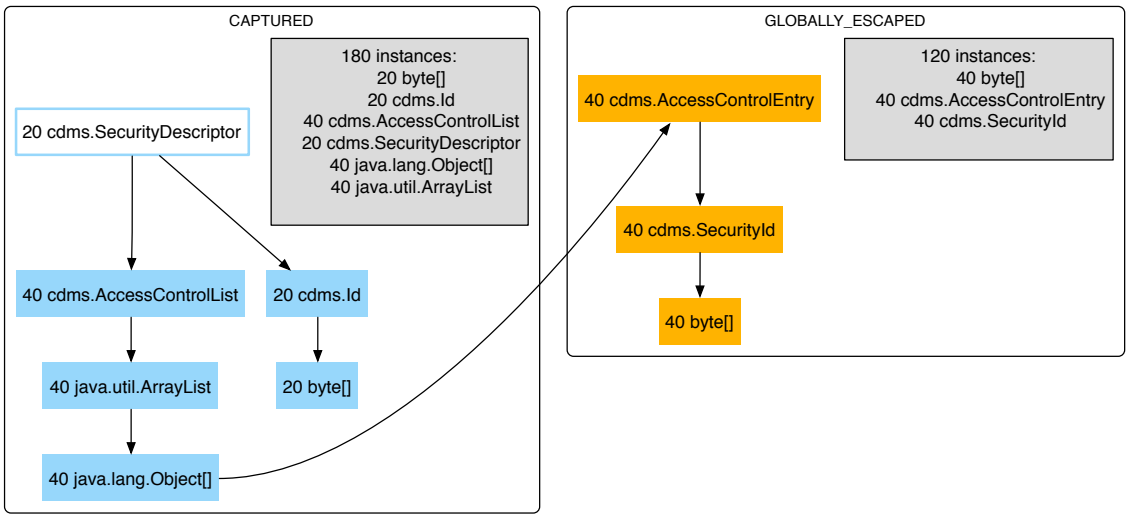


java.lang.String.toLowerCase(Ljava/util/Locale;)Ljava/lang/String; (context 7859)

hasConnectAccess



hasConnectAccess



cdms.SecurityServer.hasConnectAccess(Lcdms/Id;)Z (context 16430)

Visualisation des allocations