Génération de code : langages objets
Gestion mémoire
2 novembre 2011

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe
2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Introduction

- Les langages objets sont très populaires car ils offrent des mécanismes de structuration et d'encapsulation des données adaptés au développement de larges applications.
- Les mécanismes sophistiqués des langages objets (héritage, liaison tardive, . . .) doivent être pris en compte à la compilation.

Sommaire

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe
2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Classes

La déclaration d'une classe comporte

- des variables d'états (attributs ou champs de l'objet)
- des méthodes (correspondent à des procédures et fonctions)

- Les classes des langages objets jouent le rôle des types (en Java, il y a aussi des types de base et des tableaux).
- Au centre des langages objets, se trouve la notion d'héritage.
- Une classe D peut hériter d'une classe C.
  - Les variables d'état de C sont des variables d'état de D
  - les méthodes de C peuvent s'appliquer aux objets de la classe D.
  - Une méthode de C peut également être redéfinie dans D.
    Il y a alors surcharge du nom de la méthode : le même nom étant associé à plusieurs codes.
    Cette ambiguïté devra être résolue soit à la compilation soit à l'exécution.

Principes de la compilation

- Les variables et méthodes statiques se comportent comme des variables globales et des procédures ordinaires.
- Un objet correspond à un bloc mémoire comportant les différentes variables d'état.
  - Cette zone est allouée (dans le tas) au moment de la création de l'objet.
  - Chaque champs correspond à un décalage dans le bloc.
  - Le bloc comporte également l'indication de la classe de l'objet (typage dynamique)
  - Une valeur de type objet est l'adresse du bloc.
- Les méthodes se comportent comme des procédures : elles ont des arguments déclarés et éventuellement un résultat.
- Elles s'appliquent implicitement à un objet (référencé dans le corps de la fonction par `this` en Java).
  - On déclare une méthode `meth(x)` dans une classe *A*
  - On peut ecrire `o.meth(x)` si *o* dans (une sous-classe de) la classe *A*
  - Dans (une sous-classe de) la classe *A*, `meth(x)` s'interprète comme `this.meth(x)` (peut être résolu au niveau de l'analyse de portée)

Visibilité

- Les règles de visibilité des langages objet restreignent l'accès aux données.
- Les variables d'état privées introduites dans une classe ne seront visibles que dans les définitions de sous-classes.
- Les méthodes déclarées privées ne peuvent être utilisées en dehors de la classe.

Typage statique ou dynamique

Il y a deux manières de désambiguer le choix d'une méthode à appliquer :

- On détermine au typage la classe de l'objet et ceci permet de connaître statiquement la méthode à appliquer.
- C'est la classe de l'objet au moment de l'exécution qui détermine dynamiquement la méthode à sélectionner.

Typage dynamique

- Les classes des objets jouent un rôle important dans la compilation, aussi les objets manipulés comportent explicitement un pointeur vers le descripteur de leur classe.
- Le descripteur de classe comporte les informations utiles sur la classe, comme les labels associés aux méthodes définies dans la classe, les classes ancêtres, la taille des variables d'état ...

Héritage simple

- Dans le cas de l'héritage simple, une classe D hérite au plus d'une autre classe C.
- Organisation simple des variables d'états des objets de la classe D : extension des variables de la classe C
- Les méthodes compilées pour les objets de la classe C peuvent s'appliquer aux objets de la classe D.
- Un objet de la classe D qui hérite de C contient d'abord les variables de C puis celles de D.

Héritage simple : exemple

```
class A extends Object {var a:=0}
class B extends A {var b:=0 var c:=0}
class C extends A {var d:=0 }
class D extends B {var e:=0}
```

A:

	@A
a	0

B:

	@B
a	0
b	0
c	0

C:

	@C
a	0
d	0

D:

	@D
a	0
b	0
c	0
e	0

Chaque champs *var* a un décalage par rapport à l'adresse de base qui est connu à la compilation `dec`(*var*).

Accès aux champs d'un objet

Accès *obj.var*:

- Calculer la valeur de l'objet *o* (adresse d'un bloc dans le tas)
- Accéder au champs de l'objet (décalage connu statiquement)

$code(obj.var) = code(obj) \mid \textbf{lw}\ \$a0, \texttt{dec}(var)(\$a0)$

Mise à jour $obj.var = e$:

- Calculer la valeur de l'objet (dans $\$a0$)
- Sauvegarder cette valeur dans la pile
- Calculer la valeur de l'expression *e* (dans $\$a0$)
- Restaurer l'adresse de l'objet (dans $\$a1$)
- Stocker $\$a0$ dans le champs correspondant à *var*.

$code(obj.var = e) =$
$code(obj) \mid \texttt{pushr}\ \$a0 \mid code(e) \mid \texttt{popr}\ \$a1 \mid \textbf{sw}\ \$a0, \texttt{dec}(var)(\$a1)$

Création des objets

- Commandes explicites de création d'objet **new** C
  - Allocation d'un nouveau bloc sur le tas
  - Initialisation des champs
  - La taille du bloc est la taille des variables d'états + l'adresse du descripteur de la classe C.

    $\mathit{code}(\textbf{new}\ C) =$

    $\textbf{li}\ \$a0, \texttt{size}(C) \mid \textbf{li}\ \$v0, 9 \mid \textbf{syscall} \mid \textbf{lw}\ \$a1, \texttt{descr}(C) \mid \textbf{sw}\ a1, 0(\$a0)$
- L'initialisation se traite souvent comme une méthode particulière à laquelle on passe l'adresse de l'objet nouvellement alloué.

Héritage simple : méthodes

- méthodes statiques : compilées comme des appels à des procédures (label de la méthode connu à la compilation).
- méthodes dynamiques : l'adresse du code à appeler n'est pas connu statiquement, il sera calculé dynamiquement à partir du descripteur de classe.

```
class A extends Object {var x:=0 method f()}
class B extends A {method g()}
class C extends B {method g()}
class D extends C {var y:=0 method f()}
```

Pour compiler `c.f()` il faut

- Trouver le descripteur de classe de *c*
- Chercher le label associé au code de *f*
- Appeler le code associé

Exemple-suite

- Le programme engendré comporte quatre fonctions correspondant au code déclaré dans les classes :
  labels `A_f`, `B_g`, `C_g` et `D_f`.
- Les descripteurs de classe sont stockés dans la zone de données statiques à une adresse globale.
  Ils contiennent l'information sur les classes utiles à l'exécution :
  - l'adresse du descripteur de la classe père (pour implanter `instanceof`);
  - les labels des méthodes applicables.

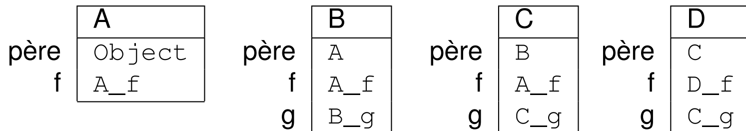
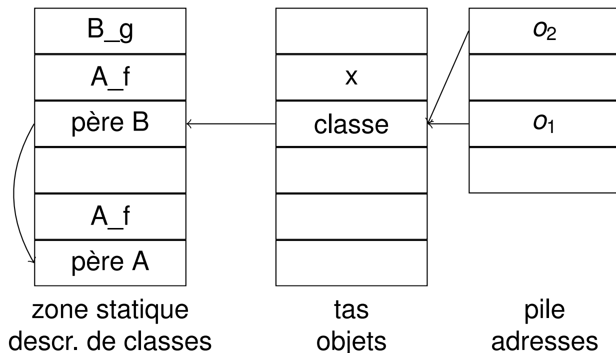


Table des symboles

- La table des symboles fournit une information statique, utilisée par le compilateur.
- Elle ne sert pas à l'exécution.
- Elle peut contenir pour chaque classe déclarée les informations suivantes :
  - adresse statique où sera stocké le descripteur de la classe,
  - classe père (pour le typage statique),
  - noms des attributs et décalages associés,
  - noms des méthodes applicables et décalage associé dans le descripteur de classe.

Compilation de l'appel de méthode

- On ne connaît plus statiquement l'adresse du code à appeler
- cette adresse se trouvera à une adresse statique en memoire



Compilation de l'appel de méthode

On décide de stocker l'objet dans $a0, l'argument de la méthode dans $a1 et la valeur de retour dans $v0.

code(o.f(a)) =

code(o)	valeur de l'objet (adresse) dans $a0
`pushr` $a0	sauvegarde dans la pile
code(a)	valeur de l'argument dans $a0
move $a1,$a0	valeur de l'argument dans $a1
`popr` $a0	restauration adresse de *o*
lw $a2 0($a0)	adresse du descripteur de classe de *o*
lw $a2 `decal`(f)($a2)	label associé à *f*
jalr $a2	appel de la méthode
move $a0,$v0	

Sommaire

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe
2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Héritage multiple

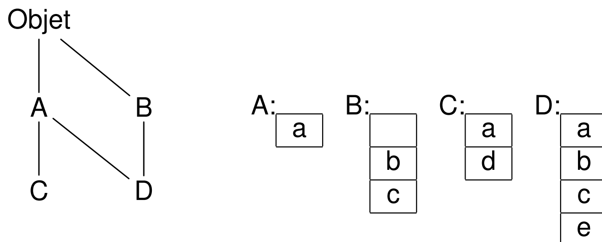
- Dans le cas de l'héritage multiple, une classe D peut hériter des objets des classes C_1 et C_2.
- On ne peut plus organiser simplement les variables de D, C_1 et C_2 pour que les procédures compilées pour C_i s'appliquent aux objets de D.

```
class A extends Object {var a:=0}
class B extends Object {var b:=0 var c:=0}
class C extends A {var d:=0}
class D extends A B {var e:=0}
```

Décalages des attributs

Trouver pour chaque variable d'état un décalage qui soit le même pour toutes les sous-classes.
Coloriage d'un graphe :

- sommet : variable d'état
- arête : variables qui coexistent dans une classe



Efficacité

- Si les objets étaient représentés en tenant compte de ces décalages alors il y aurait des trous dans la représentation.
- Seul le descripteur de classe a la forme adéquate, chaque place renvoie la position où la variable est stockée dans l'objet.
  - surcoût dans la manipulation des objets.
  - optimiser les accès au descripteur de classe.

Organisation des méthodes

- Pour organiser les méthodes il est également nécessaire de prendre en compte le graphe d'héritage et d'associer un décalage unique à chaque méthode.
- Ambiguïté possible sur le choix des méthodes.
- L'héritage multiple pose le problème de l'extensibilité (chargement dynamique de code).
- Il faut réorganiser le code ou bien être sûr d'associer un numéro unique.

Sommaire

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe
2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Tester l'appartenance à une classe

- Certains langages permettent de tester l'appartenance d'un objet à une classe (`instanceof`).
- Cela peut se faire en suivant les liens pères dans les descripteurs de classe (en supposant de l'héritage simple).
- On peut également garder dans chaque descripteur de classe un tableau des classes ancêtre.
  - Chaque classe a un niveau dans l'arbre d'héritage, `Object` de niveau 0.
  - Pour savoir si un objet x est une instance de la classe C de niveau j, il suffit de regarder la j-ème valeur dans le descripteur de x et vérifier que c'est C.

Optimisations

- Le coût le plus important dans un langage objet est l'appel de méthode.
- Il est essentiel d'analyser le code pour remplacer les appels dynamiques par des appels statiques.
- Si une méthode n'est pas redéfinie dans une sous-classe alors il n'y a qu'un seul code possible à exécuter (en l'absence de chargement dynamique).
- Une analyse statique peut déterminer qu'il n'y a qu'un seul type dynamique possible pour une expression donnée et donc connaître statiquement la méthode à appliquer.

```
x = new B();
y = new C();
z = if .. then y else x;
x.g(); y.g(); z.g();
```

Environnement d'exécution

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

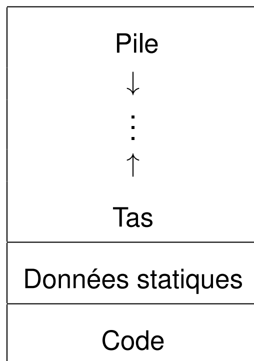
Sommaire

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Organisation de la mémoire

Chaque programme s'exécute dans une portion de la mémoire attribuée par le système d'exploitation.



Principe

- Les nouvelles valeurs calculées au cours de l'exécution sont souvent allouées dans les tableaux d'activation sur la pile.
- Ce principe fonctionne si ce qui est alloué au cours de l'appel d'une procédure n'est plus accessible à la fin de la procédure.
- Ce principe ne permet pas de couvrir toutes les constructions de programme.
  - Lorsqu'une méthode crée dynamiquement des objets
  - Lorsqu'une fonction f renvoie en valeur une autre fonction, celle-ci peut faire référence à des variables locales de f, allouées dans son tableau d'activation qui disparait à la sortie de f

Représentation des données structurées

- La manipulation dans la pile de données de taille variable est complexe.
  - Copier des valeurs est coûteux
- Manipuler des objets complexes via leur adresse est plus simple.
- Certaines instructions créent la structure:
  Exemples:
  - Tableaux en CAML :
    Création explicite `[|1;2;3|]` ou via la commande `Array.create`
  - en C : initialisation des variables, utilisation de `malloc`
  - en Java : instruction `new`
- La valeur est ensuite manipulée par son adresse.
- Si une fonction renvoie une valeur structurée sous la forme d'une adresse, il faut que la structure elle-même soit allouée dans une zone persistante de la mémoire.

Allocation dynamique

- De nouveaux espaces mémoires créés dynamiquement au cours du programme; accessibles par le programme sans forcément être directement liés à une variable.
- La durée de vie de ces objets n'est pas limitée à la durée de vie des procédures : réservation de l'espace dans le tas.
- On distingue :
  - les données directes telles que les entiers qui seront allouées et manipulées dans la pile;
  - les données indirectes dont la manipulation se fera par l'intermédiaire d'une adresse dans la mémoire.

Allocation dynamique-remarques

- La manipulation d'objets via des adresses introduit une indirection (coûteuse) mais est plus uniforme.
- Elle est nécessaire lorsque le langage contient des fonctions polymorphes (les fonctions peuvent manipuler des données de taille variable à l'exécution).

Sommaire

1. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe
2. Environnement d'exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Allocation/desallocation explicite

- Les langages comme Pascal ou C offrent des primitives (`new`/`dispose` en Pascal, `malloc`/`free` en C) pour allouer ou désallouer des parties de la mémoire.
- Le tas est organisé en une suite de blocs libres chaînés entre eux. Chaque bloc est formé d'une suite d'octets consécutifs en mémoire.
  - le bloc comporte une entête qui contient la taille du bloc en nombre d'octets.
  - les blocs libres sont chainés entre eux.
- Les blocs libres peuvent être organisés en plusieurs listes, par exemple en ayant des espaces réservés pour l'allocation de données de petite taille.

Allocation explicite

- Pour allouer de la mémoire, on cherche un bloc de taille suffisante en parcourant la liste des blocs libres.
- Si on ne trouve pas de bloc adéquat, il faut demander au système un morceau de mémoire supplémentaire.
- Si on trouve un bloc trop grand, on en réserve la partie nécéssaire et on garde le reste dans la liste des blocs libres.
- Il peut être judicieux de choisir le bloc le plus petit qui convient.
- Garder la localité des données est aussi utile.

Libération explicite

- Le morceau libéré est ajouté à la liste des blocs libres.
- Il est utile de fusionner des blocs libres consécutifs pour récupérer des espaces plus grands.
- On peut garder sur chaque bloc, en tête et en fin, un bit indiquant s'ils sont libres + la taille du bloc.
- Il suffit de mettre à jour localement si l'un des blocs adjacents est libre.

	0	20	...	20	0	1	10	...	10	1	1	12	...	12	1

	0	30	...	...	...	30	0	1	12	...	12	1

Allocation/desallocation explicite

- L'allocation/ désallocation explicite rend la programmation plus lourde.
- On risque d'utiliser inutilement la place par des données inaccessibles.
- On risque au contraire de faire disparaître des données accessibles et de provoquer des erreurs difficilement détectables ensuite suivant la manière dont cet espace est réalloué.
- De telles erreurs posent des problèmes de sécurité.

Sommaire

1 Compilation d’un langage objet
- Représentation des classes et des objets
- Héritage multiple
- Appartenance à une classe

2 Environnement d’exécution
- Organisation de la mémoire
- Allocation explicite
- **Allocation implicite**
- Mécanismes de récupération de la mémoire

Allocation/desallocation implicite

- Des langages tels que LISP, ML ou JAVA ont choisi la voie de l'allocation dynamique implicite.
- L'exécution va être amenée à allouer de la mémoire
  - `new` en Java
  - constructeurs, fonctions comme résultat en Caml
- L'environnement de programmation s'occupe également de récupérer la mémoire.
- C'est le mécanisme de Garbage collector (ramasse-miette, glaneur de cellules).

Remarque

- Le GC s'appuie sur le typage fort des langages qui restreint statiquement l'accès à la mémoire (via des variables déclarées ou bien des champs de ces variables).
- Dans un langage comme C, l'arithmétique de pointeur autorise a priori l'accès à n'importe quelle partie de la mémoire.
- Des outils d'analyse de la mémoire "faibles" permettent de détecter des erreur potentielles.

Sommaire

1. Compilation d’un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

2. Environnement d’exécution
   - Organisation de la mémoire
   - Allocation explicite
   - Allocation implicite
   - Mécanismes de récupération de la mémoire

Compteur de référence

Chaque cellule dynamiquement allouée comporte un compteur indiquant combien de pointeurs peuvent y accéder.

- `new`(p) la cellule a un compteur initialisé à 1
- Une affectation entre pointeurs a la forme `t = u`
  - `t` s'évalue en une valeur gauche x
  - x initialement a pour valeur droite une cellule m
  - `u` s'évalue en une valeur droite qui est une cellule n
- On décrémente le compteur de m et on incrémente celui de n

Compteur de référence-récupération

- Les cellules dont le compteur est zéro peuvent être récupérées.
- Il est possible que deux cellules pointent mutuellement chacune sur l'autre et gardent des compteurs à 1 (même si elles ne sont plus accessibles).
- Cette méthode est coûteuse en temps de calcul.
- La récupération de la mémoire se fait au fur et à mesure.

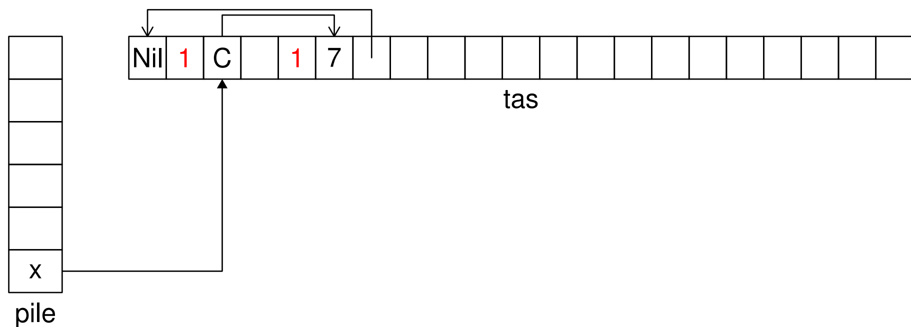
Compteur de référence : exemple

```
type list = Nil | Cons of cellule
and cellule = {value : int; mutable next : list}

let x = Cons {value=7; next=Nil}
in let y = Cons {value=9; next=x}
in let t = Cons {value=10; next=y}
in let Cons z = x in z.next<-y;;
```

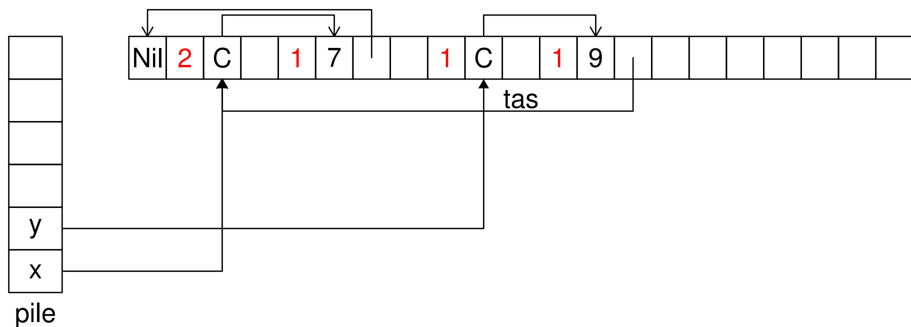
Etape 1

```
let x = Cons {value=7; next=Nil}
```



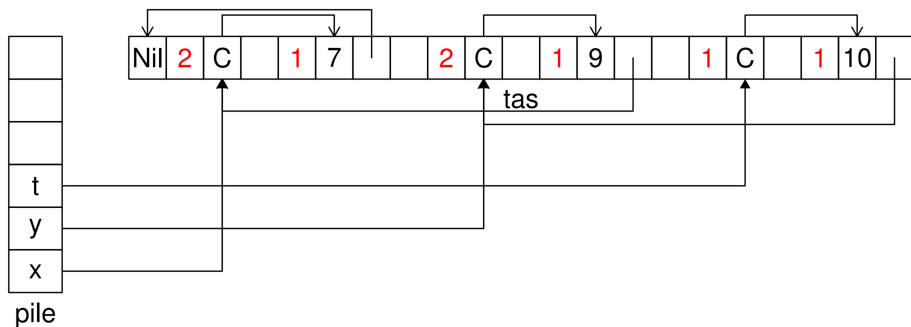
Etape 2

```
let y = Cons {value=9; next=x}
```



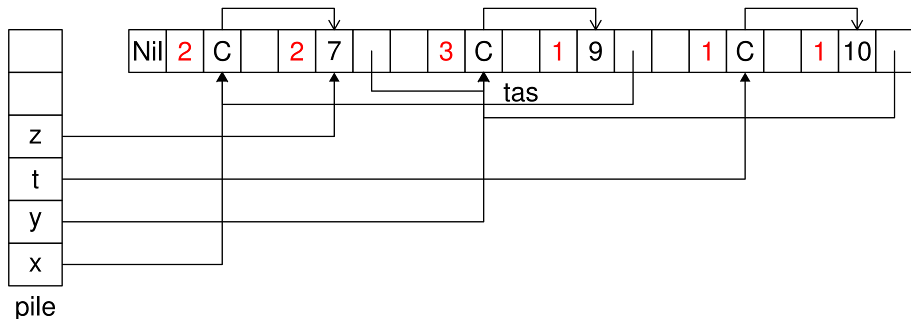
Etape 3

```
let t = Cons {value=10; next = y}
```



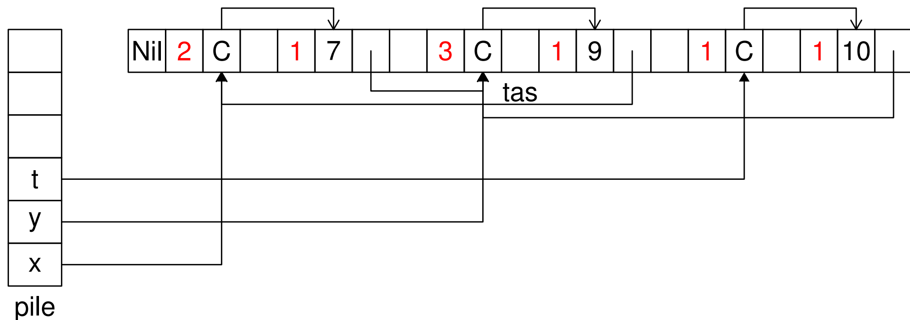
Etape 4

```
let Cons z = x in z.next<-y
```



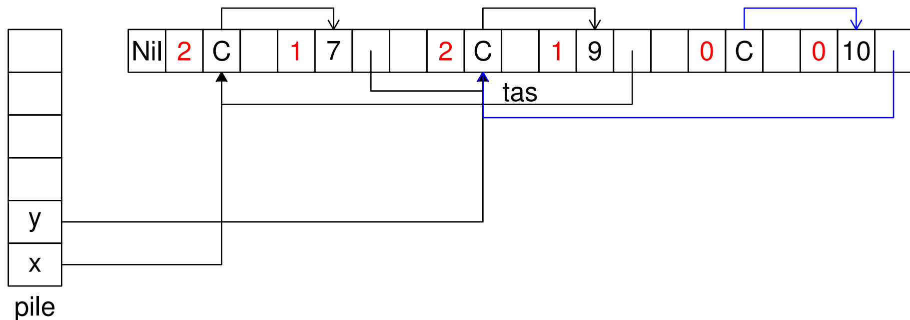
Etape 5

Libération de z



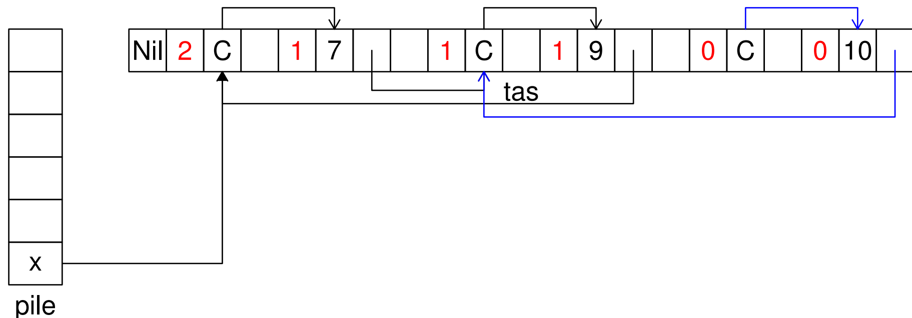
Etape 6

Libération de *t*



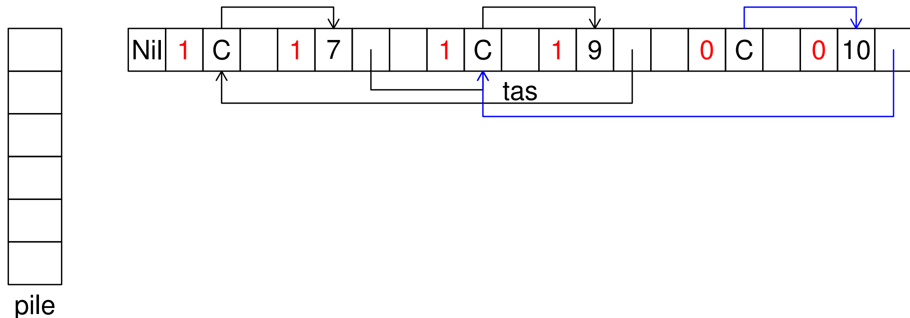
Etape 7

Libération de *y*



Etape 8

Libération de x



Il reste une structure circulaire non récupérable.

Marquer et Balayer

- À partir des variables de la pile (données accessibles au programme) on marque toutes les cellules du tas utilisables de manière récursive.
- Lorsque ce procédé est terminé, une deuxième passe libère les blocs inaccessibles.
- Algorithme (recherche d'une composante connexe)
  - On distingue les nœuds non visités, les nœuds visités dont les fils n'ont pas été visités, et les nœuds complètement examinés.
  - Initialement tous les nœuds sont non visités.
  - On marque les nœuds directement accessibles au programme comme partiellement visités et on les conserve dans une liste.
  - Tant qu'il reste un nœud partiellement visité on le marque comme visité et on le retire de la liste, s'il a des fils non encore visités, on les marque partiellement visités et on les ajoute à la liste.

Remarques

- Les données accessibles au programme peuvent aussi venir des registres.
- Cette récupération de mémoire est coûteuse en temps (problème pour les applications temps réel).
- Autre inconvénient : fragmentation de la mémoire.
- Le procédé récursif pour marquer la mémoire peut provoquer un dépassement de capacité mémoire.
  - Algorithme de Shorr-Waite : inversion des liens mémoires pour mémoriser les zones à parcourir.

Stopper et Copier

- Pour éviter les problèmes de fragmentation on peut décider de toujours allouer linéairement des données.
- Lorsque la zone est pleine on s'arrête et on recopie toute la partie utile soit vers la base du tas soit vers une seconde zone de manière continue.
- On a toujours une zone libre linéaire (plutôt qu'un chainage de blocs libres)
- Cette méthode ajoute un coût supplémentaire de recopie et modification des pointeurs.

Méthodes mixtes

- Les GC modernes utilisent un compromis entre ces différentes méthodes.
- Ocaml utilise un GC mis au point par Damien Doligez.
- L'espace mémoire est séparé en deux zones :
  - l'une dite vieille organisée à l'aide d'une liste de places accessibles. Cette zone pourra être étendue dynamiquement si nécessaire
  - l'autre dite jeune organisée comme un tableau linéaire de taille fixe.

Méthode

- Allocation: Les cellules sont allouées linéairement dans le tableau.
- gc mineur: Lorsque le tableau est plein on stoppe et on copie dans la vieille zone les objets utiles:
  - accessibles à partir des variables de la pile, des registres ou des objets de la vieille zone.
- gc majeur: on procède à un marquage/balayage incrémental de la vieille mémoire.

GC dans CAML

- Pour marquer les objets on utilise un coloriage des entêtes de blocs.
  - Nœud blanc : pas encore visité
  - Nœud gris : visité mais pas leurs fils immédiats
  - Nœud noir : visité ainsi que leurs fils immédiats.
- On garde une pile des nœuds gris.
  Lorsqu'il ne reste plus de noeuds gris alors on a fini le marquage.
- Les nœuds qui sont blancs après le marquage peuvent être rendus pour la liste libre de la vieille génération.
- Afin de gérer le GC, la représentation de CAML réserve un bit sur chaque mot pour distinguer les adresses (que le GC doit suivre) des entiers.
- Le tas est organisé en zones correspondant aux différentes structures allouées (tableau, type de données structuré, clôture,. . .).
- Dans l'entête de chaque bloc on réserve deux bits pour le marquage (blanc,gris,noir) du GC.

Remarques sur la gestion de la mémoire

- La bonne gestion de la mémoire a une influence sur les performances du programmes.
- Problème difficile qui dépend à la fois du langage de programmation et des applications considérées.