Production de code, fonctions et objets

Martin Odersky

version 1.5

Plan du cours

Variables locales et paramètres

Question : où stocker les variables locales et les paramètres de fonctions ?

Premiére idée : à des adresses fixes en mémoire (allocation statique, à la Fortran).

- Malheureusement, cela interdit certaines formes de récursivité, car chaque fonction ne possède qu'une copie de ses variables locales et paramètres.

Meilleure idée : allouer la mémoire pour les variables et paramètres dynamiquement, au moment de l'exécution.

Cette solution est utilisée par la quasi-totalité des langages actuels.

Question : quelle stratégie d'allocation utiliser pour ces variables et paramètres ?

A l'exécution, les appels de fonction forment une pile :

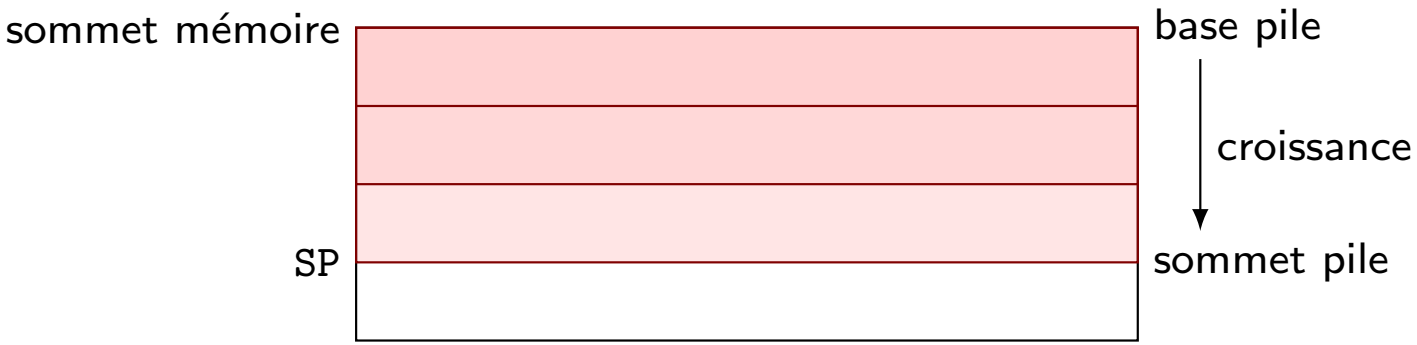
- la fonction en cours d'exécution se trouve toujours au sommet de la pile,
- lorsqu'une fonction en appelle une autre, cette dernière est placée au sommet de la pile, et s'exécute en totalité avant de retourner à la fonction qui l'a appelée (c'est-à-dire d'être dépilée).

Idée : utiliser une pile pour l'allocation mémoire des variables locales et paramètres.

La pile

Zone mémoire dans laquelle sont allouées toutes les données dont la durée de vie est incluse dans celle de la fonction qui les définit.

- Le sommet de la pile est repéré par le pointeur de pile (*stack pointer* ou *SP*), habituellement stocké dans un registre réservé à cet effet.
- Attention : en mémoire, la pile croît souvent vers le bas, donc son sommet se trouve à une adresse inférieure à sa base.



L'architecture DLX posséde deux instructions pour faciliter la gestion de la pile :

`PSH` R_a R_b `ic` R_b = R_b - `ic`; `MEM[`R_b`]` = R_a

`POP` R_a R_b `ic` R_a = `MEM[`R_b`]`; R_b = R_b + `ic`

- `PSH` alloue de la place sur la pile puis y copie une valeur.
- `POP` lit une valeur de la pile, puis libère de la place.
- Le registre R_b est le pointeur de pile. Par convention, on utilise `R30`, mais n'importe quel autre registre ferait l'affaire.

Exemple : Appel de fonction

La fonction `f(15, 16)` devient :

```
ADDI 1 0  15
PSH  1 SP 4
ADDI 1 0  16
PSH  1 SP 4
JSR  f
```

Blocs d'activation

Chaque fonction ne manipule directement qu'une petite zone au sommet de la pile, qui lui est réservée.

Cette zone se nomme bloc d'activation (*stack frame*).

- Lorsqu'une fonction commence son exécution, son bloc d'activation ne contient que les paramètres qui lui ont été passés par la fonction appelante.
- Au fur et à mesure de son exécution, la fonction peut faire grandir son bloc d'activation pour y stocker différentes données : variables locales, copies de registres à sauvegarder, paramètres pour fonctions appelées, etc.

Chaque fonction conserve un pointeur vers la base de son bloc d'activation, habituellement dans un registre réservé à cet effet et nommé FP (pour *frame pointer*).

Ce pointeur permet d'accéder aux paramètres et aux variables locales, qui se trouvent à une distance fixe par rapport à FP.

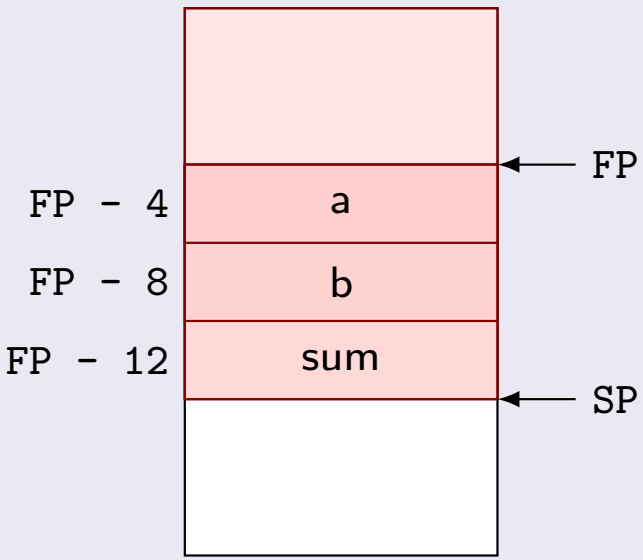
Exemple : Bloc d'activation de la méthode add

La fonction :

```
def add(a: Int, b: Int): Int {
  var sum: Int = a + b;
  return sum
}
```

devient :

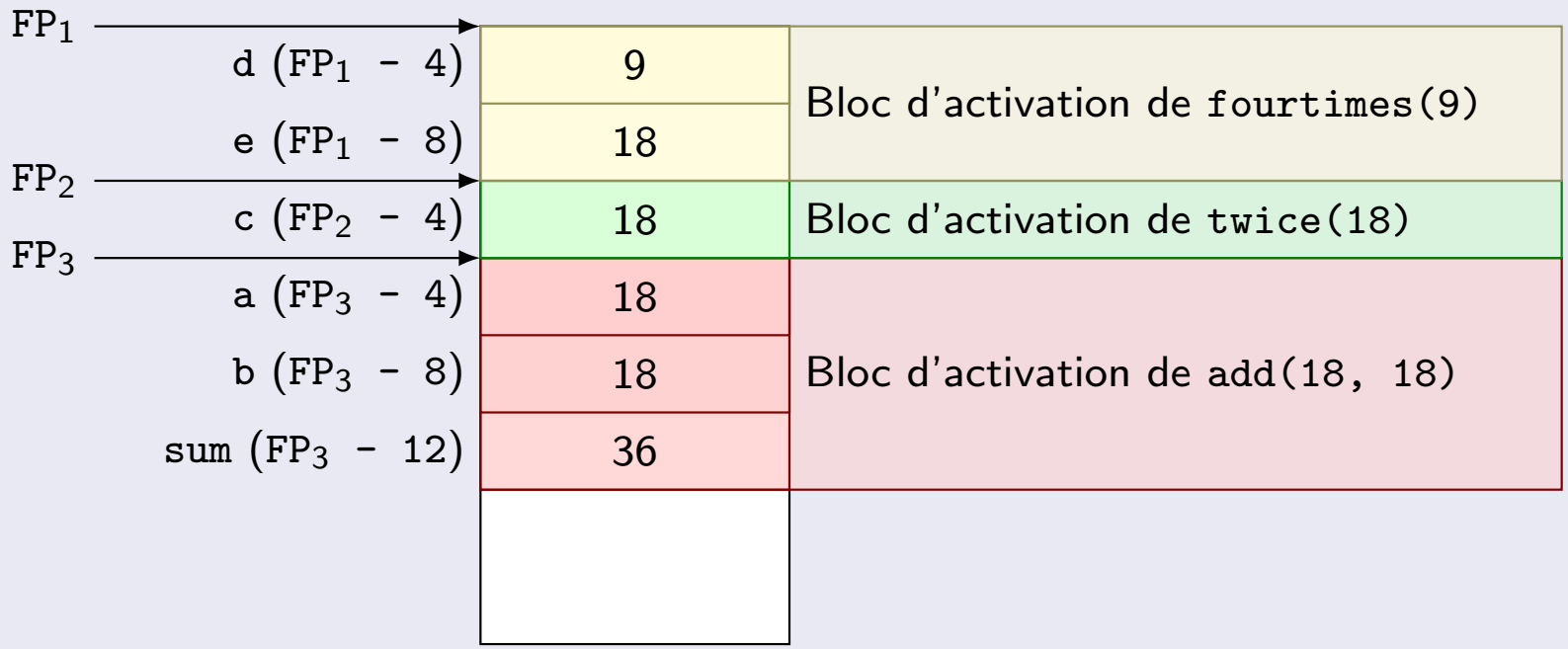
```
LDW R1 FP -4   // charge a
LDW R2 FP -8   // charge b
ADD R1 R1 R2   // calcule a+b
PSH R1 SP 4    // sauve sum
LDW R1 FP -12  // charge sum
```



Exemple : Bloc d'activation et appels de fonctions

```
def add(a: Int, b: Int): Int { var sum: Int = a + b; ★ return sum }
def twice(c: Int): Int { return add(c, c) }
def fourtimes(d: Int): Int { var e: Int = d + d; return twice(e) }
... fourtimes(9) ...
```

Et voici la pile à l'endroit ★ :



Utiliser un registre pour stocker `FP` a plusieurs inconvénients :

- un registre de moins disponible pour utilisation générale,
- il faut sauvegarder/restaurer `FP` lors d'appels de fonction.

Question : est-il possible de faire sans ?

- Invariant : `frame_size = FP - SP` ou en d'autres termes : `FP = SP + frame_size`

Conclusion : si `frame_size` est connu à tout moment lors de la compilation, on peut se passer de `FP` !

C'est ce qu'on fait dans le projet : la classe `Code` fournit les méthodes `incFrameSize`, `decFrameSize` et `getFrameSize` pour mémoriser la taille courante du bloc d'activation.

A utiliser sans faute chaque fois qu'on produit des instructions qui modifient la taille du bloc d'activation (par ex. `PSH` et `POP`).

Passage de paramètres par registres

Les paramètres d'une fonction peuvent aussi être passés dans les registres plutôt que sur la pile.

Avantages du passage par registres :

- code potentiellement plus compact, car l'appelant n'a pas besoin de placer les paramètres sur la pile (important pour les processeurs sans instruction `PSH`),
- code potentiellement plus rapide, car les procédures feuille (qui ne contiennent pas d'appels) n'ont pas besoin de sauvegarder les paramétres,
- aucune correction à apporter pour accéder aux données stockées sur la pile.

Avantages du passage sur la pile :

- nécessite moins de registres, ce qui peut être important lorsqu'il y en a peu, comme sur le *Pentium*,
- lors d'appels imbriqués, les paramètres ne doivent pas être sauvegardés et restaurés de manière répétée ; par exemple, si l'on passe les arguments dans les registres, lors de l'appel `f(x, g(y, h(z)))`, le paramètre `x` est chargé en premier, puis doit être sauvegardé durant l'appel à `g` ; de même `y` doit être chargé puis sauvegardé lors de l'appel à `h` ; cela n'est pas nécessaire si l'on passe les arguments sur la pile.

En bref, le passage par registres est meilleur lorsqu'on a suffisamment de registres à disposition.

Valeur de retour

Toute fonction doit communiquer sa valeur de retour à l'appelant.

Pour les résultats de taille inférieure ou égale à un mot, on peut utiliser un registre.

Pour les résultats plus grands, il y a deux possibilités :

1. sur la pile, ce qui complique la séquence de sortie,
2. l'appelant alloue la place dans sa portion de pile et passe un pointeur vers cet espace à l'appelé, qui y stocke son résultat.

- Lorsqu'on passe des objets en paramètre, on passe en fait un pointeur vers la première cellule de l'objet.
- Lorsqu'on retourne un objet comme résultat, on retourne également un pointeur.

En Drei, une fonction peut retourner un objet qu'elle a créé localement au moyen de la construction "new".

Exemple : Fonction et objets

```
def prependTwice(elem: List, lst: List): List =
  new List(elem, new List(elem, lst));
```

Il n'est pas possible d'allouer les objets sur la pile : ils ne survivraient pas à la fonction qui les a créés (ou alors il faudrait les copier au retour, ce qui est coûteux).

Ceci explique l'usage d'un tas et d'un ramasse-miettes.

Adresse de retour

Sur certains processeurs, la pile est aussi utilisée pour mémoriser l'adresse de retour :

- au moment de l'appel d'une fonction, l'adresse de retour est placée sur la pile,
- au moment du retour, elle est simplement dépilée.

Le processeur DLX fonctionne différemment :

- au moment d'un appel, l'instruction `JSR` place l'adresse de retour dans le registre `R31`, appelé aussi `LNK` (*link*),
- au moment du retour, l'instruction `RET` prend en argument le registre contenant l'adresse de retour (`LNK`).

Les fonctions qui appellent d'autres fonctions doivent donc sauvegarder sur la pile le registre `LNK` au début de leur exécution et le restaurer juste avant l'instruction `RET`.

Appel de fonctions “inconnues”

Dans un appel de fonction Drei, la fonction peut être une expression quelconque :
`{ var e: Fun = ...; return e}.apply(5)`.

Cela rend l'utilisation de l'instruction `JSR` impossible, car elle admet que l'adresse de la fonction appelée est connue au moment de la compilation...

Il faut donc émuler le comportement de l'instruction `JSR` :

- placer l'adresse de retour dans le registre `LNK`,
- sauter à l'adresse de la fonction (qui est connue seulement au moment de l'exécution).

L'adresse de retour est facile à calculer grâce à la fonction `pc()` de la classe `Code`.

Le saut se fait au moyen de l'instruction ... `RET` !

Sauvegarde des registres

Lorsqu'une fonction en appelle une autre, il est probable que la fonction appelée modifie le contenu de registres dont la fonction appelante aura besoin par la suite.

Il faut donc sauvegarder le contenu des registres au moment de l'appel de fonction.

La sauvegarde peut s'effectuer sur la pile avec deux stratégies :

1. sauvegarde par la fonction appelante,
2. sauvegarde par la fonction appelée.

La sauvegarde et la restauration des registres lors d'appels peuvent être faite par l'appelant (*caller-save*) ou par l'appelé (*callee-save*).

Sauvegarde par l'appelant :

- avant un appel, sauvegarder tous les registres dont le contenu sera nécessaire ensuite (`SP` excepté),
- aprés un appel, les restaurer.

Sauvegarde par l'appelé :

- au début d'une fonction, sauvegarder tous les registres qui seront modifiés dans le corps (`RES` et `SP` exceptés),
- à la fin d'une fonction, les restaurer.

Si les variables locales sont stockées dans des registres, la sauvegarde par l'appelant risque de sauvegarder et restaurer beaucoup de registres à chaque appel.

- On peut faire mieux en réservant un certain nombre de registres pour les variables locales, registres qui sont sauvegardés par l'appelé.
- Chaque fonction qui désire utiliser des registres pour y stocker ses variables locales doit sauvegarder et restaurer leur valeur.

La plupart des compilateurs utilisent la technique de la sauvegarde par l'appelant, ou une combinaison de sauvegarde par l'appelant et l'appelé.

Dans le projet, on emploiera la sauvegarde par l'appelant, sauf pour le registre LNK qui est sauvegardé par l'appelé.

Résumé : gestion des fonctions en Drei

Au début d'une fonction (prologue) :

- sauvegarder `LNK` sur la pile (optionnel pour les fonctions feuilles).

A la fin d'une fonction (épilogue) :

- restaurer `LNK`,
- libérer l'espace utilisé par les arguments (bloc d'activation).

Avant un appel de fonction :

- sauvegarder les registres dont le contenu sera nécessaire après l'appel (sauf `LNK`).

Aprés un appel de fonction :

- restaurer les registres précédemment sauvegardés.

Méthodes de *dispatching* OO

La première partie du cours considérait toujours que l'adresse de la méthode appelée était connue.

Ceci permet d'utiliser l'instruction `JSR`.

Les langages OO supportent la liaison dynamique : un appel de méthode invoque un code qui dépend du type dynamique de l'objet receveur.

A cause du sous-typage, le type dynamique peut différer du type statique connu à la compilation.

Probléme : Comment réaliser le *dispatching* dynamique efficacement ?

Héritage simple

Le dispatching dynamique est relativement simple à mettre en oeuvre dans le cas de l'héritage simple :

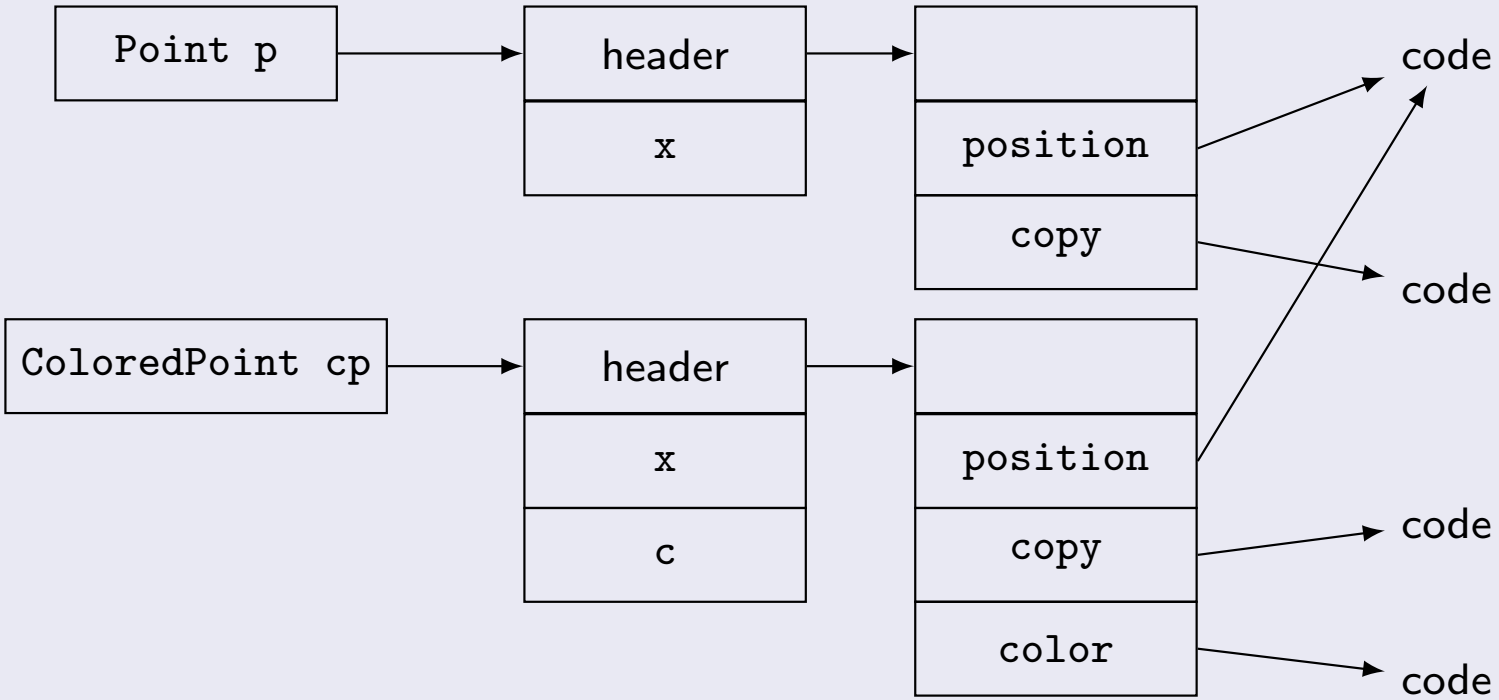
- chaque classe a au plus une super-classe.

Exemple : Dispatching dynamique en Drei

```
class Point {
  val x: Int;
  def position(): Int { return x }
  def copy(delta: Int): Point { return new Point(position() + delta) }
}
class ColoredPoint extends Point {
  val c: Color;
  def color(): Color { return c }
  def copy(delta: Int): ColoredPoint {
    return new ColoredPoint(x + delta, c) }
}
```

Continue sur le transparent suivant...

Voici l'infrastructure de dispatching pour des instances des classes `Point` et `ColoredPoint` ci-dessous :



Un appel à `p.copy()` deviendra ainsi `p.header.copy()`.

Tables virtuelles de méthodes

Une table virtuelle de méthodes (*virtual method table* ou VMT) contient des entrées qui pointent vers les adresse de départ de toutes les méthodes de la classe.

- Si la classe `A` hérite de la classe `B`, les premières entrées de la VMT de `A` sont toutes les entrées de la VMT de `B`.
- Mais `A` peut faire suivre ces entrées par des entrées supplémentaires.

Chaque objet contient comme premiére entrée un pointeur vers la VMT de sa classe.

Les VMTs sont générées comme du code :

- Dans un premier temps, l'espace des VMTs est rempli de valeurs bidon ;
  pour chaque méthode de chaque classe on émet une opération pour "prendre la place".
- Lorsque le code des méthode a été généré, on émet pour chaque méthode une opération `STW` qui remplace la valeur bidon dans la VMT par la vrai adresse du code de la méthode.

Création d'objets

Considérons le code `new C` ; comment peut-on implanter cela ?

- On calcule la taille en octets `|C|` d'un objet de la classe `C`.
- On alloue un espace mémoire de la taille `|C|` en gardant un pointeur vers ce bloc dans R_d :

```
ORIU    Rt R0 |C|
SYSCALL Rd Rt SYS_GC_ALLOC
```

- On sauvegarde le pointeur vers la VMT (à l'adresse `#C`) au sommet du bloc :

```
ORIU    Rt R0 #C
STW     Rd Rt 0
```

Ici, R_t est un registre temporaire.

Obtenir la valeur `this`

Dans les langages OO, la valeur `this` réfère l'objet courant. Comment peut-on implanter ceci ?

Idée : on passe `this` comme paramètre supplémentaire à chaque méthode appelée.

- De cette façon, `x.m(y)` veut réellement dire `x.m(x, y)`.
- `this` est simplement le premier paramètre de la méthode courante.

Dispatching OO avec DLX

Comment peut-on implanter le *dispatching* dynamique sur DLX ?

Problem : `JSR` saute vers une adresse statique.

Idée : On calcule l'adresse de la méthode dans un registre, puis on `RET` vers cette adresse.

Le code `p.copy(delta)` devient :

```
        ...                 // sauvegarde les registres
        LDW  R2  SP #p      // adresse de la valeur de p (this)
        PSH  R2  SP 4       // pousse this sur la pile
        LDW  R2  R2 0       // adresse de la VMT de p
        LDW  R1  R2 #copy   // adresse de la methode copy
        LDW  R2  SP #delta  // charge le premier parametre
        PSH  R2  SP 4       // pousse le premier parametre sur la pile
        ORIU LNK R0 #return // sauvegarde l'adresse de retour
        RET  R1             // on y va !
return: ...
```

L'adresse de retour est toujours : `pc() + WORD_SIZE * 2`.

Exemple : Compiler la fonction factorielle

```
class C {
  def fact: Int (
    // C this,
    x: Int
  ) {
    return
      if (x == 0)
        1
      else {
        var y: Int =
          this.
          fact(x - 1);
        return x * y
      }
  }
}
```

```
fact: PSH  LNK SP 4     // sauvegarde LNK
      LDW  R1  SP 4     // charge x
      BNE  R1     #else // si x est nul
      ORIU R1  R0 1     // retourne 1
      BEQ  R0     #end  // et termine
else: LDW  R2  SP 8     // sinon, charge this
      PSH  R2  SP 4     // comme 1er param
      LDW  R2  R2 0     // la VMT donne
      LDW  R1  R2 0     // l'adresse de fact
      LDW  R2  SP 8     // charge x
      ORIU R3  R0 1     // charge 1
      SUB  R2  R2 R3    // calcule x-1
      PSH  R2  SP 4     // empile x-1
      ORIU LNK R0 #ret  // sauve le retour
      RET  R1           // et execute fact
ret:  PSH  R1  SP 4     // definit y
      LDW  R1  SP 8     // charge x
      LDW  R2  SP 0     // charge y
      MUL  R1  R1 R2    // calcule x*y
      ADDI SP  SP 4     // libere y
end:  POP  LNK SP 12    // restaure LNK
      RET  LNK          // et retourne
```

Traiter le cas de `null`

Probléme : La destination `expr` (*receiver*) d'un appel de méthode `expr.m(...)` peut être `null`.

Exercice

Que se passe-t-il dans ce cas ?

Solution : On teste avant chaque appel de méthode et sélection de champ que la destination n'est pas nulle :

```
    LDW     R1  SP  #expr     // on charge la destination
    BNE     R1      #ok       // on saute si elle est non-nulle
    ORIU    R1  R0  1         // on définit le code d'erreur
    SYSCALL R1  R0  SYS_EXIT // et on termine
ok: ...
```

Code d'un programme complet

Un programme Drei complet est composé :

- des VMTs des classes,
- du code des méthodes et
- du code de l'expression principale.

Le code pour les fonctions est produit au fur et à mesure que les définitions sont examinées. Il faut donc insérer un saut depuis le début du programme vers l'expression principale.

De plus, au début d'un programme il faut initialiser le glaneur de cellules, le pointeur de pile et les VMTs.

Voici la structure générale du code d'un programme Drei :