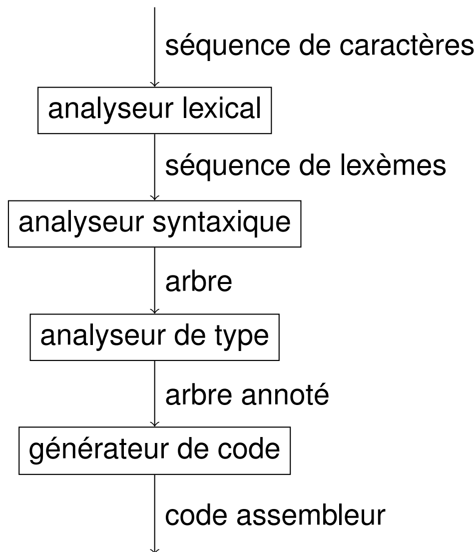
Projet du premier semestre L3

Sébastien Briais

ENS Lyon

4 janvier 2008

Structure du compilateur



L'architecture DLX

Le processeur cible est un processeur fictif qui est une version simplifié du processeur DLX.

- Processeur RISC
- 32 registres de 32 bits chacun : R0—R31
- R0 contient toujours la valeur 0
- R31 sert à stocker l'adresse de retour des routines
- La mémoire est formée de mots de 32 bits adressées par octets
- `risc-emuf`, `risc-gui` (utilisés aussi en TP de Prog.)

Opérations logiques et arithmétiques

Exemple : Addition

Ajouter les registres R2 et R3 et mettre le résultat dans R1

```
ADD R1 R2 R3
```

Ajouter l'entier *16 bits signé n* à R2 et mettre le résultat dans R1

```
ADDI R1 R2 n
```

Ajouter l'entier *16 bits non signé n* à R2 et mettre le résultat dans R1

```
ADDIU R1 R2 n
```

Si R0 au lieu de R2, copie le troisième opérande (R3 ou *n*) dans R1, puisque R0 vaut 0.

Opérations mémoire

Exemple : Lire un mot en mémoire

Lit le mot (32 bits) situé à l'adresse (alignée) R2+*n* et stocke le résultat dans R1.

```
LDW R1 R2 n
```

Exemple : Écrire un mot en mémoire

Écrire la valeur de R1 à l'adresse (alignée) R2+*n*

```
STW R1 R2 n
```

Opérations de controle

Exemple : Saut conditionnel

Saute 4 instructions plus loin si R2 vaut 0.

```
BEQ R2 4
```

Exemple : Saut à une sous-routine

Saute à la sous-routine située 5 instructions plus loin. Sauve l'adresse de retour (instruction suivante) dans R31.

```
BSR 5
```

Il y a la version absolue : `JSR`.

Exemple : Saut à une adresse

Saute à l'adresse contenue dans le registre R5.

```
RET R5
```

Passage d'un code arborescent à un code linéaire

Exemple : $1 - (2 * 3)/4$

```
ADDIU R2 R0 1        // R2 <- 1
ADDIU R3 R0 2        // R3 <- 2
ADDIU R4 R0 3        // R4 <- 3
MUL R3 R3 R4       // R3 <- R3 * R4
ADDIU R4 R0 4        // R4 <- 4
DIV R3 R3 R4       // R3 <- R3 / R4
SUB R2 R2 R3       // R2 <- R2 - R3
```

Allocation des registres basiques

Vu qu'on ne fait pas un vrai compilateur, on va avoir un schéma d'allocation des registres très simple :

- On alloue les registres dans l'ordre en fonction des besoins et on les désalloue dans l'ordre inverse : pile de registres.
- Les registres étant alloués dans l'ordre, il suffit de se rappeler le registre au sommet de la pile dans une variable globale RSP.

Concrètement,

Expression E	code(E)
BinOp(Add,E_1,E_2)	code(E_1) ; code(E_2) ; emit("`ADD`",RSP-1,RSP-1,RSP) RSP $\leftarrow$ RSP - 1
⋮	⋮
IntLit(n)	RSP $\leftarrow$ RSP + 1 load_constant(RSP,n)

Génération du code des expressions

Le code générant les expressions sera produit par une fonction

```
val generate_expression : expression -> unit
```

Chaque expression résulte à l'exécution en une valeur stockée dans un registre. Par contrat, le résultat est placé dans le registre du sommet de la pile des registres (RSP).

Les conditions ?

Une condition est une expression vue comme une valeur booléenne.

- Résultat d'une comparaison `<=, <, ==, !=, >=, >`
- Opérande d'une opération logique

Typiquement provient d'un `if` ou d'un `while`.
Pour générer le code d'une condition, on imagine où sauter quand la condition est vraie et où sauter quand la condition est fausse.
Mais une des deux destinations est toujours “l'instruction suivante”.
Le code sera généré par une fonction

```
val generate_condition :
  expression -> label -> bool -> unit
```

generate_condition(e,l,b) génére le code qui saute au label *l* quand la condition *e* vaut *b*.

Variables locales et paramètres

Où stocker les variables locales et les paramètres des fonctions ?

- Allocation statique. À des adresses fixes (comme en Fortran) ?
  Non, car interdit certaines formes de récursivité.

Variables locales et paramètres

Où stocker les variables locales et les paramètres des fonctions ?

- Allocation statique. À des adresses fixes (comme en Fortran) ?
  Non, car interdit certaines formes de récursivité.
- Allocation dynamique, au moment de l'exécution.
  Solution utilisée par la quasi-totalité des langages actuels.

Stratégie d'allocation

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

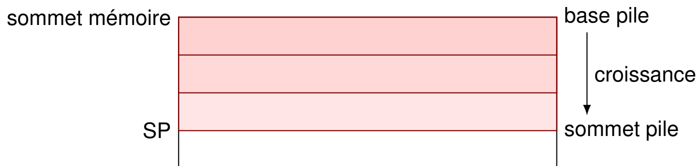
- La fonction en cours d'exécution se trouve au sommet de la pile,
- Lorsqu'une fonction en appelle une autre, cette dernière est placée au sommet de la pile et s'exécute en totalité avant de rendre la main à la fonction appelante (et donc d'être dépilée).

D'où l'idée d'une pile pour stocker les variables locales et les 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.



Manipulation de la pile en DLX

Il y a deux instructions qui facilitent la gestion de la pile :

Empiler une valeur

Empiler la valeur de R5 (rappel : R30 est le pointeur de pile)

```
PSH R5 R30 4
```

Dépiler une valeur

Dépiler le sommet de la pile et mettre le résultat dans R2.

```
POP R2 R30 4
```

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.

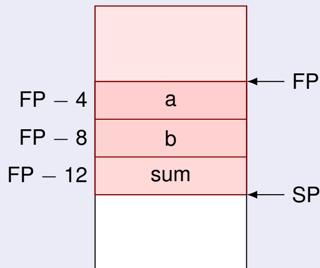
Exemple : Bloc d'activation de la méthode add

La fonction :

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

devient :

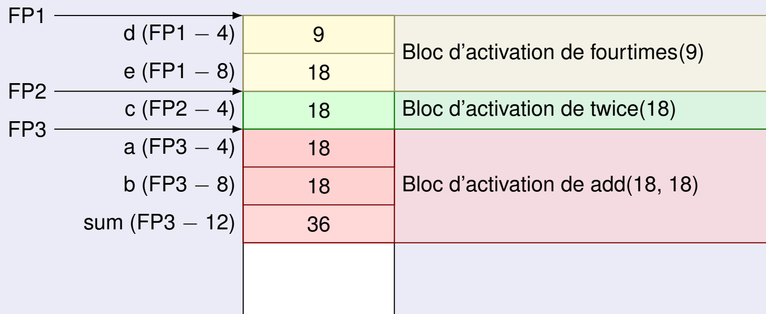
```
LDW R1 FP -4   // charge a
LDW R2 FP -8   // charge b
ADD R1 R1 R2   // calcule a+b
PSH R1 SP 4    // sauve sum
LDW R1 FP -12  // charge sum
```



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 (*) :



A-t-on vraiment besoin de FP ?

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 le cas pour `drei`...

A-t-on vraiment besoin de FP ?

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 le cas pour `drei`... mais pas pour C.

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.

En `drei`, toutes les valeurs sont courtes. Par convention, on va utiliser R1 (RES) pour communiquer le résultat d'une fonction.

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.

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.

Représentation des objets `drei`

Un objet en `drei` est composé en deux parties :

- la valeur de ses champs, spécifique à l'instance
- ses méthodes, partagées par toutes les instances de la classe

On manipule les objets par leur adresse (comme en Java). On dit aussi par référence.

- 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));
```

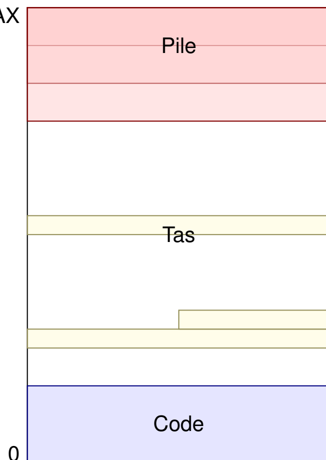
Il n'est pas possible d'allouer les objets sur la pile : ils ne survivraient pas à la fonction qui les a créés (ou alors il faudrait les copier au retour, ce qui est coûteux).

Ceci explique l'usage d'un tas.

Organisation de la mémoire

Typiquement, la mémoire est utilisée pour trois raisons :

1. Stockage du code, au bas de la mémoire, à partir de l'adresse 0.
2. Stockage des variables locales et des paramètres, sur la pile, qui croît vers le bas depuis le sommet de la mémoire.
3. Stockage des structures de données dynamiques (objets), sur le tas, entre le code et la pile.



Gestion de la mémoire dans l'émulateur DLX

Allouer 27 octets dans le tas

Allouer 27 octets dans le tas et mettre l'adresse du bloc alloué dans R1.

```
ADDI R1 R0 27  // R1 <- 27
SYSCALL R1 R1 12 // SYS_GC_ALLOC
```

Gestion de la mémoire dans l'émulateur DLX

Allouer 27 octets dans le tas

Allouer 27 octets dans le tas et mettre l'adresse du bloc alloué dans R1.

```
ADDI R1 R0 27   // R1 <- 27
SYSCALL R1 R1 12 // SYS_GC_ALLOC
```

Quand/Comment libérer la mémoire ?

Gestion de la mémoire dans l'émulateur DLX

Allouer 27 octets dans le tas

Allouer 27 octets dans le tas et mettre l'adresse du bloc alloué dans R1.

```
ADDI R1 R0 27  // R1 <- 27
SYSCALL R1 R1 12 // SYS_GC_ALLOC
```

Quand/Comment libérer la mémoire ?
Jamais. C'est fait automatiquement par le glaneur de cellules (ramasse-miettes) de l'émulateur DLX.

Initialisation de l'émulateur

Par convention, on utilisera R30 comme pointeur de pile.

Initialiser la pile en haut de la mémoire

```
SYSCALL R30 0 13 // SYS_GET_TOTAL_MEM_SIZE
```

Il faut aussi initialiser le glaneur de cellules de l'émulateur DLX en donnant l'adresse de début du tas, la taille du tas... (cf TP)

Méthodes de *dispatching* OO

Dans ce qui précède, on 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.

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

Problème : Comment réaliser le *dispatching* dynamique efficacement ?

Héritage simple

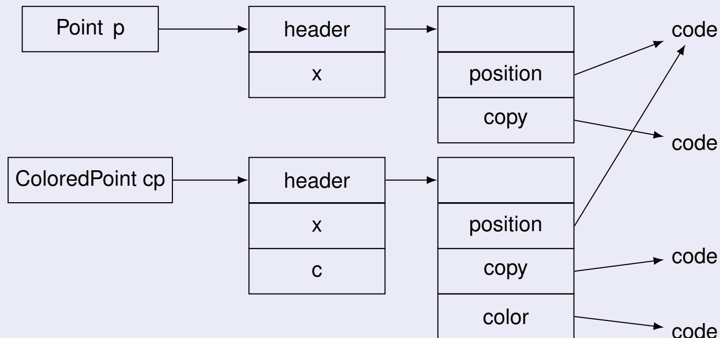
Le dispatching dynamique est relativement simple à mettre en oeuvre dans le cas de l'héritage simple :

- chaque classe a au plus une super-classe.

Exemple : Dispatching dynamique en `drei`

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

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 première entrée un pointeur vers la VMT de sa classe.

Génération des VMTs

Exemple des points

```
classPoint:
  DATA Point_position
  DATA Point_copy
classColoredPoint:
  DATA Point_position
  DATA ColoredPoint_copy
  DATA ColoredPoint_color
```

Code d'un programme complet

Un programme `drei` complet est composé :

- du code d'initialisation (pile, GC)
- des VMTs des classes,
- du code des méthodes et
- du code de l'expression principale.

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