Partie IX : Production de code pour les fonctions et optimisations

Dans cette partie nous étudions comment les déclarations et les appels de fonction sont traduits.

- Allocation de mémoire pour les paramètres et les variables locales.
- Prologues et épilogues de fonctions.
- Appels de fonctions.
- Possibilités d'optimisation.

Fonctions

- Jusqu'ici, nos programmes étaient formés d'une seule expression.
- Nous étudions à présent comment ajouter des fonctions.
- Exemple:

```
def fact (x: Int): Int =
  if (x == 0) 1 else fact (x-1) * x;
printInt (fact (2))
```

- Comment cela est-il traduit ?

Disposition du code

- Le code des fonctions est produit dans l'ordre de leur déclaration.
- Étant donné que l'expression principale vient en dernier, un saut initial de l'adresse 0 vers elle est nécessaire.
- L'appel d'une fonction se fait avec l'instruction BSR :

```
BSR L   // saute au label L, l'adresse de retour
        // est placée dans R31
```

- Le retour d'une fonction se fait avec l'instruction RET.

```
RET 31  // saute à l'adresse contenue dans R31
```

- R31 contient l'adresse de retour. On l'appelle le registre de lien (*link*), abrégé LNK.
- Pour notre processeur RISC, la signification de R31 est câblée dans l'instruction BSR. Avec d'autres processeurs il est possible de choisir le registre de retour.

Disposition du code (2)

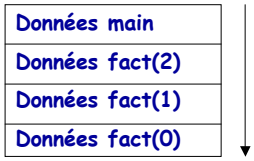
- Exemple :

```
0:        ...           // Initialisation des
                        // registres systèmes
          BRA MAIN      // Saut à l'expression
                        // principale
PRINTINT: // Code de la fonction prédéfinie printInt
          ...
          RET LNK
FACT:     // Code de la procédure fact:
          ...
          RET LNK
MAIN:     // Code de l'expression principale
          ...
          BSR FACT      // appel de fact
          BSR PRINTINT  // impression du résultat
          RET 0         // sortie du programme
```

- Note : RET 0 arrête le programme.

Disposition des données

- Comment les paramètres et les variables locales sont-ils alloués ?
- En Fortran : comme des variables globales; chaque fonction possède sa propre zone de variables globales.
- Ceci empêche toute récursivité, car différentes instances d'une fonction appelée récursivement risquent d'écraser ses variables locales.
- Les langages autorisant la récursivité stockent les paramètres et les variables locales sur une pile :

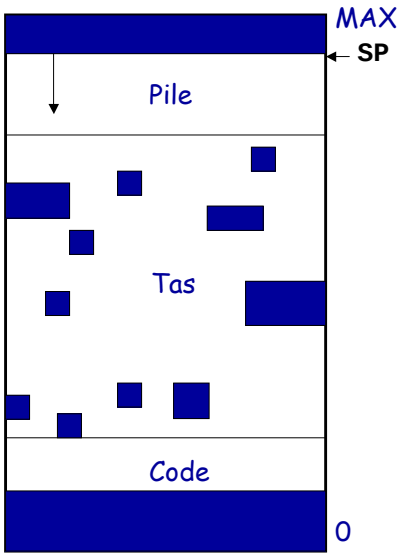


Variables locales en misc

- En misc, les fonctions n'ont pas de variables locales ; seuls les blocs (entourés d'accolades { et }) en ont.
- Pour simplifier, nous allons admettre ici que les fonctions ont des variables locales.
- Pour le projet, il faut légèrement modifier ce qui est présenté ici. Deux possibilités existent :
  1. Considérer que toutes les variables locales des blocs d'une fonction sont des variables locales de la fonction ; dans ce cas, il ne faut pas oublier que des blocs peuvent apparaître dans l'expression principale du programme, donc en dehors de toute fonction.
  2. Considérer que les fonctions n'ont effectivement pas de variables locales, et que l'allocation/libération de la mémoire pour les variables locales s'effectue au début et à la fin d'un bloc, et pas d'une fonction comme ici.
- La solution 2 semble préférable, car plus régulière.

La pile

- Dans l'architecture RISC la pile grandit en descendant, depuis le sommet de la mémoire, en direction du tas.
- L'adresse du sommet de la pile est stocké dans le registre pointeur de pile (*stack pointer*) SP = R30.
- L'organisation mémoire est donnée ci-contre.



Pile et passage de paramètres

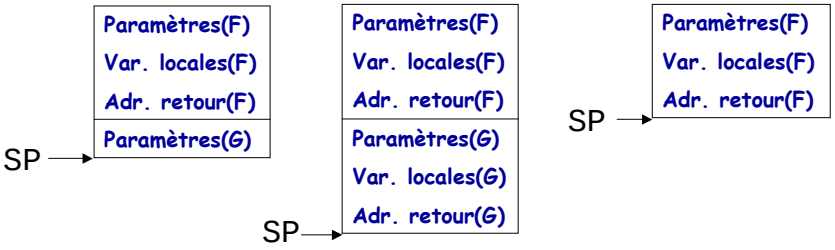
- Les paramètres peuvent aussi être passé sur la pile en utilisant l'instruction PSH.
- L'appel `fact(5)` ressemble donc à :

```
ADDI 1,0,5
PSH  1,SP,4
BSR  FACT
```

- Avant d'appeler une autre procédure, l'adresse de retour dans le registre LNK doit être sauvegardée sur la pile.

Disposition de la pile durant un appel

Voici la disposition de la pile lorsque la procédure F appelle la procédure *G* :



avant l'appel pendant l'exécution de *G* après l'appel

Prologue des fonctions

- Au début d'une fonction, on a la situation suivante :
  - les paramètres sont sur la pile,
  - l'adresse de retour est dans le registre LNK (R31)
- Avant de faire quoi que ce soit, la fonction doit :
  - sauvegarder l'adresse de retour sur la pile,
  - réserver de la place pour les variables locales sur la pile.
- Cette séquence d'instructions, qui est fondamentalement identique pour toutes les fonctions, est appelé le prologue des fonctions.
- Avec notre architecture, le prologue consiste en une simple instruction

```
PSH   LNK, SP, Taille(Var. locales) + 4
```

- Ici, `Taille(Var. locales)` est la taille de la zone nécessaire au stockage des variables locales, en octets, à laquelle on doit ajouter 4 pour pouvoir sauvegarder le registre LNK.

Epilogue des fonctions

- Avant de retourner, une fonction doit restaurer l'état de la pile avant son appel. Elle doit en particulier :
  - restaurer l'adresse de retour dans le registre LNK,
  - libérer l'espace utilisé sur la pile pour le stockage des variables locales et des paramètres.
  - retourner à l'appelant.
- On appelle cette séquence d'instructions l'épilogue de la fonction.
- Dans notre cas, deux instructions suffisent :

```
POP   LNK, SP, Taille(Var. locales)+Taille(Param.)+4
RET   LNK
```

- Ici, `Taille(Var. locales) + Taille(Param.)` est la taille de la zone de la pile nécessaire au stockage des variables locales et des paramètres de la fonction.

Résultat d'une fonction

- Comment une fonction retourne-t-elle son résultat à l'appelant ?
- Pour les résultats de la taille d'un mot : dans un registre. (En misc, tous les résultats ont la taille d'un mot)
- On réserve à cet effet un registre spécial `RES = R29`
- Pour les résultats plus grands, il y a deux possibilités :
  - sur la pile, ce qui complique la séquence de sortie,
  - 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.

Code pour la fonction factorielle

- Voici le code complet pour la fonction factorielle, fonctions prédéfinies exceptées.

```
        // Initialisation de SP
        SYSCALL SP,0,SYS_GET_TOTAL_MEM_SIZE
        // Initialisation du GC
        ...
        BRA     MAIN
        // Fonctions prédéfinies
        ...
FACT:   PSH     LNK,SP,4
        // if (n == 0)
        LDW     1,SP,4
        BNE     1,ELSE
        // 1
        ADDI    RES,0,1
        BRA     END
        // else fact(x-1)
ELSE:   LDW     1,SP,4
        SUBI    1,1,1
        PSH     1,SP,4
        BSR     FACT
```


Code pour la fonction factorielle (2)

```
        // * x
        LDW     1,SP,4
        MUL     1,1,RES
END:    POP     LNK,SP,8
        RET     LNK
        // printInt (fact (5))
MAIN:   ADDI    1,0,5
        PSH     1,SP,4
        BSR     FACT
        PSH     RES,SP,4
        BSR     PRINTINT
        RET     0
```

Adressage des variables locales

- Lors de l'adressage des variables locales, il importe de prendre en compte le fait que le pointeur de pile change lors du chargement des paramètres sur la pile.
- Exemple: si n est une variable locale à l'adresse #n par rapport au pointeur de pile après le prologue de la fonction. Un appel f(n, n) se traduit ainsi :

```
LDW 1, SP, #n
PSH 1, SP, 4
LDW 1, SP, #n + 4     // l'adresse a changé !
PSH 1, SP, 4
BSR f
```

- Pour produire le code d'accès aux variables locales et aux paramètres, il faut donc connaître à chaque instant le nombre de mots chargés sur la pile.

Listes comme paramètres et résultats de fonctions

- Lorsqu'on passe des listes en paramètre, on passe en fait un pointeur vers la première cellule de la liste.
- Lorsqu'on retourne une liste comme résultat, on retourne également un pointeur.
- En misc, une fonction peut retourner une liste qu'elle a créé localement, au moyen de l'opérateur « cons » (::). Exemple :

```
def prependTwice (elem: Int, lst: List[Int]): List[Int] =
  elem::elem::lst
```

- Cela implique qu'il n'est pas possible d'allouer les listes sur la pile, car elles ne survivraient alors pas à la fonction qui les a créées (ou alors il faudrait les copier au retour, ce qui est coûteux).
- Cette constatation explique l'usage d'un tas et d'un ramasse-miettes en misc.

Passage de paramètres dans les registres

- Il est aussi possible de passer les paramètres dans des registres (p.ex. dans les registres 1 à n).
- La fonction appelée doit alors sauvegarder les paramètres dans sa propre zone de variables locales.
- Le registre `LNK` est alors traité comme n'importe quel autre paramètre, d'où un gain en régularité.
- Exemple : Soit une fonction `f(x: Int, y: Int)`. Son prologue est :

```
SUB   SP, SP, Taille(Var. locales)+Taille(Param.)+4
STW   1, SP, #x       // sauvegarde de x
STW   2, SP, #y       // sauvegarde de y
STW   LNK, SP, 0      // sauvegarde de LNK
```

- L'appel `f(n,n)` se traduit alors :

```
LDW   1, SP, #n
LDW   2, SP, #n       // l'adresse ne change pas ici
BSR   f
```


Passage dans les registres ou sur la pile ?

- Quelle solution est la meilleure ? Passage sur la pile, ou dans les registres ?
- Avantages du passage dans les registres :
  - la régularité, dans la mesure où les paramètres, les opérandes d'expressions et l'adresse de retour sont traités de manière uniforme,
  - code potentiellement plus compact, car l'appelant n'a pas besoin de placer les paramètres sur la pile (particulièrement important pour les processeurs qui n'ont pas d'instruction PSH),
  - code potentiellement plus rapide, car les procédures feuille (celles qui ne contiennent pas d'appels) n'ont pas besoin de sauvegarder les paramètres,
  - 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 lorsque les registres ne sont pas légion, 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.
- Résumé : le passage par registres est meilleur lorsqu'on a suffisamment de registres à disposition.

Appels de fonction « embarqués »

- Examinons la variante suivante de la fonction factorielle (la partie modifiée est indiquée en rouge):
```
def fact (x: Int): Int =
  if (x == 0) 1 else x * fact (x-1)
```
- Une traduction naïve de l'appel récursif donne :
```
LDW   1, SP, #x
LDW   2, SP, #x
SUBI  2, 2, 1
BSR   FACT
MUL   1, RES, 1
```
- Mais cela est faux, car l'appel récursif va détruire le contenu du registre R1, dans lequel on a stocké x dans le but d'effectuer la multiplication.

- Dans ce cas, on peut résoudre le problème comme précédemment en réordonnant l'évaluation :

```
LDW   2, SP, #x
SUBI  2, 2, 1
BSR   FACT
LDW   1, SP, #x
MUL   1, RES, 1
```

- Mais cela n'est pas toujours possible, p.ex. :

```
f(x) + g(x)
```

- Il faut donc sauvegarder les registres en cours d'utilisation au moment d'un appel de fonction.
- Si on stocke les paramètres et/ou les variables locales dans des registres, le problème se présente encore plus fréquemment.

Sauvegarde par l'appelant ou par l'appelé ?

- La sauvegarde et restauration des registres lors d'appels peut être faite par l'appelant (*caller-save* en anglais) ou par l'appelé (*callee-save* en anglais).
- Sauvegarde par l'appelant : lors d'un appel de fonction, il faut sauvegarder tous les registres dont le contenu sera nécessaire après l'appel (registres LNK, SP et RES exceptés), puis les restaurer ensuite.
- Sauvegarde par l'appelé : lors du prologue, il faut sauvegarder tous les registres qui seront modifiés dans le corps (registres RES et SP exceptés) ; l'épilogue se charge de leur restauration. Tous les registres sont alors traités comme LNK.

- 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.
- Amélioration : on réserve un certain nombre de registres pour les variables locales, registres qui sont sauvegardés par l'appelé.
- Donc, 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.

Exemple de sauvegarde par l'appelant

- Voici le code pour l'appel récursif dans la seconde version de la fonction factorielle, en utilisant la sauvegarde des registres par l'appelant.

```
// x * fact(x-1)
LDW   1, SP, #x
PSH   1, SP, 4          // sauvegarde de x
LDW   1, SP, #x+4       // SP a changé !
SUBI  1, 1, 1
BSR   FACT
POP   1, SP, 4          // restauration de x
MUL   RES, 1, RES
```

Production de code pour les fonctions

- Compilation des déclarations de fonction :
  - allocation d'adresses pour les variables locales et les paramètres ; calcul de la taille nécessaire à leur stockage,
  - production de code pour le prologue,
  - production de code pour le corps,
  - production de code pour l'épilogue.
- Compilation des appels de fonction (en supposant un passage des paramètres par la pile) :
  - production du code de sauvegarde des registres qui seront réutilisés après l'appel,
  - production de code pour le chargement et l'empilage des paramètres sur la pile,
  - production de l'instruction d'appel BSR (ou JSR),
  - production du code de restauration des registres.
- Les symboles pour les fonctions doivent avoir un champ contenant l'adresse du code de la fonction.

Production de code pour misc : résumé

- Tâches de la production de code :
  - disposition de la mémoire et adressage des variables,
  - production de code pour les expressions, y compris les expressions conditionnelles,
  - production de code pour la définition et l'appel de fonctions.
- Réalisation au moyen d'un visiteur (en Java).
- Problèmes pas encore abordés :
  - optimisations,
  - production de code pour divers types de données, p.ex. les enregistrements et les objets,
  - compilation séparée et édition de liens (*linking*),
  - environnement d'exécution (*run-time*) : gestion de la concurrence, ramasse-miettes, etc.

Optimisations simples

- Voici quelques exemples d'optimisations simples qui pourraient facilement être ajoutées au compilateur misc.
- Le point commun de toutes ces optimisation est qu'elles peuvent s'effectuer directement sur l'arbre de syntaxe abstraite.
- Certaines optimisations plus avancées requièrent en général une autre représentation du code intermédiaire.

Réduction d'expressions constantes

- Soit la déclaration suivante :

```
var x: Int = 4 * 17
```

- On attend d'un bon compilateur qu'il produise le code suivant :

```
ADDI  1,0,68
STW   1,SP,#x
```

- Le calcul de l'expression constante doit donc se faire à la compilation.
- On peut facilement ajouter cela au visiteur gérant les opérations :

```
void caseOperation(Tree tree) {
  Item li = generate(tree.left);
  if (tree.right != null)
    Item ri = generate(tree.right);
    if (li instanceof ImmediateItem &&
        ri instanceof ImmediateItem) {
      item = constantFold(tree.op, li, ri);
    } else { ...
```

Réduction de force des opérateurs

- Quelques opérateurs « coûteux » peuvent être remplacés par d'autres opérateurs plus simples si une des opérandes est une constante particulière. P.ex.
  $x * 2^n$ devient $x << n$
- Si la spécification du langage définit le bon modèle d'arrondi (arrondi vers le bas pour les nombres négatifs et positifs), la division entière et le reste peuvent également se transformer :
  $x / 2^n$ devient $x >> n$
  $x \% 2^n$ devient $x \& (2^n-1)$
  Cette transformation est toujours correcte pour des x positifs, mais pour des x négatifs il faut que la division arrondisse vers le bas, et pas vers zéro. C-à-d que -3/10 doit valoir -1, pas 0.

  Malheureusement C, Java et la plupart des processeurs arrondissent vers 0.

Elimination de sous-expressions communes

- Soit le morceau de programme
  ```
  x = (a + b) / c ; y = (a + b) / d;
  ```
- Le compilateur pourrait décider d'évaluer (a + b) une seule fois, en transformant le programme ainsi :
  ```
  var temp: Int = a + b; x = temp / c; y = temp / d
  ```
- On appelle cette optimisation *élimination de sous-expressions communes* (*common subexpression elimination* ou *CSE* en anglais).
- Pourquoi ne pas demander au programmeur d'écrire lui-même la seconde version ?
  - le premier programme peut être jugé plus clair,
  - parfois les sous-expressions communes apparaissent dans du code que le programmeur n'a pas écrit, par exemple lors d'accès aux tableaux; ainsi, en Java, `a[i] = b[i]` contient implicitement deux sous-expressions de la forme `R = i * 4` pour le calcul de l'adresse des éléments.

- Cela dit, l'élimination de sous-expressions communes ne constitue pas toujours une amélioration.
  - Le stockage et le chargement des variables temporaires introduites ont un coût.
  - Ce coût est toutefois négligeable si les variables temporaires ne sont jamais stockées en mémoire.
  - Si l'élimination de sous-expressions communes introduit trop de variables temporaires stockées dans des registres, il est possible qu'il ne reste que peu de registres pour stocker d'autres données ; on dit alors que la pression sur les registres augmente.
  - Solution à ce dernier problème : on n'effectue l'élimination de sous-expressions communes que si on a assez de registres à disposition.

Détection de sous-expressions communes

- Soit l'extrait de programme Java suivant :

```
x = (a + b) / c;
a = a + 1;
y = (a + b) / d;
```

- Ici, `(a + b)` n'est pas une sous-expression commune car a est modifiée dans le second énoncé. Ce problème se pose dans tous les langages qui admettent les modifications de variables.
- Comment détecter alors les sous-expressions communes ?
- Solution : la numérotation des versions (*version numbering*).
  - Lors de la production de code, on donne à chaque variable locale un numéro de version, que l'on stocke p.ex. dans le symbole.
  - Chaque affectation à la variable incrémente sa version.
  - On recherche ensuite les sous-expressions communes dans lesquelles les numéros de version des variables utilisées concordent.

- Par exemple, notre extrait de programme après numérotation des versions ressemble à :

```
x1 = (a1 + b1) / c1;
a2 = a1 + 1;
y1 = (a2 + b1) / d1;
```

- La fausse sous-expression commune `(a+b)` a désormais disparu.
- Pour trouver les expressions qui ont déjà été évaluées, on a en général recours à une table de hachage associant des variables temporaires à des arbres :

```
HashTable<Tree, Symbol> evaluatedExpressions;
```


Optimisations avancées

- Les optimisations précédentes ne sont que le début. Les compilateurs réels en effectuent bien d'autres, et on en découvre constamment de nouvelles.
- Les principales sources d'optimisations sont :
  1. Éviter d'être trop « stupide » :
     - supprimer le code mort,
     - supprimer les sauts vers d'autres sauts,
     - supprimer les chargements de variable suivis d'un stockage de la même variable.
  2. Améliorer l'utilisation de la mémoire :
     - stocker les variables dans les registres,
     - améliorer la « localité » des accès aux variables, de sorte à ce qu'elles soient trouvées autant que possible dans l'antémémoire et pas dans la mémoire principale (ou, pour les gros programmes, dans la mémoire principale et pas sur disque).

3. Augmenter le parallélisme :
   - réordonner les instructions pour que plus d'instructions puissent être exécutées en parallèle et pour éviter les « bulles » dans le pipeline,
   - récrire les programmes pour diminuer le nombre de sauts, p.ex. en déroulant les boucles,
   - récrire les programmes pour utiliser les fils d'exécution multiples (*threads*).
4. Eviter les calculs répétés :
   - déplacer les instructions d'un endroit où elles sont exécutées souvent vers un endroit où elles sont exécutées moins souvent. Par exemple, déplacer les instructions en dehors d'une boucle si leur valeur est invariante par rapport à la boucle.