Langages et paradigmes

Olivier Lecarme

Master d'Informatique, première année

2007–2008

Sixième partie VI

Sous-programmes

Sous-programmes
Abstraction

- L'abstraction permet de nommer un fragment compliqué de programme qui a un rôle spécifique. Elle permet de ne plus s'occuper de l'implémentation de ce fragment.
- L'abstraction des données sert à représenter l'information.
- L'abstraction du contrôle sert à effectuer une opération précise.
- Le mécanisme d'abstraction de contrôle majeur est le sous-programme.

Terminologie

- L'appelant sous-traite le travail au sous-programme, et attend qu'il soit terminé.
- Il lui transmet des paramètres effectifs qui doivent correspondre aux paramètres formels du sous-programme.
- Un sous-programme qui rend une valeur est une fonction, s'il n'en rend pas c'est une procédure (mais la terminologie n'est pas figée).

Points importants à considérer

- séquence d'appel, qui gère la zone d'allocation du sous-programme
- représentation des fermetures
- modes de transmission des paramètres
- autres mécanismes d'abstraction du contrôle

Appel de sous-programme
Zone d'activation

À chaque sous-programme en cours d'exécution correspond une zone d'activation :

- Les zones d'activation sont placées dans la pile d'exécution.
- Le pointeur de pile pointe au sommet de la pile.
- Le pointeur de zone courante pointe dans la zone d'activation du sous-programme en cours.

- Les objets locaux de taille statique sont atteints par addition d'un déplacement à la valeur du pointeur de zone courante.
- Les autres objets locaux sont placés en sommet de pile et atteints de manière indirecte.
- Si les sous-programmes peuvent être emboîtés (Pascal, Modula-2, Ada), on place dans la zone d'activation un lien statique qui désigne la zone d'activation emboîtante.
- Dans tous les cas, l'ancienne valeur du pointeur de zone d'activation est sauvegardée comme lien dynamique.

zone d'activation

- Pour accélérer l'accès aux variables locales des sous-programmes emboîtants, on peut utiliser un tableau d'adressage :
  - l'élément *i* du tableau pointe sur la zone d'activation du sous-programme de niveau *i*
  - à chaque appel de sous-programme, on doit mettre à jour les dernières entrées dans le tableau
  - s'il est en mémoire, il impose un accès indirect systématique
  - s'il est en registres, il limite la profondeur d'emboîtement et bloque de nombreux registres
- Le fait que C (et ses descendants) n'emboîte pas les sous-programmes signifie-t-il que le concept est inutile ?

Instructions à exécuter

Trois suites d'instructions sont exécutées à l'occasion des appels de sous-programme :

- séquence d'appel dans le sous-programme appelant
- prologue en tête du sous-programme appelé
- épilogue à la fin du sous-programme appelé

instructions à l'appel

- Les actions à effectuer à l'appel sont :
  - passage des paramètres
  - sauvegarde de l'adresse de retour
  - branchement vers le sous-programme
  - allocation d'espace dans la pile d'exécution
  - sauvegarde des registres et pointeurs
  - modification du pointeur de zone d'activation
  - initialisation des objets locaux

instructions au retour

- Les actions à faire au retour sont :
  - passage des paramètres résultats
  - passage du résultat
  - finalisation des objets locaux
  - désallocation de la zone d'activation
  - restauration des registres et des pointeurs
  - retour à l'appelant
- La séparation des actions entre les trois suites d'instructions est assez délicate et dépend des possibilités de la machine.

Transmission des paramètres

- Dans les premières versions de Basic et de Cobol, pas de paramètres.
- Paramètres formels : ceux qui sont utilisés dans le sous-programme.
- Paramètres effectifs : les variables et expressions fournies à l'appel du sous-programme.
- Plusieurs modes de transmission, avec des propriétés sémantiques importantes et différentes.
- Appel en général sous forme préfixée, sauf pour Lisp et Scheme avec leur notation spécifique, et quelques langages qui permettent l'appel sous forme d'opérateur infixe.

Modes de transmission

- En Fortran, PL/I, Algol 68, C ou Lisp, il n'y a qu'un seul mode de transmission des paramètres, et c'est finalement l'appelant qui détermine comment le paramètre est transmis.
- En Algol 60, Pascal, Modula-2 ou Ada, on peut choisir parmi plusieurs modes à la déclaration du sous-programme, et l'appelant doit s'y conformer.
- Pour les langages à modèle de référence, un seul mode suffit normalement, mais une distinction est faite par ailleurs.
- Les deux modes fondamentaux sont le passage par valeur et le passage par référence.

Passage par valeur et passage par référence

- Le premier mode est par défaut en Pascal, le seul en C ou Icon : le paramètre formel est une variable locale au sous-programme, qui prend comme valeur initiale celle du paramètre effectif.
- Mais en C les tableaux ne sont jamais passés par valeur, ce qui est passé est en fait un pointeur sur le tableau.
- Pour permettre la modification du paramètre effectif, il faut en C passer explicitement un pointeur sur le paramètre véritable.

passage par valeur et passage par référence

- En Fortran, le seul mode est le passage par référence, normalement une valeur-G : le paramètre formel est un alias pour le paramètre effectif.
- Si l'on fournit une constante ou une expression comme paramètre effectif, ce qui est passé est un pointeur sur la variable temporaire contenant la valeur en question.
- En Lisp ou Smalltalk, toute variable est une référence, et le paramètre formel désigne le même objet que le paramètre effectif (passage par partage).

passage par valeur et passage par référence

- En Icon, le seul mode est le passage de valeur, et un sous-programme ne peut pas modifier un paramètre effectif.
- En Java, les paramètres effectifs de types prédéfinis sont passés par valeur, les autres par partage.
- Dans les langages fournissant les deux modes :
  - le passage par valeur permet de garantir que le paramètre effectif ne sera pas changé
  - le passage par référence permet de modifier le paramètre effectif
  - il évite de recopier la valeur
  - certains langages fournissent un mode évitant la copie mais empêchant la modification

Modes de passage des paramètres en Ada

- Trois modes : `in`, `out` et `in out`
- Les paramètres `in` peuvent être examinés mais non modifiés.
- Les paramètres `out` n'ont pas de valeur initiale, et sont affectés au paramètre effectif en fin de sous-programme.
- Les paramètres `in out` sont les deux à la fois.
- Pour les paramètres simples, `in` est un passage par valeur et `out` un passage par résultat.
- `in out` est un passage par valeur-résultat (Algol W).

modes de passage des paramètres en Ada

- Pour les paramètres composites, tout peut être implémenté par un passage par référence.
- La sémantique dépend donc de la nature des paramètres, et un programme qui en dépend est illégal.

Passage de fermetures en paramètre

- C'est le mode de transmission d'un sous-programme :

```
procedure Appliquer (function f (n : integer) : integer ;
    var T : array [bas..haut] of integer) ;
var i : integer ;
begin
    for i := bas to haut do
        T[i] := f(T[i])
end ;
```

- En C et C^{++}, on peut passer des pointeurs sur des sous-programmes.

passage de fermetures en paramètre

- En Modula-2, Algol 68 ou Ada 95, les sous-programmes sont des valeurs de première classe et peuvent être passés en paramètre.

```
TYPE fonc_int_int = PROCEDURE (INTEGER) : INTEGER ;
PROCEDURE Appliquer (f : fonc_int_int ;
                     T : ARRAY OF INTEGER) ;
VAR i : CARDINAL ;
BEGIN
    FOR i := 0 TO HIGH(T) DO
        T[i] := f(T[i])
    END
END ;
```

passage de fermetures en paramètre

- C'est évidemment normal et habituel dans les langages fonctionnels :

```
(define appliquer (lambda (f l)
    (if (null l) '()
        (cons (f (car l)) (appliquer f (cdr l))))))
```

- La fermeture est représentée par l'adresse du code et le lien statique, sauf s'il n'y a pas d'emboîtement des sous-programmes.

Passage par nom

- C'est une idée d'Algol 60 reprise par Simula 67.
- L'idée initiale était de décrire le mécanisme de passage de paramètre comme une copie de texte (comme dans un mécanisme de macros).
- Tout doit se passer comme si le texte du paramètre effectif venait remplacer le paramètre formel partout où il apparaît dans le sous-programme.
- On suppose en plus que des substitutions sont faites sur les noms pour éviter les conflits.

passage par nom

- Le paramètre effectif doit être réévalué à chaque utilisation, et dans le contexte de l'appelant.
- Le compilateur construit pour chaque paramètre effectif une procédure cachée et sans paramètre, et passe au sous-programme appelé une fermeture correspondant à cette procédure.

passage par nom

- Les utilisations peuvent être assez étonnantes :

```
real procedure somme (expr, i, bas, haut) ;
    value bas, haut ;
        comment expr et i sont passés par nom ;
    real expr ;
    integer i, bas, haut ;
begin
    real res ;
    res := 0 ;
    for i := bas step 1 until haut do
        res := res + expr ;
        comment la valeur de expr dépend de celle de i ;
    somme := res
end somme ;
```

Catégories spéciales de paramètres

Tableaux conformants

- La taille des tableaux est fixée :
  - en Pascal et Basic à la compilation
  - en Ada et Fortran 90 à l'élaboration de la déclaration
  - en APL, Icon et Perl à l'exécution
- Un paramètre tableau dont la taille n'est fixée qu'à l'appel du sous-programme est dit conformant (Pascal) ou ouvert (Modula-2, Ada).
- C permet de passer ce qu'on veut, mais sans la moindre vérification.

Paramètres facultatifs

- En Ada on peut proposer des valeurs par défaut pour certains paramètres :

```
type champ is integer range 0..integer'last ;
type nombre is integer range 2..16 ;
largeur_defaut : champ := integer'width ;
base_defaut : nombre := 10 ;
                    procedure mettre (element : in integer ;
                    largeur : in champ := largeur_defaut ;
                    base : in nombre := base_defaut) ;
```

paramètres facultatifs

- Les paramètres avec valeur par défaut sont facultatifs.
- S'ils apparaissent en dernier, on peut les omettre.
- Sinon, le mécanisme de nommage des paramètres permet de faire ce qu'on veut :

```
mettre(50, base => 8);
```

- On trouve aussi des paramètres facultatifs en C^{++} et Fortran 90.
- En Icon, les paramètres omis prennent la valeur `&null`.

Paramètres nommés

- Le mode normal d'énumération des paramètres est par position.
- En Ada ou Fortran 90, on peut nommer les paramètres (on parle de mot-clé).
- L'ordre d'énumération des paramètres effectifs ne reste important que pour ceux qui ne sont pas nommés.
- L'avantage supplémentaire est de fournir une documentation sur le rôle des paramètres spécifiés, utile s'il y en a beaucoup.

Nombre variable de paramètres

- En C ou C^{++}, on peut terminer une liste de paramètres formels par « . . . ».
- Il faut ensuite utiliser des macros spécifiques pour parcourir la liste des paramètres fournis en dernier.
- Le mécanisme ne permet pas de vérifier que les types correspondent à ce qui est attendu.
- En Icon, un en-tête spécifique permet d'avoir un paramètre qui représente la liste des paramètres.
- La vérification des types se fait comme dans le reste du langage.

Résultat de fonction

- Très souvent des restrictions sur le type du résultat, pour permettre de placer la valeur dans un registre :
  - en Algol 60 et Fortran, valeur scalaire
  - en Pascal, scalaire ou pointeur
  - en Algol 68, Ada, Modula-2 ou C, types composites permis également
  - en Algol 68, Ada 95 et dans les langages fonctionnels, résultat de type sous-programme (grâce à une fermeture)

Affectation du résultat

- dans un langage d'expression, valeur du corps de la fonction
- en Algol 60, Fortran et Pascal, affectation au nom de la fonction
- dans les langages plus récents, énoncé spécifique, par exemple **return** expression
- la plupart du temps il faut une variable locale pour le résultat, donc une copie inutile au moment de quitter la fonction

Mécanismes supplémentaires
Généricité

- Si l'on fabrique un type abstrait pour gérer des piles, on voudrait le paramétrer par le type des composants des piles.
- Le polymorphisme inclut dans le même sous-programme les traitements des divers types, mais oblige à des vérifications de types complexes ou coûteuses.

généricité

- La généricité permet d'engendrer un ensemble de sous-programmes pour chaque instance de type de composants, quand celui-ci n'est pas important pour le fonctionnement des sous-programmes.
- Principalement en Ada, C^{++} et Eiffel ; exemple en Ada :

généricité

```
generic
    type composant is private ;
    max_comp : in integer := 100 ;
package pile is
    procedure empiler(comp : in composant) ;
    function depiler return composant ;
private
    ... déclarations pour l'implémentation du type
end pile ;
package body pile is
    ... déclarations du corps des sous-programmes
end pile ;
...
package pile_operateur is new pile(operateur) ;
package pile_nombre is new pile(integer, 200) ;
```

généricité

- Mécanisme purement statique : le compilateur construit les différentes instantiations du type abstrait.
- Les parties du code qui ne dépendent pas des paramètres peuvent cependant être partagées.

Traitement d'exceptions

- Une exception est un événement imprévu ou inhabituel qui apparaît à l'exécution.
- Détecté par l'implémentation ou provoqué par le programme.
- Exemple très courant : erreur d'entrée-sortie (une donnée n'a pas la forme attendue ou une valeur en-dehors des limites).

traitement d'exceptions

- Traitements possibles :
  - choisir une valeur inventée
  - fournir une valeur indiquant l'état d'erreur
  - faire traiter l'erreur par un sous-programme spécifique passé en paramètre

traitement d'exceptions

- Première apparition en PL/I :
  - forme :

    ON condition
        énoncé ;
  - l'énoncé est exécuté quand la condition se produit, pas là où il apparaît
  - il peut abandonner le travail en cours par un énoncé de branchement

traitement d'exceptions

- Formes plus commodes en Ada, C^{++}, Java :
  - le traitement d'exception est lié à un bloc de code
  - son exécution remplace le bloc inachevé
  - si l'exception n'est pas traitée localement, elle est propagée au sous-programme appelant
- Mécanisme offrant en principe une très bonne sécurité, mais voir l'échec de la première mission Ariane V !

Coroutines

- Alors que l'appel de sous-programme établit une relation hiérarchique, l'appel de coroutine correspond à une relation de coopération égalitaire.
- Quand on revient d'une coroutine, on reprend l'exécution là où on l'avait suspendue ; quand on y retourne c'est la même chose.
- Si deux coroutines coopérent, leurs contextes d'exécution existent ensemble, et le contrôle alterne entre les deux.
- Implémentation d'une relation producteur-consommateur, client-serveur, de tâches.

coroutines

- Les tâches apparaissent en PL/I, Algol 68, Ada, Java.
- Les coroutines apparaissent en Simula, Modula-2, Icon.
- Opérations nécessaires :
  - créer un environnement d'exécution pour une coroutine
  - échanger le contrôle entre la coroutine courante et la coroutine appelée
  - détruire l'environnement d'exécution d'une coroutine

coroutines

- Gestion des zones d'activation :
  - quand on passe d'une coroutine à une autre, on doit changer de zone d'activation mais sans supprimer la première
  - plusieurs zones d'activation de même niveau dynamique coexistent
  - il faut donc les allouer dans le tas
  - chaque contexte d'exécution a sa propre pile, pour les sous-programmes internes
- L'opération d'échange est très simple et peu coûteuse.

Générateurs et itérateurs

- Un itérateur énumère des valeurs et permet de construire des boucles complexes.
- Il s'implémente facilement à l'aide de coroutines, mais n'en nécessite pas forcément.
- Un générateur le fait dans un mécanisme plus général, inclus dans les règles de base du langage.

générateurs et itérateurs

- Meilleur exemple : Icon.
  - Toute procédure peut devenir un générateur par l'énoncé `suspend`.
  - Dans les contextes d'évaluation dirigée par le but, on réveille les générateurs suspendus, en ordre inverse de leur suspension, jusqu'à réussite de l'évaluation ou épuisement de tous les générateurs.
  - Exemples :

```
every ligne := !read(fichier) do  ...
a := f(5 | 7 to 15 | 20, 10)
if a < b & c > d | g(1 to 10 by 2) then  ...
```