

# Parallélisation automatique

## Bases théoriques

Paul Feautrier

ENS de Lyon

Paul.Feautrier@ens-lyon.fr

2 février 2009



Université Claude Bernard



Lyon 1

# Comment raisonner sur un programme ?

Position du problème :

Etant donné un programme séquentiel  $P$ , trouver un programme équivalent  $P'$  qui s'exécute efficacement sur une architecture parallèle donné.

- ▶ L'équivalence de deux programmes est indécidable
- ▶ Il faut bâtir une théorie
- ▶ Il faut se contenter d'une équivalence *incomplète*

Un exemple de théorie : la sémantique dénotationnelle.

# Sémantique dénotationnelle naïve, I

On se donne deux ensembles primitifs,  $\mathcal{A}$  (les adresses) et  $\mathcal{V}$  (les valeurs). On suppose que  $\mathcal{V}$  contient entre autres les booléens **tt** et **ff**.

On construit les ensembles suivants :

- ▶  $\mathcal{M} :: \mathcal{A} \rightarrow \mathcal{V}$  (l'ensemble des fonctions des adresses vers les valeurs)
- ▶  $\mathcal{E} :: \mathcal{M} \rightarrow \mathcal{V}$  les expressions
- ▶  $\mathcal{S} :: \mathcal{M} \rightarrow \mathcal{M}$  les instructions

## Sémantique dénotationnelle naïve, II

Ces ensembles sont définis à l'aide de fonctions de traduction, traditionnellement notées  $[[ \ ]]$  qui associent un objet mathématique à une construction syntaxique. Par exemple :

- ▶ Si  $\mathfrak{n}$  est la représentation textuelle d'un entier, et  $n$  la valeur correspondante, alors :

$$\forall m \in \mathcal{M} : [[\mathfrak{n}]](m) = n.$$

- ▶ Si  $x$  est un identificateur, et si  $x$  est l'adresse correspondante, alors :

$$\forall m \in \mathcal{M} : [[x]](m) = m(x).$$

## Sémantique dénotationnelle naïve, III

- ▶ Si  $e_1$  et  $e_2$  sont deux expressions, alors :

$$\forall m \in \mathcal{M} : \llbracket e_1 + e_2 \rrbracket(m) = \llbracket e_1 \rrbracket(m) + \llbracket e_2 \rrbracket(m).$$

etc.

- ▶ Si  $x$  est un identificateur et  $e$  une expression, alors :

$$\forall m \in \mathcal{M} : \llbracket x := e \rrbracket(m) = m[x \leftarrow \llbracket e \rrbracket(m)],$$

soit, en utilisant le  $\lambda$ -calcul naïf :

$$\forall m \in \mathcal{M} : \llbracket x := e \rrbracket(m) = \lambda y \text{ **if** } y = x \text{ **then** } \llbracket e \rrbracket(m) \text{ **else** } m(y).$$

## Sémantique dénotationnelle naïve, IV

Et la règle la plus importante, qui donne le sens de l'exécution séquentielle :

Si  $S1$  et  $S2$  sont deux instructions :

$$\forall m \in \mathcal{M} : \llbracket S1; S2 \rrbracket(m) = \llbracket S2 \rrbracket(\llbracket S1 \rrbracket(m)),$$

ou encore :

$$\llbracket S1; S2 \rrbracket = \llbracket S1 \rrbracket \circ \llbracket S2 \rrbracket,$$

où  $\circ$  représente la composition de fonctions.

## Et le parallélisme ?

La sémantique dénotationnelle est insuffisante pour raisonner sur un programme parallèle :

- ▶ On peut montrer facilement :

$$[|x := x + 1; x := x + 2|] = [|x := x + 3|]$$

- ▶ Si on exécute ces codes en parallèle avec  $x := 0$  les résultats sont différents. Les valeurs possibles de  $x$  sont 0, 2, 3 pour le membre gauche et 0 ou 3 pour le membre droit.
- ▶ Il n'y a donc pas équivalence du point de vue de la programmation parallèle.

# Opérations, I

Une solution : la notion d'opération, qui permet de distinguer ce qui est exécuté de façon atomique et ce qui ne l'est pas.

- ▶ Une opération est l'exécution atomique d'une instruction
- ▶ Le choix de ce qu'est une instruction est arbitraire. On verra plus loin pourquoi
- ▶ On peut provisoirement imaginer qu'une opération est l'exécution d'une instruction en langage machine, ou d'une instruction d'un langage de haut niveau
- ▶ Dans l'exemple précédent, le membre gauche a deux opérations, le membre droit n'en a qu'une.

Pour fixer les idées : la durée d'un programme est grossièrement proportionnelle au nombre d'opérations exécutées.



## Opération, II

Pour pouvoir raisonner sur les opérations, il faut pouvoir les nommer. On verra plus loin plusieurs méthodes. Il ne suffit pas de nommer l'instruction sous-jacente, parce qu'une même instruction peut être exécutée un grand nombre de fois.

- ▶ On note  $U$  l'ensemble des opérations d'un programme
- ▶  $U$  peut être fini (programme) ou infini (système)
- ▶  $U$  dépend en général des données du programme. On convient que  $U$  est l'union de toutes les opérations exécutées pour des données légitimes. Une exécution sélectionne un sous ensemble de  $U$  parfois appelé une *trace*
- ▶ Si  $u \in U$ , on note par extension  $[[u]]$  la fonction sémantique associée à l'instruction exécutée par  $u$ .

# Ordre d'exécution

La donnée de  $U$  ne suffit pas : il faut savoir dans quel ordre les opérations sont exécutés. On postule un ordre d'exécution,  $<_{\text{seq}}$ .

- On convient de n'utiliser que des ordres *stricts*

$$u <_{\text{seq}} v, v <_{\text{seq}} w \Rightarrow u <_{\text{seq}} w,$$

$$u <_{\text{seq}} v \Rightarrow \neg v <_{\text{seq}} u,$$

ce qui implique  $\neg u <_{\text{seq}} u$

- Si le programme est séquentiel, on peut toujours dire, de deux opérations, laquelle est exécutée la première.  $<_{\text{seq}}$  est un ordre *total* :

$$u <_{\text{seq}} v \vee v <_{\text{seq}} u \vee u = v.$$

## Construction de $U$ et $<_{\text{seq}}$

Cette construction dépend du langage de programmation utilisé :

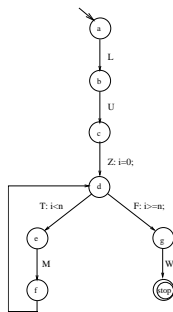
- ▶ Langage sans procédures :  $U$  est un langage rationnel
- ▶ Langage avec procédures :  $U$  est un langage hors-contexte (langage algébrique). Les instructions CALL et RETURN fonctionnent comme les parenthèses d'un langage de Dick
- ▶ Langage sans tests ni boucles WHILE ni GOTO :  $U$  est une union de Z-polyèdres
- ▶ Il est beaucoup plus difficile de définir  $U$  pour un langage fonctionnel ou pour un langage logique.

**Polyèdre** : ensemble des solutions d'un système de contraintes linéaires, ou bien intersection d'un nombre fini de demi-espaces linéaires ou bien coque convexe d'un ensemble fini de points

**Z-polyèdre** Ensemble des points entiers d'un polyèdre.

# Langage sans procédure

```
L: read y;  
U: x = 1;  
   for(i=0; i<n; i++)  
M:   x *= y;  
W: print x;
```



- ▶ Les états de l'automate sont les points stables entre instructions
- ▶ Chaque arc porte une instruction ou un test
- ▶  $U$  est le langage engendré par l'automate

- ▶ Chaque mot du langage est une trace
- ▶ Chaque préfixe d'une trace est une opération. L'ordre d'exécution est l'ordre préfixe
- ▶ Une trace : LUATMTMFW
- ▶ Il est difficile de raisonner sur cette représentation, car beaucoup de questions que l'on se pose sont indécidables.

## Programmes avec fonctions, esquisse

- ▶ On peut associer une grammaire à l'automate fini. On associe un non-terminal à chaque état. Par exemple, la production associée à l'état  $d$  est

$$d ::= Te|Fg$$

La production associée à l'état terminal  $g$  est  $d ::= W$

- ▶ une procédure est représentée par le non-terminal de son état initial
- ▶ si une instruction appelle une procédure, c'est le nom de la procédure qui figure sur l'arc correspondant
- ▶ par exemple, si on remplace  $x *= y$  ; par  $h(x,y)$  ; la production pour  $e$  devient

$$e ::= hf$$

et la grammaire devient hors contexte.

- ▶ Les définitions de  $u$ , des traces et de l'ordre d'exécution ne changent pas.

# Programmes à contrôle statique

- ▶ Un programme est à contrôle statique s'il n'a qu'une trace. Toutes les opérations de  $U$  sont exécutées.
- ▶ Pas de test et pas de boucle WHILE ; pas de GOTO
- ▶ La seule façon de réitérer une instruction est de l'inclure dans une boucle dont on sait compter les itérations :

```
for(i=0; i<100; i++)
```

- ▶ On peut légèrement généraliser en paramétrant le nombre de tours :

```
for(i=0; i<n; i++)
```

- ▶ et même utiliser des bornes variables :

```
for(i=0; i<n; i++)  
  for(j=0; j<i; j++)
```

On peut encore calculer le nombre de tours si la borne de la boucle interne dépend linéairement du compteur externe.

# Nids de boucles

Un nid de boucles est un ensemble de boucles emboîtées contenant des instructions exécutables (affectations).

- ▶ La *profondeur* d'une instruction est le nombre de boucles qui l'entourent.
- ▶ Le nid de boucles est *parfait* si toutes les instructions sont à la profondeur maximale.
- ▶ Chaque boucle a un *compte-tour* qui démarre à 0, qui croît par pas de 1 et qui n'est pas modifié dans le corps de boucle. On suppose que les compte-tours ont des noms distincts.
- ▶ On distingue les boucles `do`, qui font un nombre de tours déterminé d'avance, et les boucles `while` dont le nombre de tours est pratiquement imprévisible.

# Vecteur d'itération

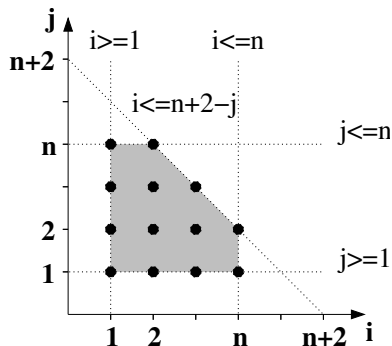
- ▶ Vecteur d'itération d'une instruction : liste des compte-tours des boucles englobantes, rangés de l'extérieur vers l'intérieur, et traitée comme un vecteur (pour simplifier les notations).
- ▶ Le vecteur d'itération est un objet mathématique ; ses composantes sont des variables dont on peut changer les noms et à qui on peut donner des valeurs (numériques ou algébriques).
- ▶ Une *opération* est caractérisée par le nom de l'instruction exécutée et par la valeur courante du vecteur d'itération :  $\langle S, i \rangle$ .



# Espace et Domaine d'itération

- ▶ Espace d'itération :  $\mathbb{N}^d$ , où  $d$  est le nombre de boucles englobantes.
- ▶ Domaine d'itération : l'ensemble des valeurs légales du vecteur d'itération.
- ▶ Le domaine d'itération est délimité par les bornes des boucles et par des tests.
- ▶ Il peut être connu à la compilation (boucles `do`) ou seulement à l'exécution (boucles `while`).
- ▶ Il peut être paramétré.

# Un exemple



```
for(i=1 ; i<=n; i++)  
  for(j=1; j<=n; j++)  
    if(i+j <= 2*n-2)  
      S;
```

# Ordre d'exécution, I/III

## Relation d'ordre

- ▶ Toute relation réflexive, symétrique et transitive.
- ▶ Ordre strict associé :  $x < y \equiv x \leq y \ \& \ x \neq y$ .
- ▶ Ordre total :  $x \leq y \vee y \leq x$ .
- ▶ Un ordre qui n'est pas total est partiel.

## Ordre d'exécution, II/III

- ▶ Dans un programme séquentiel, l'ordre d'exécution des opérations,  $<_{\text{seq}}$ , est entièrement fixé par le texte du programme et les instructions de contrôle.
- ▶ L'ordre d'exécution est *total*. Il vaut mieux travailler avec un ordre strict : éviter de dire qu'une opération est exécutée avant elle même.
- ▶ L'ordre engendré par les boucles est l'ordre lexicographique  $\ll$ .

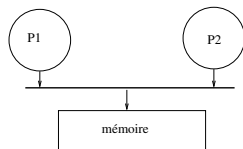
$$\begin{aligned}\langle S, i \rangle <_{\text{seq}} \langle T, j \rangle &\equiv i[1..N_{ST}] \ll j[1..N_{ST}] \\ &\vee (i[1..N_{ST}] = j[1..N_{ST}] \ \& \ S <_{\text{text}} T)\end{aligned}$$

- ▶  $<_{\text{text}}$  l'ordre *textuel*.
- ▶  $N_{ST}$  est le nombre de boucles englobant à la fois  $S$  et  $T$ .

# Qu'est-ce qu'un programme parallèle ?

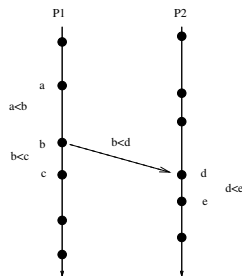
Observation : un programme parallèle a un ordre d'exécution *partiel*

## Exemple 1

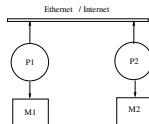


Symetric Multipro-  
cessor

- ▶ Les horloges ne sont jamais exactement à la même fréquence
- ▶ Si deux opérations exécutées par des processeurs différents, on ne peut pas dire dans quel ordre, sauf opération spéciale de synchronisation.



## Un autre exemple



- ▶ Deux ordinateurs reliés par un réseau, possiblement à très grande distance
- ▶ On suppose que P2 demande à P1 la valeur d'une de ses variables
- ▶ La valeur retournée va dépendre de l'état d'avancement de P1, en général imprévisible
- ▶ Solution : c'est P1 qui doit prendre l'initiative et P2 doit l'attendre
- ▶ On retrouve le schéma précédent

# Bilan

- ▶ On retrouve la même situation dans d'autres cas : par exemple, en multitraitement, l'état d'avancement des processus dépend de la politique du système d'exploitation : imprévisible
- ▶ La situation pour les machines synchrones est différente, on y reviendra plus tard
- ▶ Bilan :
  - ▶ Un programme séquentiel a un ordre d'exécution total
  - ▶ Un programme parallèle a un ordre d'exécution partiel. Plus la relation d'ordre est "petite" (en tant qu'ensemble) et plus le programme est parallèle
  - ▶ Le programme totalement parallèle a un ordre d'exécution vide !

## Parallélisme vrai / entrelacement

Que se passe-t-il si deux opérations veulent écrire simultanément en mémoire ?

- ▶ Rien de spécial : l'arbitre du bus en retarde une
- ▶ Mais il est pratiquement impossible de savoir laquelle est exécutée en dernier
- ▶ Autres exemples :
  - ▶ la détection des collisions sur un cable Ethernet
  - ▶ la modification d'une variable en section critique
  - ▶ le démon d'impression
- ▶ Tout se passe comme s'in n'y avait jamais de vrai parallélisme mais seulement *entrelacement* d'actions instantanées
- ▶ Exception : les réseaux recombinaents.



# Sémantique par entrelacement

Soit un programme dont l'ordre d'exécution est  $<$ .

- Une trace est une énumération des opérations

$$u_1, u_2, \dots, u_n$$

compatible avec  $<$  :

$$u_i < u_j \Rightarrow i < j.$$

- Si  $<$  est total, alors il n'y a qu'une trace
- Si  $<$  est partiel, il y a plusieurs traces. En particulier, si  $<$  est l'ordre vide, il y a  $n!$  traces.

# Déterminisme et indéterminisme

Effet d'une trace : composition des effets de chaque opération :

$$[|u_1; \dots; u_n|] = [|u_1|] \circ \dots \circ [|u_n|].$$

- ▶ Un programme séquentiel n'a qu'une seule trace et donc qu'un seul effet
- ▶ Pour qu'un programme parallèle soit équivalent à un programme séquentiel, il faut que toutes ses traces aient le même effet – qu'il soit *déterministe* – que celle de la version séquentielle

# Commutation

On considère le programme séquentiel minimum :

$u_1 ; u_2 ;$

et sa seule parallélisation possible :

$u_1 \text{ // } u_2 ;$

- ▶ Le programme parallèle a deux traces, qui doivent avoir le même effet :

$$[|u_1|] \circ [|u_2|] = [|u_2|] \circ [|u_1|]$$

- ▶ On dit que  $u_1$  et  $u_2$  commutent et on note  $u_1 \smile u_2$
- ▶ Si  $u_1$  et  $u_2$  ne commutent pas, on dit qu'elles sont en dépendance et on écrit  $u_1 \perp u_2$ .

# Exemples

- ▶ Il est facile de voir que  $x = 1$  et  $y = 2$  commutent (du moins si  $x$  et  $y$  sont des adresses distinctes)
- ▶ Par contre  $x = 0$  et  $x = 1$  ne commutent pas :

$$[|x = 0; x = 1|](m) = m[x \leftarrow 1],$$

$$[|x = 1; x = 0|](m) = m[x \leftarrow 0],$$

# Dépendances

La relation de dépendance est l'intersection de la relation de non commutation et de l'ordre séquentiel :

$$\delta = \perp \cap <_{\text{seq}} .$$

La relation de dépendance rassemble les contraintes que doivent satisfaire toute les versions parallèles du programme.

- ▶ Graphe de dépendance détaillé (GDD) : un graphe dont les sommets sont les opérations, avec un arc de  $u$  vers  $v$  si  $u \delta v$
- ▶ Ordre d'exécution parallèle : tout ordre plus fin que la fermeture transitive de la relation de dépendance :

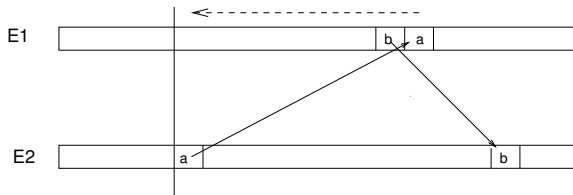
$$<_{//} \supseteq (\perp \cap <_{\text{seq}})^+$$

# Lemme de commutation

## Theorem

Toutes les exécutions possibles selon  $<_{//}$  ont le même effet.

## Démonstration.



On dit que l'ordre  $<_{//}$  *satisfait* toutes les dépendances. Cette démonstration dépend explicitement du fait que les exécutions sont finies.



# Calcul des dépendances

- ▶ Pour savoir si deux opérations commutent, on peut avoir besoin de toutes les mathématiques
  - ▶ commutativité et associativité des opérateurs de l'arithmétique
  - ▶  $a[x^3] := \dots$  et  $\dots := a[y^3 + z^3]$  : le grand théorème de Fermat
- ▶ il est (actuellement) impossible de mettre un démonstrateur de théorème dans un compilateur
- ▶ il faut trouver une condition *suffisante* de commutation plus simple
- ▶ on emploie un raisonnement *pessimiste*

# Conditions de Bernstein / I

S1 :  $y := f(x)$ ;

S2 :  $u := g(v)$ ;

- ▶ Si  $y \equiv u$ , alors  $S1; S2 \approx u = g(v_0)$ ,  $S2; S1 \approx u = f(x_0)$  et *il n'y a pas de raison* pour que ces quantités soient égales. Il y a dépendance.
- ▶ Si  $y \not\equiv u$  mais  $y = v$ ,  $S1; S2 \approx y = f(x_0)$ ,  $u = g(f(x_0))$ ,  $S2; S1 \approx u = g(y_0)$ ,  $y = f(x_0)$ , dépendance.
- ▶ Situation symétrique pour  $u = x$ .



## Condition de Bernstein / II

- Condition suffisante d'indépendance :

### Theorem

*Pour que S1 et S2 soient indépendantes, il suffit que  $y \neq u, y \neq v, u \neq x$ .*

- Plus généralement,  $\mathcal{R}(S)$  ensemble des variables lues,  $\mathcal{M}(S)$  ensemble des variables modifiées par  $S$ .

### Theorem

*Pour que S1 et S2 soient indépendantes, il suffit que :*

$$\mathcal{M}(S1) \cap \mathcal{M}(S2) = \emptyset, \mathcal{M}(S1) \cap \mathcal{R}(S2) = \emptyset, \mathcal{R}(S1) \cap \mathcal{M}(S2) = \emptyset.$$

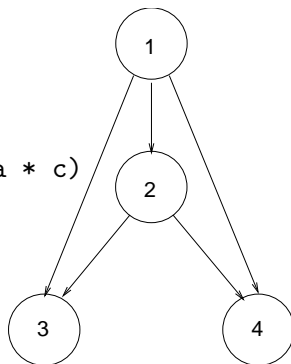
- Ces conditions ne sont pas nécessaires :

S1 :  $x := x + 1;$

S2 :  $x := x + 3;$

# Un Exemple

```
1  read a, b, c
2  delta = sqrt(b*b - 4.0 * a * c)
3  x1 = (-b + delta)/2.0/a
4  x2 = (-b - delta)/2.0/a
```



## Lemme de commutation, le retour

Que se passe-t-il pour une exécution infinie ?

- ▶ Systèmes d'exploitation, systèmes transactionnels, traitement du signal
- ▶ Dans le cas d'un programme qui ne se termine pas, on ne peut plus parler de résultat
- ▶ Il faut considérer l'histoire de chaque variable

# Définitions

- ▶ Trace : l'enregistrement de tout ce qui se passe lors d'une exécution. Tuples  $\langle$  opération, variable affectée, valeur affectée, variables lues  $\rangle$ .
- ▶ Histoire d'une variable  $x$  : une trace réduite aux opérations qui affectent  $x$ .
- ▶ Source de  $x$  au point  $k$  d'une trace  $t$  : l'opération antérieure la plus proche de  $k$  qui affecte  $x$  :

$$\sigma(x, t, k) = \max\{i \mid i < k, x \in M(t_i)\}.$$

## Lemme de commutation, II

### Lemma

*Soit  $t$  une trace respectant l'ordre  $<_{//}$  et  $x$  une variable. Dans l'histoire de  $v$  selon  $t$ , les opérations apparaissent dans le même ordre que dans le programme séquentiel original.*

### Démonstration.

En effet, toutes les opérations qui affectent  $x$  sont en dépendance, donc sont ordonnées par  $<_{//}$  dans le même ordre que dans  $<_{seq}$ . □

# Lemme de commutation, II

## Lemma

*Dans les mêmes conditions, la fonction source pour  $x$  dans  $t$  est la même que dans le programme séquentiel.*

## Démonstration.

Il suffit d'observer que le calcul d'une source de  $x$  ne fait intervenir que les opérations de l'histoire de  $x$  et leur ordre. □

## Lemme de commutation, III

### Theorem

*Une variable quelconque a la même histoire dans toutes les exécutions selon l'ordre  $<_{//}$ .*

### Démonstration.

Soit deux traces  $t_1$  et  $t_2$ , soit  $x$  une variable ayant pour histoires  $h_1$  et  $h_2$ , et supposons que ces histoires divergent en position  $k$ . Il correspond à  $k$  deux positions  $k_1$  et  $k_2$  dans  $t_1$  et  $t_2$  et nous dirons que la divergence se manifeste à l'instant  $\max(k_1, k_2)$ . Comme les opérations sont déterministes et ont les mêmes sources dans les deux exécutions, il faut qu'il y ait une divergence pour l'une des sources aux instants respectifs  $k'_1 < k_1$  et  $k'_2 < k_2$ . Il en résulte que  $\max(k'_1, k'_2) < \max(k_1, k_2)$ . On forme donc ainsi une suite de divergences qui se manifestent de plus en plus tôt. Ceci ne peut se poursuivre indéfiniment, puisque une trace a un début. On a donc trouvé une contradiction. □

## Autres dépendances

Par extension, on introduit d'autres sortes de dépendances : tout ce qui contraint l'ordre d'exécution :

- ▶ Dépendances de contrôle : le prédicat d'un test doit être évalué avant ses branches. Idem pour le prédicat d'un `while`.
- ▶ Dépendances de ressources : deux opérations qui utilisent la même ressource ne peuvent pas être exécutées simultanément

On parle également de dépendances CC (*input dependence*) quand deux opérations lisent la même cellule de mémoire. Utile pour l'optimisation de la localité.



# Agrégation

- ▶ Une propriété importante des ensembles lus et modifiés :

$$\mathcal{M}(u; v) \subseteq \mathcal{M}(u) \cup \mathcal{M}(v)$$

et la formule analogue pour  $\mathcal{R}$

- ▶ Il est facile de trouver des exemples où l'inclusion est stricte. Il s'agit donc d'un cas de pessimisme
- ▶ Ceci justifie tous les cas d'agrégation, où l'on considère un groupe d'opération (e.g. l'exécution d'une boucle ou d'une fonction) comme une seule opération.