Optimisation de programmes annotés par des assertions

Pierre Amiranoff

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimisation de programmes annotés par des assertions

Pierre Amiranoff

N ° 3983

16 août 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3983--FR

INRIA
ROCQUENCOURT

Optimisation de programmes annotés par des assertions

Pierre Amiranoff

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n ° 3983 — 16 août 2000 — 28 pages

Résumé : Dans les *codes scientifiques*, les traitements répétitifs prennent la forme de boucles qui monopolisent l'essentiel des ressources en temps et en mémoire du calculateur. Les techniques d'optimisation ciblent donc particulièrement ces boucles. Cependant, les techniques efficaces existantes ne sont pas généralisables à tous les programmes.
Nous montrons sur des exemples que des propriétés sémantiques de haut niveau, qui, habituellement, ne sont plus apparentes après l'implémentation, peuvent être mises en mémoire sous forme d'assertions, ce qui permet de valider une *parallélisation de boucle*. En effet, ces assertions expriment l'indépendance des références-mémoires internes à la boucle. Nous détaillons la procédure de preuve de ces assertions.
Nous analysons un programme utilisé dans le domaine des *maillages* qui combine la restructuration de données avec la recherche et l'exploitation de celles-ci pour des applications diverses. Cette analyse nous conduit à construire un programme équivalent qui peut être parallélisé. Nous resituons notre exemple dans le cadre du modèle *Entité-Relation* des Bases de Données Relationnelles.
Nous instancions notre programme sur 3 exemples d'applications. Nous dégageons des classes de programmes suivant les modalités d'exploitation des résultats acquis lors de la phase de recherche. Ces classes se différencient selon le type de *dépendances de données dans les boucles*. Enfin, nous illustrons les possibilités de parallélisation par un programme emprunté à la Géométrie et deux programmes inspirés de l'Arithmétique.
Plus généralement, nous souhaitons contribuer à renforcer l'influence des *méthodes de spécifications formelles* dans le développement des Applications Numériques.

Mots-clé : calcul scientifique, optimisation, spécifications formelles, preuves formelles, assertions, parallélisation de boucle, maillages, bases de données

(Abstract: pto)

Merci à *François Thomasset* pour ses conseils attentifs

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Optimization of programs annotated with assertions

Abstract: In *scientific codes*, repetitive processings take shape as loops that corner main part of time and memory resources of the processor. The optimization techniques thus target particularly these loops. Unfortunately, the effectively existing techniques do not extend to every species of programs.
We show on examples that high level semantic properties, which usually become hidden with the implementation, can be put in memory in the form of assertions, which makes it possible to validate *loop parallelizations*. Indeed, these assertions express the independence of the memory references appearing in the body of the loop. We detail the proof procedures of these assertions.
We analyze a program used in the field of *Mesh Generation* which combines data reorganization with search and exploitation of those for various applications. This analysis leads us to build an equivalent program which can be parallelized. We put our example in perspective within the framework of the *Entity-Relation* model of the Relational Data-Bases.
We instanciate our program on 3 application examples. We relieve classes of programs with respect to the exploitation mode of the results acquired during search phase. These classes distinguish themselves on the basis of *loop data dependence relations*. Lastly, we illustrate the possibilities of parallelization with some programs borrowed from the fields of Geometry and of Arithmetic.
Broadly speaking, we wish to contribute to reinforce the *formal specification methods* influence within Numeric Applications Development.

Key-words: scientific code, optimization, formal spécification, formal proof, assertion, loop parallelization, mesh generation, relational data-base

Table des matières

Chapitre 1

Introduction

Dans les *codes scientifiques*, les traitements répétitifs prennent la forme de boucles qui monopolisent l'essentiel des ressources en temps et en mémoire du calculateur. Les techniques d'optimisation ciblent donc particulièrement ces blocs d'instructions.
Ces optimisations demandent une connaissance des références en mémoire manipulées par les opérations de la boucle. Il existe des techniques efficaces pour des configurations particulières. C'est le cas des références mémoire fonctions linéaires des indice de boucle [Wol96] [Fea91]. D'autres techniques analysent des programmes utilisant des pointeurs [Ghi98].
Dans les situations où ces analyses sont impuissantes, il s'avère que des propriétés sémantiques de haut niveau, caractérisant la spécification du problème disparaissent lors de l'implantation, c'est-à-dire dans l'écriture du programme. Or, ces propriétés pourraient être exploitées pour organiser ou justifier une optimisation.
Les méthodes formelles [Mon96] permettent d'établir des assertions qui caractérisent les propriétés d'un programme, assertions validées par une preuve formelle. Nous nous intéressons aux assertions qui permettraient l'optimisation du programme. Nous exposons également de façon détaillée une procédure de démonstration de ces assertions.
Une telle optimisation peut être réalisée en suivant différentes stratégies:

- l'assertion est une précondition pour réaliser une transformation optimisante du programme-source, transformation mise en oeuvre manuellement. L'assertion - qui doit elle-même être prouvée - valide la correction du programme transformé
- l'assertion prouvée et insérée dans le programme-source est une information exploitable par le compilateur en tant que précondition pour effectuer automatiquement une tâche d'optimisation
- l'assertion est insérée dans le programme-source, et un code paramétré par sa valeur booléenne est généré; l'exécution du code tient alors compte de la valeur de l'assertion calculée dynamiquement

Par *programme*, nous entendons ici la combinaison d'un ensemble de données en entrée et d'un algorithme qui manipule ces données.
Nous allons nous focaliser ici sur un type particulier d'optimisation du programme-source: la parallélisation de boucle. Nous mènerons de pair l'optimisation de programme et la preuve de l'assertion qui valide cette optimisation.
Dans cet article, nous étudions un programme relatif aux maillages. Sauf indication ou évidence contextuelle, nous parlerons de *programme* pour désigner un couple de programmes, l'un dédié au rangement des données, le second à la recherche et à l'exploitation de ces données pour les besoins de l'application cliente. Nous formalisons une propriété caractéristique qui portent sur les valeurs des données fournies en entrée. Nous montrons que la validité de ce prédicat est exigible à la fois pour assurer la correction du programme et pour ouvrir la voie à son optimisation. Ainsi, l'assertion, qui est logiquement indépendante de l'algorithme du programme-source, est -sémantiquement parlant- partie intégrante de ce programme.
Le chapitre 2 est consacré à une reformulation du programme, dans le but de rendre possible la parallélisation de boucle. Le nouveau programme vérifiera alors une nouvelle assertion contraignant de façon équivalente les valeurs prises par les données.
Le 3ème chapitre expose quelques exemples de programmes dont le corps est une boucle. Cette boucle est parallélisable sous le contrôle d'une assertion accompagnant le programme.

Progression du rapport

Le $2^{ème}$ chapitre étudie une $1^{ère}$ version du programme.
En section 2.1, nous présentons le programme qui sera notre support dans la démarche d'optimisation. Une première série de modifications non essentielles d'ordre technique est effectuée.
Section 2.2, nous présentons la propriété caractéristique de la sémantique du programme. Nous rappelons la méthode de preuve de Hoare [Hoa69] [BB83] au cours de sa mise en pratique. La preuve d'un invariant de boucle est exposée, validant la propriété supputée.
La section 2.3 expose les bénéfices de ce travail: une preuve d'arrêt de la boucle de recherche et les perspectives de son optimisation.
Le chapitre 3 réexamine l'application à l'aune de ces perspectives.
Cela conduit, section 3.1, à une reformulation du programme, préalable à sa parallélisation, et à la batterie de preuves que nécessite la vérification de la propriété caractéristique dans ce nouveau contexte.
La section 3.2 exploite les résultats obtenus pour permettre une parallélisation des calculs. 3 applications sont alors encapsulées dans notre moule logiciel.
La section 3.3 tente de réinterpréter notre propos sous l'angle du modèle *Entité-Relation* des Bases de Données Relationnelles.
Précédant la conclusion, le chapitre 4 illustre dans les domaine des maillages et de la théorie des nombres les optimisations réalisables grâce au nouveau modèle de rangement/recherche-exploitation.
Le rapport peut être parcouru en sautant l'exposition des démonstrations, selon le désir du lecteur.

Chapitre 2

Correction d'un programme couplé de rangement/recherche-exploitation

2.1 Les programmes "*Chaînuple*" et "*Déchaîne*"

2.1.1 Présentation des programmes

Le couple de programmes que nous allons étudier est une production de l'équipe Gamma de L'INRIA [FG99], qui s'intéresse aux maillages automatiques. L'application se décompose en une étape de rangement structuré de données et une étape de recherche-exploitation de ces données depuis la structure construite. Notre objectif ici ne sera pas de prouver la correction sémantique totale des programmes manipulés mais sera circonscrit à l'optimisation de la recherche-exploitation. En passant, la validation de la propriété caractéristique explicitée plus loin produira un résultat annexe, la preuve de l'arrêt de la boucle.
Le tableau $Tria(1 : n_e, 1 : 3)$ est une donnée. Il conserve les (indices des) sommets appartenant à chaque élément, ici des triangles. Une ligne de *Tria* conserve les 3 sommets d'un triangle.
Une nouvelle structure de données, composée de 2 tableaux, *List* et Tab permet de conserver ces informations sous une forme efficace relativement à la recherche de tous les éléments possesseurs d'un sommet donné. Le programme "*Chaînuple*" a pour fonction de réorganiser les données, tandis que le programme "*Déchaîne*" effectue la recherche relative à un sommet donné. Alors que le tableau $Tria$ délivre les sommets d'un triangle donné, le couple $(List, Tab)$ retrouve les triangles attachés à un sommet S. L'ensemble de ces triangles constitue *la boule de* S. Nous pourrions le qualifier de *structure duale* de $Tria$. Quoiqu'il contienne sous une autre forme la même information que $Tria$, dans la pratique il ne le remplace pas mais joue un rôle complémentaire.
Au cours du programme de rangement "*Chaînuple*", le stockage des données est linéarisé: l'indice w, que nous appelons *rang* en référence à l'ordre de parcours de $Tria$ croissant lexicographiquement suivant (i, j), est substitué au couple (i, j) représentatif d'une occurrence locale j de sommet dans un triangle i. A chaque itération, $Tab(s)$ enregistre le rang de cette itération, mémorisant la dernière occurrence visitée du sommet s. En fin de parcours, Tab contient ainsi la liste des dernières occurrences de chaque sommet. $Tab(s)$ joue alors le rôle de pointeur vers la tête d'une *chaîne* qui garde la trace des occurrences du sommet s. "*Chaînuple*" entrelace dans List cet ensemble de *chaînes* .
Le programme de recherche/exploitation "*Déchaîne*" se comprend de lui-même.
La sémantique de cette construction repose sur une propriété fondamentale. Par définition de $Tria$, chaque ligne a pour rôle de mémoriser les 3 sommets d'un triangle: les 3 valeurs référencées sur une telle ligne sont distinctes. Les données initiales de "*Chaînuple*" vérifient donc l'axiome:

$$\forall\, i_1, j_1, i_2, j_2, \quad Tria(i_1, j_1) = Tria(i_2, j_2) \quad \Rightarrow \quad j_1 = j_2 \ \vee \ i_1 \neq i_2 \qquad (A_0)$$

rangement

Initialisation

```
Pour S = 1, ns
    Tab(S) := -1
Fin pour
```

Chaînuple

```
w := 0
Pour i = 1, ne
    Pour j = 1, 3
        s         := Tria(i,j)
        List(w)   := Tab(s)
        Tab(s)    := w
        w         := w + 1
    Fin pour
Fin pour
```

recherche

Initialisation

```
w := Tab(S)
```

Déchaîne

```
Tant_que w <> -1

/*  S est le j_ième sommet de l'élément i   */

    i := w/3 + 1
    j := w - 3*(i-1)+1
    Calculs sur  i et j ...
    w := List(w)
Fin_tant_que
```

2.1.2 Modification des programmes

Nous opérons 3 modifications mineures sur ces textes :

- recaler les bornes initiales des indices i, j et w sur 0. Cela simplifie les relations entre eux; nous aurons maintenant : $w = 3 * i + j$ $i = w/3$ $j = w \ mod \ 3$
- fusionner les 2 boucles imbriquées en une boucle d'indice w
- déplier le tableau à 2 dimensions $Tria(i, j)$ en un tableau à 1 dimension $Triaj(w)$
- changer l'initialisation de $Tab(s)$ dans le but de faciliter, comme nous le verrons, l'expression des invariants de boucle.

Les nouveaux programmes sont les suivants :

rangement

Initialisation

```
Pour S = 0, ns-1
    Tab(S) := -3
Fin pour
```

Chaînuple

```
Pour w = 0, 3*ne-1

   s         := Triaj(w)
   List(w)   := Tab(s)
   Tab(s)    := w
Fin pour
```

recherche

Initialisation

```
w := Tab(S)
```

Déchaîne

```
Tant_que w <> -3

/*  S est le sommet local j de l'élément i   */

    i := w/3
    j := w mod 3
    Calculs sur  i et j ...
    w := List(w)
Fin_tant_que
```

2.2 Preuve d'une propriété de "*Chaînuple*"

Tout au long du rapport, nous emploierons les techniques de démonstration dérivées de la logique de Hoare. Pour plus de détails concernant les règles utilisées, voir par exemple [Hoa69, BB83, BB88].
Le but de cette section est de démontrer qu'une *chaîne* ne peut visiter 2 fois le même triangle. Cette propriété sera notre levier pour optimiser l'opération de recherche dans la base de données construite par "*Chaînuple*".
Notons L la borne supérieure de l'indice de boucle du programme de rangement: $L = 3 * ne - 1$.
Avec les nouvelles notations, l'axiome 'A_0' s'écrit :

$$\forall\ w_1, w_2 \in [0, L], \quad Triaj(w_1) = Triaj(w_2) \quad \Rightarrow \quad w_1 = w_2 \ \vee\ w_1/3 \ \neq\ w_2/3 \qquad (A)$$

2.2.1 La propriété caractéristique du tableau *List*

Soit la formule : $$\forall\alpha, \quad 0 \leq \alpha \leq w-1 \quad \Rightarrow \quad List(\alpha)/3 \ < \ \alpha/3 \qquad (B)$$

Nous voulons prouver que le programme respecte en postcondition la propriété :

$$\forall\alpha, \quad 0 \leq \alpha \leq L \quad \Rightarrow \quad List(\alpha)/3 \ < \ \alpha/3 \qquad (\ B(w \hookleftarrow L+1)\ )$$

Cette propriété caractérise au niveau de *List* le fait que tout triangle a ses 3 sommets distincts.

Argumentation informelle :
Si α représente une référence de *List* , $\alpha/3$ représente l'indice i de la matrice $Tria$, c-à-d. le numéro du triangle concerné.
Or $List(\alpha)$ représente une référence de *List* relative à un triangle scruté antérieurement par la boucle de rangement. Le numéro de ce triangle - $List(\alpha)/3$ - est donc strictement inférieur au précédent.

2.2.2 Règle de déduction pour la boucle '*Pour*'

Nous employons la notation de Hoare [Hoa69] : '*précondition* $\{Manip\}$ *postcondition*' signifie :
Si '*précondition*' est vraie avant l'exécution du programme ou de la séquence d'instructions "*Manip*", alors, si "*Manip*" se termine, '*postcondition*' est vraie après l'exécution de "*Manip*".
La notation '$I(w \hookleftarrow w+1)$' signifie 'l'expression I dans laquelle on a substitué $w+1$ à w'.

Règle : Soit la boucle 'Pour w:= *ini jusqua fin faire Manip*'
La séquence d'instructions "*Manip*" ne doit pas modifier les valeurs de *ini*, *fin*, w.
Nous avons la règle suivante :

$$Si \quad (ini \leq w \leq fin) \quad \& \quad I\ \{Manip\}\ I(w \hookleftarrow w+1)\ , \quad alors :$$

$$I(w \hookleftarrow ini)$$

$$\boldsymbol{\{pour\ w{:=}\ ini\ \ jusqua\ \ fin\ \ faire\ \ Manip\}}$$

$$(ini > fin)\ \&\ I(w \hookleftarrow ini) \quad \vee \quad (ini \leq fin)\ \&\ I(w \hookleftarrow fin+1)$$

La signification intuitive est : si le prédicat I se retrouve préservé après une itération arbitraire de la boucle (passage de l'indice de boucle de w à $w+1$), alors s'il est vrai à l'entrée de la boucle, il est toujours vrai à la sortie de la boucle.
I est appelé *invariant* de la boucle '*Pour*'.

Remarque : il faut prendre garde de ne pas confondre la propriété d'invariance d'une formule I lors de l'exécution d'une boucle avec la vérité de cette formule en un point particulier du programme :

- Au cours de l'exécution d'une itération de la boucle, la vérité de I n'est pas assurée.
- la vérité de I en entrée et en sortie des itérations n'est assurée que si I est vrai à l'initialisation, c'est-à-dire en entrée de la boucle.

2.2.3 Un invariant pour "*Chaînuple*"

En supposant le problème non trivial, la condition '$ne > 0$' est vérifiée, donc également la condition '$0 \leq 3 * ne - 1$' *(instanciation de '$ini \leq fin$')*.
Nous voulons unifier '$I(w \hookleftarrow fin + 1)$' avec '$B(w \hookleftarrow L + 1)$', la postcondition que nous voulons démontrer.
Nous avons donc '$ini = 0$' et '$fin = L$'.
En fait, pour obtenir un invariant, nous devons ajouter 2 conditions à la condition 'B', ce qui nous donne :

- $\forall \alpha, \quad 0 \leq \alpha \leq w - 1 \quad \Rightarrow \quad List(\alpha)/3 \ < \ \alpha/3 \qquad (B)$
- $\forall\, \alpha, \quad w \leq \alpha \leq L \ \Rightarrow \quad Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \qquad (C)$
- $\forall\, w_1, w_2 \in [0, L], \quad Triaj(w_1) = Triaj(w_2) \ \Rightarrow \ w_1 = w_2 \ \vee \ w_1/3 \neq w_2/3 \quad (A)$

La formule 'C' est ajoutée à 'B' pour obtenir la partie de l'invariant dépendante de l'indice de boucle.
'A' est un axiome insensible aux instructions de la boucle.
La formule 'B' exprime que le chaînage des occurrences déjà rangées a lieu dans le bon sens, c'est-à dire que les chaînes sont dirigées vers la tête de *List*. En effet, la case de *List* numérotée $\alpha/3$ pointe vers la case numérotée $List(\alpha)/3$.
'C' signifie :
juste avant l'itération 'w', une occurrence qui sera visitée ultérieurement lors de l'itération 'α' représentera le sommet $S =$ '$Triaj(\alpha)$'.
La dernière occurrence dejà visitée de ce sommet S est référencée par '$Tab(Triaj(\alpha))$', donc le dernier triangle visité attaché à S est : '$Tab(Triaj(\alpha))/3$'.
Or, les 2 occurrences de S 'α' et '$Tab(Triaj(\alpha))$' sont distinctes, puisque l'une a déjà été visitée et pas l'autre. Elles concernent donc des triangles différents.

Exécution de la preuve

Notre invariant 'Inv' est donc : $Inv = \ B \ \& \ C \ \& \ A$.

Calcul de '$Inv(w \hookleftarrow fin + 1)$'

Nous unifions ini avec 0, fin avec L. '$Inv(w \hookleftarrow fin + 1)$' est :

$\forall \alpha, \quad 0 \leq \alpha \leq L \quad \Rightarrow \quad List(\alpha)/3 \ < \ \alpha/3 \qquad \& \qquad (\ B(w \hookleftarrow L + 1)\ )$

$\forall \alpha, \quad L + 1 \leq \alpha \leq L \quad \Rightarrow \quad Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \qquad \& \qquad (\ C(w \hookleftarrow L + 1)\ )$

$\forall\, w_1, w_2 \in [0, L], \quad Triaj(w_1) = Triaj(w_2) \ \Rightarrow \ w_1 = w_2 \ \vee \ w_1/3 \neq w_2/3 \qquad (A)$

Calcul de '$Inv(w \hookleftarrow ini)$'

'$Inv(w \hookleftarrow ini)$' est :

$\forall \alpha, \quad 0 \leq \alpha \leq -1 \quad \Rightarrow \quad List(\alpha)/3 \ < \ \alpha/3 \qquad \& \qquad (\ B(w \hookleftarrow 0)\ )$

$\forall\, \alpha, \quad 0 \leq \alpha \leq L \quad \Rightarrow \quad Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \qquad \& \qquad (\ C(w \hookleftarrow 0)\ )$

$\forall\, w_1, w_2 \in [0, L], \quad Triaj(w_1) = Triaj(w_2) \ \Rightarrow \ w_1 = w_2 \ \vee \ w_1/3 \neq w_2/3 \qquad (A)$

Clairement, l'invariant est effectivement vérifié à l'initialisation de la boucle, c'est-à-dire comme précondition de "*Chaînuple*".

Preuve des prémisses de la règle '*Pour*'

Nous devons prouver les prémisses de la règle de déduction, à savoir ' $I \ \{Manip\} \ I(w \hookleftarrow w+1)$ ', soit :

$$Inv \ \{ \ s := Triaj(w); \ List(w) := Tab(s); \ Tab(s) := w \ \} \ Inv(w \hookleftarrow w+1)$$

Nous utilisons pour cela les règles suivantes:

- l'affectation : $S(x \hookleftarrow E) \ \{x := E\} \ S$
- la séquence : $Si \ E \ \{P\} \ F \ et \ F \ \{Q\} \ S \ , \ alors \ E \ \{P \ ; \ Q\} \ S$
- la règle de l'affectation $a(z)$:=E avec un tableau $a(z)$ soumis aux 2 contraintes :
  - z est une expression ne contenant aucune référence (dite "imbriquée") au tableau 'a'
  - la postcondition S ne contient aucune référence au tableau imbriquée.

  Si y désigne une expression quelconque et z une valeur d'index du tableau a telles que $a(y)$ représente une occurrence de $a(z)$, c'est-à-dire que $y = z$, alors :

  $S(\ a(y) \hookleftarrow E \ ) \ \{a(z)$:=$E\} \ S$ est un axiome.

La démonstration a lieu en 2 temps :

- calcul de la plus faible précondition qui assure la postcondition en sortie de programme (à l'aide des règles précédentes)
- vérification que cette préconditon est une conséquence de l'invariant.

Calcul de la plus faible précondition

La technique est de remonter le programme de la postcondition vers cette précondition par applications des règles correspondant aux instructions traversées. Le cheminement est donc inverse de l'exécution du programme.

$$Inv(w \hookleftarrow w+1)$$

$$\boldsymbol{\{Tab(s) := \ w\}}$$

$$\begin{array}{lll} & A \ \ \& \ \ B(w \hookleftarrow w+1) & \& \\ si \ Triaj(\alpha) \neq s & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \ ) & \& \\ si \ Triaj(\alpha) = s & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ w/3 \ < \ \alpha/3 \ ) & \& \end{array}$$

$$\boldsymbol{\{List(w) := \ Tab(s)\}}$$

$$\begin{array}{lll} & A & \& \\ \forall \alpha, \ \ 0 \leq \alpha \leq w-1 & \Rightarrow \ List(\alpha)/3 \ < \ \alpha/3 & \& \\ & \quad Tab(s \ )/3 < \ \alpha/3 & \\ si \ Triaj(\alpha) \neq s & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \ ) & \& \\ si \ Triaj(\alpha) = s & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ w/3 \ < \ \alpha/3 \ ) & \& \end{array}$$

$$\boldsymbol{\{s := \ Triaj(w)\}}$$

$$\begin{array}{lll} & A & \& \ P_1 \\ \forall \alpha, \ \ 0 \leq \alpha \leq w-1 & \Rightarrow \ List(\alpha)/3 \ < \ \alpha/3 & \& \ P_2 \\ & \quad Tab(Triaj(w) \ )/3 < \ \alpha/3 & \ P_3 \\ si \ Triaj(\alpha) \neq Triaj(w) & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ Tab(Triaj(\alpha))/3 \ < \ \alpha/3 \ ) & \& \ P_4 \\ si \ Triaj(\alpha) = Triaj(w) & (\ \forall \ \alpha, \ \ w+1 \leq \alpha \leq L \ \Rightarrow \ \ w/3 \ < \ \alpha/3 \ ) & \& \ P_5 \end{array}$$

La précondition trouvée se compose donc des 5 conditions, 'P_1 ' à 'P_5 '.

Vérifions qu'elle est une conséquence de l'invariant :

'Inv' est égal - rappelons-le - à 'B & C & A'

- $Inv \Rightarrow$ 'P_1 ': trivial car 'P_1 ' == 'A'
- $Inv \Rightarrow$ 'P_2 ': trivial car 'P_2 ' == 'B'
- $Inv \Rightarrow$ 'P_3 ': 'P_3 ' est la propriété 'C' instanciée pour w.
- $Inv \Rightarrow$ 'P_4 ': 'P_4 ' est une restriction de la propriété 'C'.
- $Inv \Rightarrow$ 'P_5 ': 'P_5 ' est une conséquence immédiate de 'A'.

Nous avons donc démontré que 'Inv' est un invariant pour le programme "*Chaînuple*". Cela veut dire que si l'initialisation vérifie cet invariant, alors cet invariant reste vrai en postcondition.
La règle de la boucle '*Pour*' nous permet ainsi de conclure que '$Inv(w \hookleftarrow L+1)$' est une postcondition vérifiée compte tenu que '$Inv(w \hookleftarrow 0)$' est vrai à l'initialisation de la boucle.
En particulier, nous avons : '$B(w \hookleftarrow L+1)$'.

2.3 Application au programme de recherche "*Déchaîne*"

La connaissance de propriétés caractérisant notre programme contribue à la vérification de sa correction. C'est le cas de la nécessaire preuve d'arrêt d'une boucle '*Tant_que*'.
Cela favorise également l'optimisation du programme. Notre exemple s'attache à la parallélisation de boucle.

2.3.1 Preuve d'arrêt

L'extraction d'une propriété du programme "*Chaînuple*" garantie par sa preuve est justifiée par la remarque suivante :
On ne peut extraire de la lecture du programme "*Déchaîne*" aucune information autorisant à conclure à l'arrêt (la terminaison) de celui-ci. Cette information est encapsulée dans l'initialisation de ce programme comme conséquence de la propriété '$B(w \hookleftarrow L+1)$' de "*Chaînuple*" dégagée ci-dessus.
Rappelons qu'un variant de boucle est une expression entière, positive à l'entrée et à l'intérieur de la boucle, qui décroît strictement à chaque itération de boucle, à l'exception éventuelle de la dernière. L'existence d'un variant assure la sortie de la boucle, sous réserve que le corps de la boucle termine. Remarquons qu'il n'y a pas de contrainte sur le variant en sortie de boucle. Dans notre exemple, nous allons montrer que le choix de l'expression 'w' comme variant nous sort d'embarras, grâce à la connaissance de la propriété '$B(w \hookleftarrow L+1)$'.

$$\forall \alpha, \quad 0 \leq \alpha \leq L \quad \Rightarrow \quad List(\alpha)/3 \quad < \quad \alpha/3 \qquad (\ B(w \hookleftarrow L+1)\ )$$

Règle de la preuve d'arrêt

Soit la boucle '*Tant_que Cond faire Manip*'
Si w représente le vecteur des variables et Var le variant :

$$Inv\ \&\ Cond\ \&\ (Var(w) = V_0)\ \ \{Manip\}\ \ \neg Cond\ \vee\ (0 \leq Var(w) < V_0) \quad \forall\ V_0\ \in\ Nat$$

Application à notre exemple

La règle se décline donc ici comme :

$$Inv\ \ \&\ \ \ \&\ \ (w = V_0)\ \ \ \{Déchaîne\}\ \ \ (w = -3)\ \ \vee\ \ (0 \leq w < V_0)\ \ \ \forall\ V_0\ \in\ Nat$$

La propriété '$B(w \hookleftarrow L+1)$' est une constante pour le programme de recherche, donc nous pouvons la choisir comme composante de l'invariant de celui-ci. Pour réaliser la preuve, nous allons enrichir '$B(w \hookleftarrow L+1)$' avec une nouvelle propriété, 'D' :

$$\forall \alpha,\ \ 0 \leq \alpha \leq L \quad \Rightarrow \quad (List(\alpha) = -3\ \ \vee\ \ 0 \leq List(\alpha)) \qquad (D)$$

'D' n'est pas isolément un invariant de "*Chaînuple*", mais une composante de l'invariant suivant:

$$\forall\alpha,\ \ 0 \le \alpha \le L \quad \Rightarrow \quad (List(\alpha) = -3 \ \ \vee \ \ 0 \le List(\alpha)) \\ \& \ \ (Tab(Triaj(\alpha)) = -3 \ \ \vee \ \ 0 \le Tab(Triaj(\alpha))$$

Cela exprime simplement que $List$ et Tab prennent leurs valeurs dans $\{-3\} \cup Nat$.
Nous laissons en exercice la démonstration de l'invariance de cette propriété pour "*Chaînuple*" et par conséquent sa validité en postcondition de ce programme et par suite à l'entrée de "*Déchaîne*", programme pour lequel 'D' est constant donc invariant.
Au cours du calcul de la plus faible précondition, la seule instruction de "*Déchaîne*" qui modifie la précondition est 'w:=$List(w)$' .
Donc, nous avons à vérifier:

$$B(w \hookleftarrow L+1)\ \&\ D\ \ \&\ \ (w = V_0)\ \ \{\ w := List(w)\ \}\ \ (w = -3)\ \ \vee\ \ (0 \le w < V_0)\ \ \ \forall\, V_0\ \in\ Nat$$

La plus faible précondition correspondante est:

$$(List(w) = -3)\ \ \vee\ (0 \le List(w) < V_0)\ \ \forall\, V_0\ \in\ Nat$$

Il est facile de voir que c'est une conséquence de la précondition effective:

$$B(w \hookleftarrow L+1)\ \ \&\ \ D\ \ \&\ \ (w = V_0)\ \ \ \Rightarrow\ \ (List(w) = -3)\ \ \vee\ \ (0 \le List(w) < V_0)\ \ \forall\, V_0\ \in\ Nat$$

Cela induit l'achèvement de la preuve.

2.3.2 Parallélisation de boucle?

'$B(w \hookleftarrow L+1)$' nous apprend également que nous pourrions utiliser comme variant '$w/3$', soit l'indice i de ligne du tableau $Tria$. La démonstration est calquée sur celle ci-dessus.
Cette information peut s'avérer précieuse pour l'optimisation des calculs.
Elle assure que les itérations de la boucle "*Déchaîne*" concernent des valeurs de i distinctes. Cette boucle sera donc susceptible de subir une parallélisation s'il n'y a pas de dépendances entre les calculs relatifs à des itérations distinctes (*loop carried dependences*). Examinons cela plus en détail.

2.3.3 Exploitation de la recherche

Nous relèverons 2 stratégies de tracé de l'algorithme.

Traitement personnalisé de chaque triangle
L'accès aux triangles attachés à un sommet S fixé a pour finalité le traitement d'informations concernant les triangles trouvés. Si ces informations caractérisent globalement le triangle, un tableau $Tri[i]$ les collectera. Si elles discriminent les différents sommets (ou angles) du triangle, le tableau sera du type $Tri[i,j]$. Pour simplifier l'écriture et sans nuire à la généralité du problème, nous regroupons les 2 termes en décidant que i représentera éventuellement le couple (i,j).
3 classes d'opérations sont envisageables:

- la donnée est indépendante de l'indice de boucle, *(w dans notre exemple)*
  l'instruction de calcul est de la forme $Tri[i] := Constante$
- la donnée est extraite ou calculée à partir des données relatives à ce triangle, elle ne dépend de l'indice de boucle qu'indirectement, par i
  l'instruction de calcul est de la forme $Tri[i] := E(i)$, E étant une expression (utilisant éventuellement fonctions et/ou tableaux)
- la donnée dépend directement de l'indice de boucle,
  l'instruction de calcul est de la forme $Tri[i] := E(i,w) = E'(w)$

Suposons que 2 itérations puissent délivrer la même valeur de i. C'est une hypothèse H contraire à la spécification de notre programme, mais non au texte-source de l'algorithme. Par conséquent, l'analyse des dépendances de données par le compilateur ne peut mettre au jour cette contrainte.
Dans l'un ou l'autre des 2 premiers contextes, l'action des 2 itérations est redondante, mais, sémantiquement

parlant, elles n'entrent pas en conflit. Dans cette situation boîteuse, on pourrait donc envisager que le compilateur soit en droit de paralléliser la boucle. En revanche, la dernière situation manifeste une concurrence dans l'écriture de $Tri[i]$. La parallélisation est alors interdite.
L'adjonction au texte-source d'une assertion garantissant la négation de l'hypothèse H permet la levée de cette interdiction quelle que soit la nature du traitement de données relatif aux triangles.

Affectation des résultats à l'indice de boucle

L'intérêt des calculs relatifs à chaque triangle est souvent circonscrit à un calcul global à la boule analysée, calcul qui les encapsulent tous. Sous contrôle de l'assertion qui garantit la bijection entre le numéro i du triangle de la boule et l'indice de la boucle w, ce dernier est en droit de "réquisitionner" le résultat du calcul de chaque itération. Dans ce cas, l'instruction dans l'itération d'indice w sera de la forme $Result[w] := E(i)$. Cette transformation étant justifiée par l'assertion, la nouvelle boucle est de fait parallélisable sans plus de référence à une assertion.
Après cette recherche/collecte d'information, la concaténation adéquate de ces résultats peut être opérée, par exemple grâce une technique dite de *réduction*, qui permet de paralléliser la concaténation [Wol96].

Nous illustrerons ces possibilités au cours des divers exemples présentés dans les 2 chapitres suivants.

Chapitre 3

Reformulation du programme en vue de sa parallélisation

3.1 Les programmes "*Train_de_wagons*" et "*Un_wagon*"

3.1.1 Pipeline logiciel

A la sous-section 2.3.2, en avançant la perspective de la parallélisation, nous avons négligé le type de boucle en jeu. En effet, dans notre boucle *Tant_que*, l'indice de la $i+1^{ème}$ itération de la boucle du programme "Recherche" est calculé au cours de la $i^{ème}$ itération. On peut donc envisager de "pipeliner" l'exécution de la boucle en plaçant l'instruction qui commande le calcul de cet indice avant les *Calculs sur i et j*: dès que cet indice est calculé, l'itération suivante peut commencer, sans attendre la fin des calculs de l'itération courante. L'implantation de ce pipeline pourrait être faite manuellement ou automatiquement. Une piste de recherche est de travailler à la détection systématique, donc automatisable, d'un contexte "pipelinable" selon ce modèle dans un programme. En tout état de cause, cela demande des dispositions particulières. Nous voudrions pour lors profiter du jeu d'instructions parallèles d'un langage comme *High Performance FORTRAN*, qui permet d'ordonner au compilateur la mise en oeuvre d'un mécanisme de parallélisation.

3.1.2 Présentation des programmes

Un autre argument nous pousse à revoir l'architecture logicielle: c'est la possibilité d'améliorer encore le programme de recherche. Nous aimerions que les cases-mémoire successives accédées dans *List* (nous appellerons *wagon* un tel intervalle de *List* relatif à un même sommet) soient consécutives. De cette manière, disparaîtrait l'indirection exigée par le calcul de l'indice w (w:=*List*(w)). De plus, la localité spatiale lors de l'accès à *List* serait améliorée.
Dans notre programme de rangement initial, le rang d'un élément de *List* dénote le rang dans *Triaj* d'une occurrence de sommet en cours de traitement. Par le nouveau procédé, la nature du tableau *List* a changé: c'est le contenu de *List* et non son index qui référencera une position dans *Triaj*.
Cela devient possible si nous acceptons de construire notre base de données en balayant *Triaj* en une seconde passe, afin de comptabiliser pour chaque sommet le nombre de triangles attachés à lui.
Dans ce but, un tableau *Tete*(S) est construit en 2 étapes:

- lors d'un premier balayage de *Triaj*, *Der*(S) repère le nombre d'occurrences du sommet S.
- muni de cette information, *Der*(S) enregistre alors pour chaque sommet S la place réservée dans (la nouvelle version de) *List* à la dernière occurrence de S rencontrée lors du balayage. Cette case est la dernière de la suite des cases contiguës de *List* référençant la présence de S. De plus, l'ordre relatif de ces suites correspond à l'ordre croissant des sommets. *Tete* se déduit naturellement de *Der*. Les 2 tableaux *List* et *Tete* constituent la nouvelle structure duale de *Tria*.

Cela dirige l'exécution du programme de rangement. Celui-ci manipule un tableau local *courant*(S) qui donne la position de la prochaine case libre à affecter à S dans *List*. L'initiale de l'identificateur '*courant*' est en minuscule pour signifier que '*courant*' est une variable dans la boucle "*Keuleuleu*", fonction de w .
La nouvelle version des programmes est la suivante:

rangement **recherche**

Initialisation *Initialisation*

```
Pour S = 0, ns-1
    Der(S) := 0                                  Der(S) := Tete(S+1) - 1
Fin pour

    Der(-1) := -1   /* <-- absorbe le cas-limite 'S = 0' lors de la recherche  */
```

Train_de_wagons *Un_wagon*

```
Pour w = 0, L                                 Pour k = Tete(S), Der(S)
    s         := Triaj(w)
    Der(s)    := Der(s)+1                         w := List(k)
Fin pour
                                          /*  S est le sommet local j de l'élément i   */
Pour s = 1, ns-1
    Der(s)    := Der(s-1) + Der(s)                  i := w/3
Fin pour                                         j := w mod 3

Tete(S) := Der(S-1) + 1                       Calculs sur  i et j ...

Pour s = 0, ns-1
    courant(s) := Tete(s)                     Fin pour
Fin pour

Pour  l = 0, L
    List(l) := -1
Fin pour

Pour w = 0, L               /* boucle ''Keuleuleu'' */
    s := Triaj(w)
    List(courant(s)) := w
    courant(s) := courant(s)+1
Fin pour
```

3.2 Procédure générale pour trouver un invariant encapsulant une propriété

A la section 3, nous avons démontré que la formule 'A *&* C *&* B' était un invariant de "*Chaînuple*", car nous voulions vérifier la validité de la propriété 'A' en postcondition. Cela soulève 2 questions.

- comment construire cet invariant?
- comment démontrer cet invariant?

Commençons par le $2^{ème}$ point.

3.2.1 Petit lemme

L'hypothèse 'I invariant & $(I \Rightarrow J)$' n'autorise pas à conclure à l'invariance de J. En revanche, nous disposons du résultat qui suit:

$I\ \{Manip\}\ I(w \hookleftarrow w+1)\ \ \&\ \ J_0\{Manip\}J(w \hookleftarrow w+1)\ \ \&\ \ (I\ \&\ J \Rightarrow\ J_0)$
$\Rightarrow\ \ \ I\ \&\ J\ \{Manip\}\ I(w \hookleftarrow w+1)\ \&\ J(w \hookleftarrow w+1)$

Dans ce contexte, I *&* J est un invariant de "*Manip*".

Utilité

Ce lemme démontre comment l'on peut enrichir l'invariant par étapes au lieu de démontrer un invariant complexe d'un seul bloc.
En effet, soit I un invariant de notre programme ou de notre portion de programme. Nous voulons enrichir I avec J. La démarche est la suivante : nous calculons J_0, la plus faible précondition associée à J. Il nous suffit alors de démontrer '$(I \ \& \ J \Rightarrow \ J_0)$'.
Dans l'exemple de "*Déchaîne*", '$A\&C$', puis '$A\&C\&B$' sont des enrichissements successifs de 'A' et nous aurions pu mobiliser ce lemme dans notre démarche de preuve.

3.2.2 Construire l'invariant

La construction de cet invariant peut se réaliser selon le procédé itératif suivant:
soit P la propriété que nous visons, et I l'invariant connu au début de notre recherche. on calcule la plus faible précondition P_0 de la propriété P. 2 cas se présentent :

- nous pouvons démontrer '$I \ \& \ P \ \Rightarrow \ P_0$'. D'après le lemme, '$I \ \& \ P$' est un invariant (CQFD).
- cas contraire : nous ne pouvons pas faire la démonstration, soit parce que le pas de la prémisse au but est trop grand, soit parce que la proposition ci-dessus est fausse. Il faut essayer de déduire un prédicat minimal P_1 à ajouter à notre invariant en construction tel que '$I \ \& \ P_1 \ \& \ P \ \Rightarrow \ P_0$' (γ). Dans le cas le moins favorable, on choisit $P_1 = P_0$.
  On réitère le procédé appliqué à P avec P_1 : on calcule la plus faible précondition P_{10} de la propriété P_1 et on regarde si on est capable d'inférer '$I \ \& \ P_1 \ \Rightarrow \ P_{10}$'. Si oui, '$I \ \& \ P_1$' est un invariant et par ricochet 'γ' entre dans le contexte du lemme pour inférer l'invariance de P. Si non, on itère le procédé.

3.3 Preuve d'une propriété de "*Train_de_wagons*"

Une preuve d'arrêt de "*Un_wagon*" est superflue, car nous avons une boucle '*Pour*'.
Ce programme recherche les occurrences d'un sommet dans $Triaj$ en balayant un seul wagon. Nous désirons prouver, comme auparavant, que toutes les cases de $Triaj$ séjournant dans ce wagon concernent des triangles différents (indice i).
A l'inverse de "*Déchaîne*", les triangles sont retrouvés dans l'ordre croissant.
La propriété désirée s'exprime comme suit:

$$\forall \lambda, \quad Tete(S) < \lambda \leq Der(S) \quad \Rightarrow \quad List(\lambda - 1)/3 \quad < \quad List(\lambda)/3 \qquad (WG_s)$$

En cours de boucle, la propriété s'écrit:

$$\forall \lambda, \quad Tete(S) < \lambda \leq courant(S) - 1 \quad \Rightarrow \quad List(\lambda - 1)/3 \quad < \quad List(\lambda)/3 \qquad (WG)$$

En effet, nous montrerons qu'à la sortie de boucle :

$$w = L + 1 \quad \Rightarrow \quad \forall S, \ courant(S) - 1 = Der(S) \qquad (CDR)$$

3.3.1 Notations

Nous utiliserons les notions de la théorie des ensembles.
Ainsi, l'expression $card(_{[0,w-1]<}Triaj^{-1}[S])$ signifie : " le cardinal de l'ensemble des antécédants du sommet S par la restriction de la fonction '$Triag$' au domaine $[0, w-1]$. Il s'agit donc ici du nombre de cases de $Triaj$ occupées par S parmi les $(w-1)$ premières.

Nous utiliserons la notation suivante : $Tete(S) = Der(S-1) + 1 \qquad (TD)$

3.3.2 Une base de prédicats

Ces prédicats sont les briques constitutives des propriétés que nous allons démontrer.

$$Der(S) = Der(S-1) + card(Triaj^{-1}[S]) \qquad (DR)$$

$$courant(S) = Tete(S) + card(_{[0,w-1]<}Triaj^{-1}[S]) \qquad (CT)$$

$$courant(S) \in [Tete(S), Tete(S+1)] \qquad (TCT)$$

$$courant(S) \in]Tete(S_1), Tete(S_1+1)[\Rightarrow \quad S = S_1 \qquad (T^+CT)$$

$$courant(Triaj(w)) \leq Der(Triaj(w)) \qquad (CW)$$

$$\forall \lambda, \quad 0 \leq \lambda \leq L \quad \Rightarrow \quad List(\lambda) < w \qquad (LW)$$

$$\forall \lambda, \quad Tete(S) \leq \lambda < courant(S) \quad \Rightarrow \quad Triaj(List(\lambda)) = S \qquad (TS)$$

$$\forall \lambda \mid 0 \leq \lambda \leq L, \quad Tete(Triaj(w)) \leq \lambda < courant(Triaj(w)) \quad \Rightarrow \quad List(\lambda)/3 < w/3 \qquad (LW_3)$$

'*courant*' désigne le tableau de pointeurs vers chaque prochaine case libre attribuée aux sommets respectifs de '*List*', au cours du rangement.
'DR' et 'CT' reflètent la définition des tableaux *Der* et *courant*.
'TCT' : en fin de course, '$courant(S)$' aboutit en '$Tete(S+1)$'; à ce moment, il est obsolète, c'est-à-dire que le compartiment qu'il cible ne sera plus affecté au sommet S, il appartient à $S+1$.
'T^+CT' : est une conséquence directe de 'TCT'.
'CW' : à l'itération 'w', une occurence du sommet '$Triaj(w)$' vient d'être découverte; elle sera donc placée dans le wagon affecté à ce sommet dont le dernier compartiment est '$Der(Triaj(w))$'.
'LW' : à l'itération 'w', les valeurs de *List* sont les numéros des itérations précédentes, ou la valeur d'initialisation (-3).
'TS' : dans les compartiments déjà occupés du wagon affecté à S, les occurences rencontrées sont obligatoirement représentatives de S.
'LW_3' : l'ordre d'occupation dans un wagon s'aligne sur l'ordre croissant des numéros de triangle; le triangle '$w/3$' visité à l'itération 'w' porte un numéro supérieur au numéro '$List(\lambda)/3$' d'un triangle quelconque traité précédemment pour ce wagon, qui est celui réservé au sommet '$Triaj(w)$'.

3.3.3 Exercices

'DR' et 'CT' sont 2 invariants de "*Keuleuleu*". Afin d'alléger notre propos et parce que cela ne présente pas de difficultés, nous laisserons le lecteur démontrer 'DR', propriété issue des boucles du programme de rangement qui préparent le contexte de la boucle principale, "'*Keuleuleu*".
Dans ce but, nous pouvons vérifier -par la méthode pratiquée plus haut- que '$Der(S) = card(Triaj^{-1}[S])$' est une postcondition vérifiée en sortie de la $1^{ère}$ boucle de "'*Train_de_wagons*", d'où il découle que 'DR' est vrai en entrée de "'*Keuleuleu*".
Cette proposition est une constante lors de l'exécution de "'*Keuleuleu*". Remarquons que, dans ce cas, les notions d'invariance et de vérité se confondent.
Nous laissons également le lecteur vérifier la vérité de 'CT' en entrée de "'*Keuleuleu*" et son invariance lors de sa traversée.

3.3.4 Preuves intermédiaires

Preuve de 'CDR'

De la vérité de 'CT', 'DR' et 'TD' en sortie de "'*Keuleuleu*", on infère 'CDR', propriété mentionnée et exploitée plus haut.

Cinq propositions logiques

$$DR \;\&\; CT \;\Rightarrow\; TCT \qquad (P_1)$$

$$TCT \;\Rightarrow\; T^{+}CT \qquad (P_2)$$

$$DR \;\Rightarrow\; Der\ croissant \;\&\; Tete\ croissant \qquad (P_3)$$

$$Tete\ croissant \;\&\; TCT \;\Rightarrow\; courant\ croissant \qquad (P_4)$$

$$CT \;\&\; DR \;\Rightarrow\; CW \qquad (P_5)$$

Les démonstrations de P_1 à P_4 sont laissées en exercice.

Démonstration de 'P_5'

Nous laissons en exercice la démonstration de la formule suivante :

$CT \;\&\; DR \;\Rightarrow\; courant(S) = Der(S) - card({}_{[w,L]<}Triaj^{-1}[S]) + 1.$

Instanciée avec '$S = Triaj(w)$', la conclusion donne :

$courant(Triaj(w)) = Der(Triaj(w)) - card({}_{[w,L]<}Triaj^{-1}[Triaj(w)]) + 1$

Or, $w \in {}_{[w,L]<}Triaj^{-1}[Triaj(w)]$, donc le cardinal de cet ensemble est strictement positif, ce qu'exprime 'CW'.

'LW' est un invariant de "*Keuleuleu*"

$$\forall\lambda,\ \ 0 \le \lambda \le L \quad\Rightarrow\quad List(\lambda) < w+1 \qquad (\ LW(w \hookleftarrow w+1)\ )$$

$$\{courant(s) := courant(s) + 1\}$$

$$\{List(courant(s)) := w\}$$

$$\forall\lambda \neq courant(s),\ \ 0 \le \lambda \le L \quad\Rightarrow\quad List(\lambda) < w+1$$

$$\lambda = courant(s) \ \ \&\ \ 0 \le \lambda \le L \quad\Rightarrow\quad w < w+1$$

$$\{s := Triaj(w)\}$$

$$\forall\lambda \neq courant(Triaj(w)),\ \ 0 \le \lambda \le L \quad\Rightarrow\quad List(\lambda) < w+1$$

$$\lambda = courant(Triaj(w)) \ \ \&\ \ 0 \le \lambda \le L \quad\Rightarrow\quad w < w+1$$

'LW' induit la première assertion de la plus faible précondition trouvée, la seconde étant triviale.

'DR & CT & TS' est un invariant de "*Keuleuleu*"

$$\forall\lambda,\ \ Tete(S) \le \lambda < courant(S) \quad\Rightarrow\quad Triaj(List(\lambda)) = S \qquad (\ TS(w \hookleftarrow w+1))$$

$$\{courant(s) := courant(s) + 1\}$$

Si $s \neq S$ $\qquad \forall\lambda,\ \ Tete(S) \le \lambda < courant(S) \quad\Rightarrow\quad Triaj(List(\lambda)) = S$

Si $s = S$ $\qquad Tete(s) \le \lambda \le courant(s) \quad\Rightarrow\quad Triaj(List(\lambda)) = S$

$$\{List(courant(s)) := w\}$$

Si $s \neq S$ & $\lambda \neq courant(s)$

$$\forall\lambda,\ \ Tete(S) \leq \lambda < courant(S) \ \ \Rightarrow \ \ Triaj(List(\lambda)) = S$$

Si $s \neq S$ & $\lambda = courant(s)$

$$Tete(S) \leq courant(s) < courant(S) \ \ \Rightarrow \ \ Triaj(w) = S$$

Si $s = S$

$$\forall\lambda,\ \ Tete(s) \leq \lambda < courant(s) \ \ \Rightarrow \ \ Triaj(List(\lambda)) = s$$

$$Tete(s) \leq courant(s) \ \ \Rightarrow \ \ Triaj(w) = S$$

$$\{s := Triaj(w)\}$$

Si $Triaj(w) \neq S$ & $\lambda \neq courant(Triaj(w))$

$\forall\lambda,\ \ Tete(S) \leq \lambda < courant(S) \ \ \Rightarrow \ \ Triaj(List(\lambda)) = S \quad (1)$ (1)

Si $Triaj(w) \neq S$ & $\lambda = courant(Triaj(w))$

$Tete(S) \leq courant(Triaj(w)) < courant(S) \ \ \Rightarrow \ \ Triaj(w) = S$ (2)

Si $Triaj(w) = S$

$\forall\lambda,\ \ Tete(Triaj(w)) \leq \lambda < courant(Triaj(w)) \ \ \Rightarrow \ \ Triaj(List(\lambda)) = Triaj(w)$ (3)

$Tete(Triaj(w)) \leq courant(Triaj(w)) \ \ \Rightarrow \ \ Triaj(w) = S$ (4)

Les cas (1) et (3) sont des conséquences de 'TS'.
Le cas (2) est impossible, le contexte est contradictoire. Preuve :
Posons $Triaj(w) = T$.
D'après 'TCT', assertion elle-même induite par 'DR & CT' , $courant(S) - 1 < Tete(S+1)$. Couplée avec la prémisse de (2), cette formule conduit à : $courant(T) \in [Tete(S), Tete(S+1)[$. (α)
Par ailleurs, 'CW' et 'TD' nous donnent : $courant(T) < Tete(T+1)$, alors que 'TCT' impose :
$Tete(T) \leq courant(T)$, soit finalement : $courant(T) \in [Tete(T), Tete(T+1)[$ (β).
Compte-tenu de la croissance de Tete, (α) et (β) imposent $T = S$, c'est-à-dire $Triaj(w) = S$, ce qui est contradictoire avec le contexte.

Enfin, la conclusion du cas (4) est incluse dans son contexte.
L'invocation du lemme nous permet de conclure.

Preuve de la proposition: 'A & TS & LW $\Rightarrow$ LW_3'

Instancions 'TS' avec '$S = Triaj(w)$' :

$$\forall\lambda\ |0 \leq \lambda \leq L,\ \ Tete(Triaj(w)) \leq \lambda \leq courant(Triaj(w)) - 1 \ \ \Rightarrow \ \ Triaj(List(\lambda)) = Triaj(w)$$

La prise en compte de 'LW' permet d'appliquer l'axiome 'A', qui règle la configuration de $Triaj$:

$$\forall\lambda \mid 0 \leq \lambda \leq L,\ \ Tete(Triaj(w)) \leq \lambda \leq courant(Triaj(w)) - 1 \ \ \Rightarrow \ \ List(\lambda)/3 < w/3 \qquad \text{CQFD}$$

Preuve de l'invariant final 'A & DR & CT & TS & LW & WG'

Nous sommes maintenant armés pour faire la preuve de cet invariant, que nous baptisons Inv.
Nous savons déjà que 'A & DR & CT & TS & LW' est un invariant de "*Keuleuleu*", et nous voulons l'enrichir avec 'WG', à l'aide du lemme.

$$\forall\lambda, \quad Tete(S) < \lambda < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3 \qquad (WG(w \hookleftarrow w+1))$$

$$\{courant(s) := courant(s) + 1\}$$

Si $s \neq S$

$$\forall\lambda, \quad Tete(S) < \lambda < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3$$

Si $s = S$

$$\forall\lambda, \quad Tete(s) < \lambda \leq courant(s) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3$$

$$\{List(courant(s)) := w\}$$

Si $s \neq S$ & $\lambda \neq courant(s)$ & $\lambda - 1 \neq courant(s)$

$$Tete(S) < \lambda < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3$$

Si $s \neq S$ & $\lambda = courant(s)$

$$Tete(S) < courant(s) < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3$$

Si $s \neq S$ & $\lambda - 1 = courant(s)$

$$Tete(S) < courant(s) + 1 < courant(S) \quad \Rightarrow \quad w/3 \quad < \quad List(\lambda)/3$$

Si $s = S$

$$\forall\lambda, \quad Tete(s) < \lambda < courant(s) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3$$

$$Tete(s) < courant(s) \quad \Rightarrow \quad List(courant(s)-1)/3 \quad < \quad w/3$$

$$\{s := Triaj(w)\}$$

Si $Triaj(w) \neq S$ & $\lambda \neq courant(Triaj(w))$ & $\lambda - 1 \neq courant(Triaj(w))$

$$Tete(S) < \lambda < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3 \qquad (1)$$

Si $Triaj(w) \neq S$ & $\lambda = courant(Triaj(w))$

$$Tete(S) < courant(Triaj(w)) < courant(S) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3 \qquad (2)$$

Si $Triaj(w) \neq S$ & $\lambda - 1 = courant(Triaj(w))$
$Tete(S) < courant(Triaj(w)) + 1 < courant(S) \quad \Rightarrow \quad w/3 \quad < \quad List(\lambda)/3 \qquad (3)$

Si $Triaj(w) = S$

$$\forall\lambda, \quad Tete(Triaj(w)) < \lambda < courant(Triaj(w)) \quad \Rightarrow \quad List(\lambda-1)/3 \quad < \quad List(\lambda)/3 \qquad (4)$$

$$Tete(Triaj(w)) < courant(Triaj(w)) \quad \Rightarrow \quad List(courant(Triaj(w))-1)/3 \quad < \quad w/3 \qquad (5)$$

Les propositions (1) et (4) sont des restrictions de 'WG'.

Chacun des contextes des propositions (2) et (3) sont contradictoires.
Preuve: voir ci-dessus le cas (2) de la preuve d'invariance de 'DR & CT & TS'.

La propriété (5) est une instanciation (restreinte) de 'LW_3' avec '$\lambda = courant(Triaj(w)) - 1$'.
Or, 'LW_3' est une conséquence de l'invariant en cours de preuve, comme il a été démontré juste au dessus.

3.3.5 Preuve finale

Y-a-t'il des triangles jumeaux dans les *wagons*?
Nous avons vérifié que la plus faible précondition trouvée est une conséquence de l'invariant final Inv en cours de preuve, ce qui autorise le lemme à estampiller effectivement cette formule comme invariant de "*Keuleuleu*". Nous laissons le lecteur vérifier que cet invariant est vrai à l'entrée de "*Keuleuleu*", ce qui est une condition d'application de la règle de déduction de la boucle '*Pour*'. La proposition 'WG_s' est donc valide en sortie de "*Keuleuleu*", aucun triangle n'a de jumeau dans l'un des *wagons* où il est présent.
Remarquons au passage que notre invariant concentre les propriétés essentielles de "*Train_de_wagons*".
Ainsi, à la fin du rangement, la proposition 'CDR' est vérifiée et 'TS' se réécrit en :

$$\forall \lambda, \quad Tete(S) \leq \lambda \leq Der(S) \quad \Rightarrow \quad Triaj(List(\lambda)) = S \qquad (TS_s)$$

Cette formule exprime donc que le wagon affecté à S dans $List$ référence exclusivement des occurrences de S dans $Triaj$, alors que 'WG_s' indique que toutes ces occurrences sont distinctes. Finalement, nous pouvons en déduire que toutes les occurrences des sommets sont bien rangées. Cela prouve la correction de notre programme de rangement relativement à la fonction qu'on en attend. Ainsi, lors de l'exécution du programme de recherche, 'TS_s' nous garantit que la référence (i, j) concerne effectivement le sommet S sélectionné.

3.4 Application

3.4.1 La visite des triangles est strictement monotone !

Notre but ultime est de démontrer que l'indice i de ligne de la place w de $Triaj$ accédée lors de la recherche -indice qui représente aussi le numéro d'un triangle possédant le sommet choisi- est différent à chaque itération de la boucle de "*Un_wagon*". Plus précisément, cet indice est strictement croissant.
Cette propriété, notée 'UW', peut être exprimée sous la forme :

$$WG_s \ \ \& \ \ k \in]Tete(S), Der(S)] \ \ \& \ \ (i = List(k-1)/3) \ \& \ \ (i = i_0)$$

$$\{\text{Un\_wagon}\}$$

$$(i > i_0) \quad \forall\, i_0 \ \in \ Nat$$

Nous avons installé l'invariant '$i = List(k-1)/3$' en précondition. (la vérification de l'invariance pour "Un_wagon" est laissée au lecteur).
Le calcul de la plus faible précondition relative à la postcondition de 'UW' donne :

$List(k)/3 > i_0, \quad \forall\, i_0 \ \in \ Nat$

Par la précondition de 'UW', nous transformons cette formule :

$\forall k \in]Tete(S), Der(S)], \quad List(k)/3 > List(k-1)/3.$

Ce prédicat n'est autre que 'WG_s'. (CQFD)

3.4.2 Optimisation des maillages

Motivations

Nous introduisons maintenant 3 exemples d'applications prises en charge par l'architecture logicielle que nous avons relatée. Elles suivent la stratégie d'*affectation des résultats à l'indice de boucle* telle que décrite ci-dessus à la sous-section 2.3.3. Rappelons que l'assertion caractéristique justifie cette stratégie qui la prend implicitement en compte dans la rédaction de l'algorithme. De ce fait, l'adjonction de cette assertion au nouveau programme n'a plus sa raison d'être.

Bougé de points vers le barycentre des sommets

La technique illustrée ici [FG99] vise à *lisser* le maillage afin d'obtenir un résultat de qualité supérieure au regard de critères variables spécifiques à l'application.
Soit P un sommet du maillage T, un élément de *la boule de* P est tout triangle de T comptant P parmi ses sommets. Dans ce qui suit, on ne considère que le cas d'une boule fermée : le sommet P n'est pas sur la frontière du maillage.
Une des méthodes dites du *bougé de points* consiste à déplacer le point P du maillage vers un point G_P, barycentre pondéré des sommets de sa boule. Pour simplifier la formulation, nous retenons ici l'hypothèse équipondérée.
Soit O l'origine du repère utilisé. Le calcul du barycentre, à implanter dans "*Un_wagon*" pour chaque sommet S concerné par le bougé, se déroule comme suit :

- repérage des t triangles constituant la boule de S
- pour chacun de ces triangles ("Calculs sur i et j")
  - repérage des 2 sommets S_1 et S_2 distincts de S par lecture du tableau $Tria$
  - sommation: $\overrightarrow{OS_1} + \overrightarrow{OS_2}$
- sommation globale sur l'ensemble des triangles et division par $2 * t$, ce qui fournit $\overrightarrow{OG_S}$.

La parallélisation concerne les calculs distribués sur les triangles. La sommation globale a lieu dans un $2^{ème}$ temps.

Bougé de points vers le centre de gravité

Une version différente de cette méthode consisterait à calculer le barycentre de chaque triangle de la boule, puis le barycentre global. Dans ce cas, nous obtenons le centre d'inertie de la boule, c'est-à-dire le barycentre surfacique ou centre de gravité, le premier calcul privilégiant les zones où se concentrent les sommets.

Résolution des équations de Maxwell

Un autre exemple d'application de "*Un_wagon*" est le suivant :
Selon un schéma de résolution des équations de Maxwell, on désire calculer $Q(s)$, la valeur de l'inconnue aux sommets, à partir des valeurs $Q(B_i)$ calculées au barycentre de chacun des triangles de la boule.
$Q(s)$ se calcule comme la moyenne, pondérée par les aires, des valeurs $Q(B_i)$.

Les programmes sont présentés ci-dessous. En raison d'une similitude structurelle, nous présentons dans le même algorithme le calcul du centre d'inertie de la boule et celui des équations de Maxwell.

Les programmes

```
/*  Calcul du barycentre des sommets de la boule  */


      Der(S) = Tete(S+1)-1
      Pour k = Tete(S), Der(S)          /*  boucle parallèle  */
         w := List(k)
         i := w/3
         j := w mod 3

  Somme(k) := (Abcisse(Tria(i,(j+1) mod 3)) + Abcisse(Tria(i,(j-1) mod 3)),
              Ordonnee(Tria(i,(j+1) mod 3))+ Ordonnee(Tria(i,(j-1) mod 3)))
    Fin pour

    Barycentre(S) := (0,0)
    Pour k = Tete(S), Der(S)
        Barycentre(S) := Barycentre(S) + Somme(k)
    Fin pour
    Barycentre(S) := Barycentre(S)/(2*(Tete(S+1)-Tete(S)))


/*              Calcul du centre d'inertie de la boule  ... et ...              */
/*  calcul impliqué dans le schéma de résolution des équations de Maxwell  */

    Der(S) = Tete(S+1)-1
    Pour k = Tete(S), Der(S)          /*  boucle parallèle  */
         w := List(k)
         i := w/3
         j := w mod 3

         barycentr(k) := (0,0)
         Pour j = 0, 2
           x(j) := Abcisse(Tria(i,j))
           y(j) := Ordonnee(Tria(i,j))
         Fin pour

        Pour j = 0, 2
            barycentr(k) := barycentr(k)+ (x(j), y(j))
            Aire(k) = Aire(k) + x(j)*(y(j+1 mod 3) - y(j-1 mod 3))
        Fin pour
        barycentr(k) := barycentr(k)/3
    Fin pour

    Centre_boule(S) := (0,0)
    Q(S) := 0
    Aire_boule := 0
    Pour k = Tete(S), Der(S)
        Centre_boule(S) := Centre_boule(S) + barycentr(k)
        Q(S) :=  Q(S) + Aire(k)*Q(barycentr(k))
        Aire_boule := Aire_boule + Aire(k)
    Fin pour

    Centre_boule(S) := Centre_boule(S)/(Tete(S+1)-Tete(S))
    Q(S) :=  Q(S)/ Aire_boule
```

3.5 Optimisation et modèle *Entité-Relation*

La propriété caractéristique exploitée dans les 2 sections précédentes s'inscrit dans un contexte plus général. Elle ne fait rien de plus qu'exprimer que la collection des couples $(triangle, sommet(triangle))$ est un ensemble, i.e. ne contient pas de doublon. Autrement dit, l'association entre les triangles et les sommets constitue une *relation* au sens ensembliste. La propriété elle-même exprime la contrainte d'intégrité élémentaire propre à une relation dans un système de gestion de base de données relationnelles (SGBDR), contrainte qui impose l'absence de doublons dans l'ensemble des tuples d'une relation. Notre relation a néanmoins des spécificités : un triangle est en relation avec exactement 3 sommets. La réciproque est aléatoire.
On sait [Car90] qu'un schéma *entité-relation* peut être modélisé par des *schémas de relation*. Chaque entité est représentée par la liste de ses attributs et les relations entre entités deviennent des relations entre les clefs primaires de ces entités. Les entités n'ayant qu' un attribut n'ont pas de relation associée. C'est le cas dans notre exemple pour les triangles comme pour les sommets.
Le schéma de la relation $triangle \leftrightarrow sommet$, que nous baptisons la relation *angle* est composé de 3 attributs : le numéro de triangle, le numéro de sommet et le numéro local de sommet. L'ensemble des tuples est donc de la forme (i, S, j). i et S sont les *clefs* associées respectivement aux entités *triangle* et *sommet*. L'attribut j, qui ne peut constituer une *clef*, joue ici un rôle secondaire.
Le tableau $Tria$ est un classement des données selon les triangles, alors que le couple $(Tete,\ List)$ est un classement selon les sommets, i.e. selon la relation inverse $angle^{-1}$. La dyssymétrie est l'expression, pour cette relation, de la cardinalité aléatoire des triangles attachés à un sommet donné, ce qui appelle la linéarisation des données en lieu et place d'un rangement en matrice. Sous cet angle de vue, il apparait que le tableau $List$ aurait pu tout aussi bien contenir les valeurs i des triangles plutôt que l'indice w égal à $3*(i-1)+j$. L'intérêt de w est de maintenir la correspondance bijective entre les occurrences de la relation *angle* et de son inverse. w joue ainsi le rôle d'une *clef* commune aux 2 relations. Remarquons que l'index de $List$ est une seconde *clef* de $angle^{-1}$, *clef* non partagée avec *angle*.
Finalement, la faculté de parallélisation de la boucle de recherche/exploitation apparait comme attachée à la notion de *relation ensembliste*. Un ensemble de contraintes suffisant pour la parallélisation est:

- les opérations dans les tuples de la relation sont en lecture seule
- une valeur lue lors d'une itération est créee avant la boucle ou dans cette itération
- l'indice de boucle est une *clef* de la relation ou en relation injective avec une telle *clef*
- chaque index d'un tableau en écriture lors de la boucle est en relation injective avec l'indice de boucle

Ce qui vaut pour une relation vaut évidemment au même titre pour la relation inverse.
L'exemple que nous avons traité pourrait être décliné en dimension 3. Les tétraèdres d'un maillage sont *en relation avec* leurs sommets, leurs faces, leurs arêtes.
Un système de base de données ou tout protocole qui s'attacherait à respecter ces exigences serait concerné par la problématique que nous présentons.

Chapitre 4

Quelques exemples de programmes annotés avec une assertion

Ce chapitre illustre la stratégie que nous avons appelée *traitement personnalisé* en sous-section 2.3.3, alternative à celle utilisée pour les applications de type *maillage* du chapitre 2: maintenant, les boucles de recherche devront être estampillées par une assertion qui lève toute condition de dépendance entre les itérations.

4.1 Modélisation des arêtes d'un maillage

4.1.1 La structure de données

Soit un nuage de n points dans le plan, numérotés dans l'intervalle $[0..n-1]$. Notre programme se compose de 2 étapes. En premier lieu, nous cherchons à numéroter les arêtes orientées (soit des vecteurs) constituées en reliant 2 points distincts. Les vecteurs de longueur nulle sont exclus de l'ensemble que nous construisons, pour éviter les singularités géométriques. Dans un second temps, nous construirons le tableau des coordonnées de ces arêtes. Notre modélisation se présente comme suit:
Nous dressons une matrice *Arete* de dimensions $n*n$, où l'indice i de ligne désigne le numéro de l'origine, l'indice j de colonne le numéro de l'extrêmité, que nous appelons *cible*. Nous numérotons les cases de 0 à n^2-n-1, dans l'ordre croissant des lignes prioritairement, secondairement des colonnes. Les cases de la diagonale, invalides, doivent être sautées. Cela délivre la numérotation des arêtes.
Voici les formules mathématiques qui expriment ce modèle:

$$Arete[i,j] = i*(n-1) + j - \delta[j-i]$$

$$i = Arete[i,j]/(n-1)$$

$$j = Arete[i,j] mod (n-1) + \delta[j-i]$$

$$avec \qquad \delta[j-i] = Arete[i,j]/n - Arete[i,j]/(n-1) + 1$$

$\delta[\alpha]$ vaut 1 si $\alpha > 0$, 0 sinon. Montrons la validité des 4 égalités ci-dessus.

Preuves
La $1^{ère}$ formule exprime notre choix de numérotation et ne nécessite pas de preuve.

Preuve des $2^{ème}$ et $3^{ème}$ formules:

- lorsque '$j < n-1$' est vrai, la démonstration est triviale
- lorsque '$j = n-1$', $\delta[j-i]$ *vaut* 1, car '$i \neq j \Rightarrow i < j$'. Donc, '$j - \delta[j-i]$', strictement inférieur à $n-1$, constitue toujours le reste de la division entière de $Arete[i,j]$ par $n-1$. La preuve de la $3^{ème}$ formule suit immédiatement.

Preuve de la $4^{ème}$ formule :
Posons : $ar = Arete[i,j]$. Il est aisé de vérifier :

- lorsque ar est le numéro d'une case précédant la case diagonale,
  $i*(n-1) \leq ar \leq i*n-1$,
  $ar/(n-1) = i$, $ar/n = i-1$ et $\delta[j-i] = 0$
- Lorsque ar est le numéro d'une case qui suit la case diagonale,
  $i*n \leq ar \leq (i+1)*(n-1)-1$,
  $ar/(n-1) = i$, $ar/n = i$ et $\delta[j-i] = 1$

La preuve se déduit directement.
Cette structure établie, nous laissons le lecteur se convaincre qu'elle réalise une bijection entre l'ensemble des arêtes et les bipoints $(origine,\ cible)$ tels que $origine \neq cible$.
Appelons $Id([0..n-1])$ l'ensemble des couples d'entiers égaux compris dans l'intervalle $[0..n-1]$, et $Domar$ le domaine des arêtes, à savoir $[0..n-1]*[0..n-1] - Id([0..n-1])$. L'assertion s'écrit:

$$\forall a,\ b \in Domar,\quad Arete[a] = Arete[b] \Leftrightarrow a = b \qquad (\ Diff_{all}\ )$$

4.1.2 Calcul des coordonnées des arêtes

Voici maintenant la seconde phase de notre projet.
Nous disposons des coordonnées des points du nuage par 2 tableaux Xnu et Ynu et nous devons remplir le tableau de celles des arêtes, Xar et Yar. Le programme correspondant est le suivant :

/* *ArCoord* : calcul des coordonnées des arêtes */

```
Pour i = 0, n-1
   Pour b = 0, 1
      Pour j = b*(i+1), n-1+(b-1)*(n-i)
               ar := Arete[i,j]
          Xar[ar] := Xnu[j]-Xnu[i]
          Yar[ar] := Xnu[j]-Xnu[i]
      Fin pour
   Fin pour
Fin pour
```

4.1.3 Parallélisation du programme

Dans le schéma de "*Un_wagon*", les objets manipulés, les triangles, étaient référencés par l'appel à un tableau indexé par l'indice de boucle ($i := List(k)/3$). Une assertion garantissait la bijection entre objet et indice. Dans le contexte présent, les objets sont les arêtes dont la valeur est lue dans le tableau $Arete$ indexé par le triple indice de boucle (i,b,j). L'analogie avec les échantillons de la sous-section 3.2.2 s'arrête là.
Ici, la "réquisition", puis la mise en commun par concaténation des résultats fournis par chaque itération n'a pas lieu. Nous sommes dans la configuration du *traitement personnalisé* signalé en 2.3.3. Mais la distinction essentielle consiste en ce que le résultat stocké (les coordonnées de l'arête) est dérivé des composantes de l'indice de boucle, tandis que le lieu de stockage est propriété de l'objet ar.
En conséquence, si la même arête était pointée lors de 2 itérations distinctes, il y aurait effectivement dépendance de données, conflit d'écriture en mémoire et pas simplement redondance. Nous sommes dans la dernière des sous-configurations listées en 2.3.3. L'adjonction de $Diff_{all}$ est indispensable pour signaler au compilateur que la parallélisation de la boucle est légale.
On peut observer à ce propos une autre différence de "*ArCoord*" avec "*Un_wagon*". La propriété $Diff_{all}$ est présente au sein du texte-source de "*ArCoord*" par le biais des formules vérifiées ci-dessus et utilisées lors de la construction du tableau $Arete$. Mais un compilateur ne saurait l'en extraire, ce qui impose qu'on la lui fournisse.

4.2 Calculs parallèles dans un Corps de Galois

4.2.1 L'inversion

Lorsque p est un naturel premier, l'anneau Z/pZ des entiers *modulo p* est un corps fini ou *corps de Galois*, c'est-à dire un groupe cyclique pour la multiplication .
Chaque élément de ce groupe est une puissance de g, l'un quelconque des éléments générateurs du groupe, et l'on a : $g^{p-1} = 1$ *modulo p*.
L'objectif du programme "*L_Inverse*" est de déterminer pour chacun des $p-1$ éléments du groupe son inverse.
Si q est l'inverse de p : $p * q = q * p = 1$ *modulo p*.
Soit e et f les exposants appelés *ordres*, relatifs respectivement à p et q, tels que $g^e = p$ et $g^f = q$.
Nous obtenons : $g^{e+f} = 1$ *modulo* $p = g^{p-1}$ *modulo p*. Cela donne : $f = p - 1 - e$.
La connaissance de l'ordre des éléments p nous offre ainsi la table des inverses.
"*L_Inverse*" calcule donc en prélude la liste des puissances de g et inversement la liste des ordres des éléments.
Le principe de ce prélude tient en la boucle qui suit, dans laquelle *Elem* est un élément.

```
 Pour e = 1, p-1
        Elem[e] := g^e mod p
  Ordre[Elem[e]] := e
Fin pour
```

Assertion

Le tableau Elem contient des éléments tous distincts ($Diff_{all}$)

Preuve : Ce sont les $p-1$ éléments du groupe.
Cela écarte l'éventualité d'une dépendance entre les instructions de 2 itérations différentes de la boucle.
Cependant, un problème apparait. Lorsque p devient grand, g^e explose. Il serait donc loisible de réécrire cette boucle.

```
Elem[e] := 1
Pour e = 1, p-1
           Elem[e] := Elem[e]*g  mod p
     Ordre[Elem[e]] := e
Fin pour
```

Cette dernière version contrarie notre problématique. Elle brise la faculté de la boucle d'être exécutée en parallèle. En effet, nous avons introduit une dépendance entre les occurrences de la première instruction de 2 itérations successives.
Suposons que la valeur de p soit inférieur à 1000.
Pour neutraliser l'obstacle, nous construisons une table de référence des $i^{èmes}$ et des $j^{èmes}$ puissances de g *modulo* p pour $j = 10 * i$ et i variant de 1 à 10.

```
U[0]  := Elem[0] := 1 ;
Pour i = 1, 10
     U[i] := (U[i-1]*g) modulo p
Fin pour
     D[1] := U[10]
Pour i = 2, 10
     D[i] := (D[i-1]*D[1]) modulo p
Fin pour
```

Dans le cas où p est un *grand nombre*, il existe des algorithmes plus efficaces de calculs de *puissances modulo* [Sch96].

```
Pour i = 1, p-1                                  /* boucle ''L_Element_et_son_Ordre'' */
          Elem[i] :=  U[i modulo 10]*D[i/10]  modulo p
    Ordre[Elem[i]] := i ;
```

Il ne nous reste qu'à dresser la table des inverses.

```
Pour i = 1, p-1                                 /* boucle ''La_Table_des_Inverses'' */
     Inverse[i] := Elem[p-1-Ordre[i]]
Fin pour
```

"*La_Table_des_Inverses*" est exécutable en parallèle sans autre forme de procès.
A contrario, le mode parallèle de la boucle "*L_Element_et_son_Ordre*" est conditionné à l'assertion $Diff_{all}$. Tout comme dans "*ArCoord*", les données ne sont pas injectées a priori en entrée, c'est le programme qui les calcule et $Diff_{all}$ peut être inférée du texte du programme. Cependant, sauf compilateur dédié à cette tâche, c'est une assertion à rattacher au source pour le compléter et valider par là-même le parallélisme.

4.2.2 Génération pseudo-aléatoire

Voici un exemple dans la veine du précédent.
On souhaite tester une famille de f générateurs d'entiers G_k de séquence périodique pseudo-aléatoire de même période T. Soit N la valeur maximale d'un élément, supposée commune à toute la famille. La famille est paramétrée par la graine $K_0 = graine(k)$ du générateur. On dispose en outre des formules $Xi = F(i, K_0)$ qui permettent de connaître pour chaque générateur et pour toutes les valeurs du rang i le retour de la séquence en fonction de la graine.
Voici par exemple le fameux générateur de bits BBS, inventé par Blum, Blum et Shub [BBS86], [SSS93].

```
p, q : grands nombres tels que :
       p = 3 mod 4 &  p, p/2, p/4 : premiers
       q = 3 mod 4 &  q, q/2, q/4 : premiers
n    :  n := p*q
X    :  entier trouvé aléatoirement, premier avec n
Xo   :  la graine  := X^2 mod n
Xi   :  Xi := (Xi-1)^2 mod n
```

Il est prouvé que : $X_i = X_0^{2^i \ mod \ (p-1)*(q-1)} \ mod \ n$
Le bit de poids le plus faible du nombre X_i génère la séquence aléatoire. La famille est ici paramétrée par X. Si ses différentes valeurs sont bien sélectionnées, la période T sera constante pour toute la famille et de l'ordre de $n/8$.

Notre projet est de dresser la liste des rangs d'un élément donné $el, el \in [0..N]$, quand la famille est balayée. Le programme à gauche ci-dessous est modifié en vue de transformer la boucle $Tant\_que$ en une boucle $Pour$.

```
Pour k = 1, f
       Ko := graine(k)
        i := 0
        Tant_que Y <> el
             i := i+1
             Y := F(i,Ko)
        Fin_tant_que
        Rangel[k] := i
Fin_pour
```

```
Pour k = 1, f
   Ko := graine(k)
    i := 0
    Pour i = 0, T-1    /*  boucle parallèle  */
         i := i+1
         Y := F(i,Ko)
         Si Y = el, alors :
             Rangel[k] := i
         Fin_si
     Fin_pour
 Fin_pour
```

La propriété de périodicité assure que, pour un générateur G_k donné, au plus un élément produit est égal à *el*. Cela constitue l'assertion qui doit accompagner la parallélisation du programme pour lever toute dépendance de données.
En pratique, la valeur de T est très grande et l'architecture effective du parallélisme dépendra des ressources dont on dispose. Par ailleurs, un mécanisme d'arrêt des calculs doit être mis en oeuvre dès que l'élément *el* est repéré, déclenchant l'incrémentation de la boucle externe. Dans le même ordre d'idées, la boucle externe elle-même est composée d'itérations indépendantes et peut être parallélisée inconditionnellement.

4.3 Assertion, bijection, périodicité

A y regarder de plus près, il existe une similitude entre les 3 exemples de ce chapitre.
Ces 3 programmes manipulent *une relation bijective*. Dans le 1^{er} cas, l'indice (vecteur) de boucle est (*origine*, *cible*) et le résultat -les coordonnées fonctions de cet indice- est rangé dans 2 tableaux indexés par *ar*, l'élément en bijection avec l'indice de boucle.
Dans les 2 exemples arithmétiques, l'élément périodique constitue l'indice de boucle et son correspondant, *l'ordre* (exemple 2) ou le *rang* (exemple 3), est utilisé comme index du tableau de rangement du résultat, égal cette fois-ci à l'indice de boucle.

Chapitre 5

Conclusion

Notre but était d'illustrer l'emploi des méthodes formelles pour exhiber les propriétés d'un programme ou en extraire de l'information en vue de son optimisation. Nous avons voulu montrer par l'exemple que cette préoccupation est susceptible de couvrir un éventail diversifié de programmes de calculs scientifiques.
Nous nous sommes limités ici aux techniques de parallélisation, et dans une moindre mesure au pipeline logiciel. Par ailleurs, les assertions que nous avons rencontrées s'appliquent aux données du programme plutôt qu'à ses structures de contrôle. Les données externes comme celles calculées par le programme appellent des assertions si nous voulons que le compilateur tire profit de propriétés les concernant. Lorsque les données sont extraites d'une base de données, cela pourrait faire partie du travail de celle-ci de fournir ces renseignements.

Selon le cas, les propriétés utiles affleurent plus ou moins à la conscience du concepteur du programme. L'intérêt des techniques de formalisation est de garantir la réalité et la rigueur de ces connaissances.
Dans certains cas, cela permet de valider une transformation de programme réalisée par le développeur. Mais nous envisageons également l'incorporation de ces données au texte-source du programme sous une forme qui reste à définir selon le contexte et les besoins. Certains langages exigent déjà cette démarche: une méthode formelle telle que *B* [Abr96] exige ainsi un *invariant* explicite pour chaque composant du programme et un couple $(invariant, variant)$ pour chaque boucle aux fins de preuve de correction.
Nous aimerions élargir cet horizon dans la perspective de l'optimisation (au sens a priori le plus ouvert du terme) des programmes : "*dans quelle mesure l'explicitation de propriétés -(sous la forme d'assertions intégrées au texte du programme)- peut-elle aider le travail de la compilation et de son optimisation?*" nous semble être une question digne d'approfondissement.
Cela doit conduire à élaborer une typologie des propriétés en fonction de leur utilité en vis-à-vis des différentes techniques d'optimisation automatique à la compilation relevées dans la littérature.
En corollaire, un langage d'instructions dédié à l'écriture de ces assertions s'imposerait.

Bibliographie

[Abr96] Jean-Raymond Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

[BB83] Pierre Berlioux and Philippe Bizard. *Algorithmique: construction, preuve et évaluation des programmes.* Dunod informatique. Dunod, 1983.

[BB88] Pierre Berlioux and Philippe Bizard. *Algorithmique 2: structures de données et algorithmes.* Bordas, 1988.

[BBS86] L Blum, M Blum, and M Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15, n.2, 1986.

[Car90] Christian Carrez. *Des structures aux bases de données.* Dunod informatique. Bordas, 1990.

[Fea91] Paul Feautrier. Data flow analysis of array and scalar references. *International Journal of Parallel Processing*, 20(1):23–53, 1991.

[FG99] Pascal Jean Frey and Paul-Louis George. *Maillages: applications aux éléments finis.* Hermès, 1999.

[Ghi98] Rakesh Ghiya. *Putting Pointeur Analysis to Work.* PhD thesis, School of Computer Science, MacGill University, January 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Mon96] Jean-Francois Monin. *Comprendre les méthodes formelles: panorama et outils logiques.* Collection technique et scientifique des télécommunications. Masson, 1996.

[Sch96] Bruce Schneier. *Applied cryptography: protocols, algorithms and source code in C.* J. Wiley and sons, 1996.

[SSS93] S J Sherperd, P W Sanders, and C T S. The quadratic residue cipher and some notes on implementation. *Cryptologia*, XVII, (3):264–282, July 1993.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399