Manipulation explicite de pointeurs et efficacité en Scratchpad

Jean MOULIN OLLAGNIER [1]

Conférence à l'occasion de la journée Scratchpad du 4 décembre 1990

[1]Laboratoire d'Analyse et applications, Université Paris–Nord & LIX, Ecole Polytechnique

Le point de départ de l'histoire que je vais vous raconter se situe à Paris–Nord où plusieurs collègues s'initient à l'usage de Scratchpad sur un 6150.

Laissant agir une légitime curiosité, Aviva Szpirglas voulut tester le savoir-faire de notre magnifique système sur les permutations ; le dialogue fut en substance le suivant.

```
->list1 : L I := [1..10]

   (1)  [1,2,3,4,5,6,7,8,9,10]
                                                        Type: List Integer

->list2 : L I := [11..12]

   (2)  [11,12]
                                                        Type: List Integer

->perm1 : PERM I := list1

   (3)  PERM( (1,2,3,4,5,6,7,8,9,10))
                                                Type: Permutations Integer

->perm2 : PERM I := list2

   (4)  PERM( (11,12))
                                                Type: Permutations Integer

->sign (perm1 * perm2)

   (5)  - 1
                                                             Type: Integer

->sign perm1 * sign perm2

   (6)  1
                                                     Type: PositiveInteger
```

Ainsi, la signature implantée dans le domaine `Permutations` n'était-elle pas un homomorphisme.

L'erreur apparaissait de façon évidente (un très bon point pour Scratchpad : la transparence) dans le code algébrique et j'envoyai un message pour le signaler à l'un des auteurs, Johannes Grabmeier.

Comme on pouvait s'y attendre, l'erreur a été corrigée dans les versions ultérieures du module.

Interessé par la réalisation de ce domaine des permutations à support fini sur un ensemble, j'avais joint à mon courrier un certain nombre de remarques proposant entre autres un mode de représentation différent des objets du domaine et des algorithmes plus efficaces pour les opérations élémentaires sur ces permutations.

Dans sa réponse, Grabmeier défendait à juste titre son point de vue et m'encourageait à développer le mien (souplesse de Scratchpad).

Je vais donc décrire ce que j'ai réalisé sur ce sujet en illustrant mon propos par des morceaux choisis des sources des divers domaines que j'ai écrits ou modifiés.

Des extraits de la trace d'une session vous permettront ensuite de comparer l'efficacité des deux réalisations.

1 Le domaine Qermutations

Il s'agit de représenter les permutations à support fini sur un ensemble et de réaliser les opérations de groupe, la signature, la décomposition en cycles, etc...

Mon parti-pris a été de considérer que l'ensemble paramétrant le domaine était muni d'un ordre total. Ceci permet de définir une représentation canonique des objets du domaine de sorte que l'égalité soit celle des représentations. Même sans faire appel à l'axiome du choix (ce ne serait pas de meilleur goût à propos de mathématiques constructives), on peut se rendre compte assez clairement que tous les domaines imaginables sont d'une façon ou d'une autre totalement ordonnés ; bien sûr, l'ordre correspondant peut n'avoir aucun "sens".

Ainsi une permutation p à support fini sur un ensemble ordonné S est-elle représentée par la liste finie des éléments non-diagonaux de son graphe ; chaque point du graphe est naturellement codé par le couple (`record`) de ses coordonnées et la liste est rangée par ordre croissant des premières coordonnées.

Le calcul de l'inverse consiste alors à trier relativement à la première coordonnée la liste des symétriques des points du graphe par rapport à la diagonale ; la complexité de cette opération est raisonnable si la fonction de tri de Scratchpad est correcte, ce qui semble être le cas expérimentalement.

Le calcul du produit $p*q$ consiste à réordonner q par rapport à la seconde coordonnée, puis à composer la liste représentant la première permutation q, classée en fonction des images avec la seconde permutation p classée en fonction des preimages ; cette composition se fait de manière sequentielle et ressemble à une fusion. Il n'y a plus qu'à trier pour obtenir la représentation canonique de la permutation composée $p * q$. Le travail de fusion est effectué par la fonction `delta`.

Cette méthode a une compléxité en $n \log n$ alors que le produit dans le module `Permutations` est calculé en temps quadratique.

Quelques lignes du code source du domaine `Qermutations`.

```
)abbrev domain QERM Qermutations

Qermutations (S:OrderedSet) : public == private where

  E             ==>     Expression
  I             ==>     Integer
  B             ==>     Boolean
  L             ==>     List
  OUTFORM       ==>     OutputForm  -- for a later work !
  DLIST         ==>     DoublyLinkedList
  WDLIST        ==>     WellDoublyLinkedList
  POINT         ==>     Record (xx:S,yy:S)
  PATH          ==>     Record (he:WDLIST S,ta:WDLIST S,status:I)
  OandC         ==>     Record (op:L PATH,cl:L PATH)
  TRI           ==>     MergeSortPackage (POINT)
```

```
public ==> Group with

  coerce              :       $ -> E
  coerce              :       L S -> $
  coerce              :       L L S -> $
  imageOf             :       ($,S) -> S
  cyclesOf            :       $ -> L L S
  theCyclesOf         :       $ -> L L S
  order               :       $ -> I
  sign1               :       $ -> I
  sign2               :       $ -> I


private ==> add

  Rep := L POINT

  delta (r,q) ==
    r = [] => q
    q = [] => r
    (first r).yy < (first q).xx => cons (first r,delta(rest r,q))
    (first q).xx < (first r).yy => cons (first q,delta(r,rest q))
    m := construct$POINT ((first r).xx,(first q).yy)
    m.xx = m.yy => delta (rest r, rest q)
    cons (m,delta (rest r, rest q))

  1 == nil$Rep

  inv p ==sort([construct(m.yy,m.xx)$POINT for m in p],#1.xx <$S #2.xx)

  q * p == sort (delta (sort(p,#1.yy <$S #2.yy),q),#1.xx <$S #2.xx)
```

2 La décomposition en cycles

La réalisation de Grabmeier et al. utilise la décomposition en cycles pour le calcul de la signature ainsi que pour l'affichage des permutations à support fini.

Cette décomposition présente un intérêt intrinsèque et j'ai donc cherché à la réaliser.

Appelons chemin une liste finie d'éléments de S dans laquelle le suivant d'un élément est son image par la permutation p que l'on étudie.

La donnée de p correspond alors à une liste de chemins de longueur 2 ; les éléments qui apparaissent sont ceux du support de p et ils apparaissent exactement deux fois, une fois en début de chemin, une fois en fin.

La décomposition en cycles est en fait une recomposition consistant à recoller des chemins jusqu'à n'avoir plus que des chemins circulaires.

Pour réaliser ce travail de recollement économiquement, il faut maintenir des pointeurs sur le premier et le dernier élément d'une liste et pouvoir passer d'un élément au précédent.

La structure de liste doublement chainée `DLIST` s'impose donc et les chemins correspondent à la macro-définition suivante

```
PATH ==> Record (he:DLIST S, ta : DLIST S, status : I).
```

où les deux premiers champs de l'enregistrement sont des pointeurs sur la tête et sur la queue de la liste, le troisième champ étant là pour les besoins de l'algorithme : `status` représente les "points de vie" de l'enregistrement et prend les valeurs 0, 1 et 3.

Tant qu'il reste des chemins non-circulaires, le travail n'est pas terminé. On considère alors les deux listes de même longueur de ces chemins, l'une ordonnée relativement à l'élément de S qui est au début du chemin, l'autre ordonnée relativement à l'élément de S qui est à la fin du chemin.

Le parcours simultané de ces deux listes rencontre tous les couples de chemins que l'on peut recoller ; il ne faut pas bien entendu utiliser le même chemin plusieurs fois et c'est à cela que sert le champ `status` des chemins.

Après un tel parcours, le nombre de chemins non-circulaires est au plus les deux tiers de sa valeur précédente : le nombre d'étapes est alors logarithmique et la compléxité de toute l'opération en $n \log n$.

C'est la fonction `glueByPairs` qui réalise le travail de recollement correspondant à un parcours.

```
glueByPairs l ==
  r : L PATH := sort (l,former2Than)  -- l is increasing wrt .he.0
  newList : L PATH := nil$(L PATH)
  while ^null r repeat
    r0 : PATH := first r
    l0 : PATH := first l
    n : I := (l0.status) + (r0.status)
    -- no Pascal-like case instruction in Scratchpad !
    if n <= 2 -- both records are living : join lists and kill records
    then
      newList := cons (together(r.0,l.0),newList)
      (r.0).status := 3
      (l.0).status := 3
    else
      if n = 3 -- one only is living and it is a newcomer : mark it
      then
        if (r.0).status = 0
        then (r.0).status := 1
        else (l.0).status := 1
      else
        if n = 4 -- it has been seen : save the list and kill the record
        then
          if (r.0).status = 1
          then
            newList := cons ([(r.0).he,(r.0).ta,0]::PATH,newList)
            (r.0).status := 3
          else
            newList := cons ([(l.0).he,(l.0).ta,0]::PATH,newList)
            (l.0).status := 3
        -- else if n = 6, both are dead : nothing more to do
    l := rest l
    r := rest r
  newList
```

Je ne rentre pas dans le détail des fonctions qui conduisent à une forme canonique de la décomposition en cycles. Ajoutons simplement que diverses fonctions du module `DLIST` n'existent que pour leurs effets de bord ; pour eviter des opérations inutiles j'ai modifié ce module en un nouveau, `WDLIST`.

Voici ces nouveautés :

```
)abbrev domain WDLIST WellDoublyLinkedList

WellDoublyLinkedList(S: Set): public == private where

   NNI ==> NonNegativeInteger
   I   ==> Integer
   B   ==> Boolean

   public == DoublyLinkedListCategory(S) with

     empty           :     () -> $      -- the empty list
     splitBefore     :     $ -> $
     splitAfter      :     $ -> $
     joined?         :     ($,$) -> B   -- for side effects, no control !


   private == add

      Rep := Record(nx:$,pv:$,value:S)

      foo:$ := (NIL$Lisp):$

      empty () == foo

      splitBefore (x) ==
        null x => error "cant split..."
        null (x.pv) => empty
        prev:$ := x.pv ; prev.nx := foo ; x.pv := foo
        prev

      splitAfter (x) ==
        null x => error "cant split..."
        null (x.nx) => empty
        next:$ := x.nx ; next.pv := foo ; x.nx := foo
        next

      joined? (x,y) ==
        null x => false
        null y => false
        x.nx := y ; y.pv := x
        true
```

3 La signature

On peut calculer la signature d'une permutation assez rapidement sans utiliser la décomposition en cycles : il suffit de calculer la signature de la permutation mise en œuvre quand on trie par rapport à la seconde coordonnée la liste des points du graphe, initialement en ordre

croissant par rapport à la première coordonnée.

Comme le fait remarquer D. E. Knuth, il est possible de calculer petit à petit la parité du nombre d'inversions dans un tri-fusion sans compromettre l'efficacité de la méthode.

C'est ce que je réalise dans le domaine sans objet `TRI` dont voici quelques extraits.

```
)abbrev package TRI MergeSortPackage

MergeSortPackage (S:Set) : public == private where

  GT  ==>     Mapping(Boolean,S,S)
  E   ==>     Expression
  L   ==>     List
  B   ==>     Boolean
  I   ==>     Integer
  Run ==>     Record (run:L S,sgn:I,par:I)
  RLS ==>     Record (l: L S, s : I)     -- Tuple is not a valid type !!

  public ==> with

    mergeSort           :       (L S,GT) -> RLS

  private ==> add

    mmerge              :       (L Run,GT) -> L Run
    merge               :       (Run,Run,GT) -> Run
    mergeWithSign       :       (L S,L S,I,GT) -> RLS
    listOfRuns          :       (L S,GT) -> L Run

    mergeSort (ls,gt) ==
      null ls => construct$RLS (ls,1)
      lr : L Run := listOfRuns (ls,gt)
      while ^null (rest lr) repeat lr := mmerge (lr,gt)
      construct$RLS ((first lr).run,(first lr).sgn)
```

4 Expériences

Les résultats expérimentaux sont en accord qualitatif avec l'analyse : les gains de temps sont notables aussi bien dans la lecture d'un objet à partir d'une décomposition en cycles que dans la multiplication des permutations. La décomposition en cycles est également plus efficace.

On peut juger de tout cela dans la trace de session qui suit. La fonction interprétée `decoupe` sert à fabriquer de grandes listes de listes d'entiers qui soient des décompositions en cycles de permutations ; ces listes de listes sont le moyen de créer les objets du domaine.

Dans le domaine `Qermutations`, `sign1` désigne la signature calculée à l'aide du tri-fusion et `sign2` celle calculée à partir de la décomposition en cycles.

L'étonnant appel

```
toto := p :: E ; .
```

force le module `Permutations` à décomposer en cycles la permutation p pour pouvoir l'afficher.

La durée de l'évaluation correspondante est à comparer avec celle nécessaire au calcul de `theCyclesOf$dom2 q` où q est la permutation p comme objet du domaine `Qermutations`.

Voici donc, sinon des preuves, du moins une honnête publicité comparative.

```
;)load qerm
;dom2 := QERM I

   (1)  Qermutations Integer
                                                              Type: Domain
                                      Time: .067 (IN) + .3 (OT) = .367 sec

;)load perm
;dom1 := PERM I

   (2)  Permutations Integer
                                                              Type: Domain
                                    Time: .067 (IN) + .067 (OT) = .133 sec

;)load WDLIST I
;)load TRI
;decouper (o:I,n:I,k:I) : L L I ==
;  o > n => [] :: (L L I)
;  (o = n) or ((o div k).remainder = 0) =>
;    fl : L I := list o
;    cons (fl,decouper(o+1,n,k))
;  l : L L I := decouper (o+1,n,k)
;  fl : L I := first l
;  fl := cons (o,fl)
;  cons (fl,rest l)
Function declaration decouper : (Integer,Integer,Integer) -> List List
Integer has been added to workspace.
                                                                Type: Void
                                    Time: 1.2 (IN) + .367 (OT) = 1.567 sec

;l1 : L L I := decouper (1,103,11)  ;
                                                                Type: Void
                                 Time: 5.833 (IN) + 39.367 (OT) = 45.2 sec

;l2 : L L I := decouper (1,101,13)  ;
                                                                Type: Void
                                                      Time: .1 (IN) = .1 sec

;l3 : L L I := decouper (1,503,22)  ;
                                                                Type: Void
                                Time: .1 (IN) + .1 (EV) + .1 (OT) = .3 sec

;l4 : L L I := decouper (1,509,19)  ;
                                                                Type: Void
                                           Time: .1 (IN) + .1 (EV) = .2 sec

;l5 : L L I := decouper (1,1009,103)  ;
                                                                Type: Void
```

```
                                    Time: .1 (IN) + .233 (EV) = .333 sec

;l6 : L L I := decouper (1,1013,101)  ;
                                                                 Type: Void
                                   Time: .133 (IN) + .267 (EV) = .4 sec

;l7 : L L I := decouper (1,2000,199)  ;
                                                                 Type: Void
                                      Time: .1 (IN) + .4 (EV) = .5 sec

;l8 : L L I := decouper (1,2000,197)  ;
                                                                 Type: Void
                                                  Time: .6 (EV) = .6 sec

;p1 : dom1 := l1 ;
                                                                 Type: Void
                              Time: 12.35 (IN) + 8.583 (OT) = 20.933 sec

;p2 : dom1 := l2 ;
                                                                 Type: Void
                                               Time: .433 (IN) = .433 sec

;p3 : dom1 := l3 ;
                                                                 Type: Void
                                            Time: 6.267 (IN) = 6.267 sec

;p4 : dom1 := l4 ;
                                                                 Type: Void
                                              Time: 6.7 (IN) = 6.7 sec

;p5 : dom1 := l5 ;
                                                                 Type: Void
                                          Time: 21.933 (IN) = 21.933 sec

;p6 : dom1 := l6 ;
                                                                 Type: Void
                                          Time: 23.867 (IN) = 23.867 sec

;q1 : dom2 := l1 ;
                                                                 Type: Void
                                   Time: .4 (IN) + .517 (OT) = .917 sec

;q2 : dom2 := l2 ;
                                                                 Type: Void
                                               Time: .167 (IN) = .167 sec

;q3 : dom2 := l3 ;
                                                                 Type: Void
                                               Time: .767 (IN) = .767 sec
```

```
;q4 : dom2 := l4 ;
                                                                  Type: Void
                                                      Time: .7 (IN) = .7 sec

;q5 : dom2 := l5 ;
                                                                  Type: Void
                                             Time: 1.0 (IN) + .1 (OT) = 1.1 sec

;q6 : dom2 := l6 ;
                                                                  Type: Void
                                                    Time: 1.4 (IN) = 1.4 sec

;p := p1 * p2 ;
                                                                  Type: Void
                          Time: .467 (IN) + .8 (EV) + 1.667 (OT) = 2.933 sec

;q := q1 * q2 ;
                                                                  Type: Void
                            Time: .2 (IN) + .067 (EV) + 1.2 (OT) = 1.467 sec

;lq0 : L L I := cyclesOf$dom2 q ;
                                                                  Type: Void
                               Time: .3 (IN) + 1.1 (EV) + .2 (OT) = 1.6 sec

;lq : L L I := theCyclesOf$dom2 q ;
                                                                  Type: Void
                                 Time: .1 (IN) + .4 (EV) + .1 (OT) = .6 sec


;toto := p :: E ;
                                                                  Type: Void
                                     Time: .033 (IN) + 4.867 (OT) = 4.9 sec

;pp : dom1 := lq ;
                                                                  Type: Void
                                                      Time: .6 (IN) = .6 sec

;(p =$dom1 pp)

   (30)  true
                                                               Type: Boolean
                            Time: .133 (IN) + .433 (EV) + .133 (OT) = .7 sec

;p := p3 * p4 ;
                                                                  Type: Void
                                      Time: .2 (IN) + 20.1 (EV) = 20.3 sec

;q := q3 * q4 ;
                                                                  Type: Void
                                    Time: .233 (IN) + .767 (EV) = 1.0 sec
```

```
;lq0 : L L I := cyclesOf$dom2 q ;
                                                                Type: Void
                                                  Time: 2.9 (EV) = 2.9 sec

;lq : L L I := theCyclesOf$dom2 q ;
                                                                Type: Void
                                          Time: .1 (IN) + 2.9 (EV) = 3.0 sec

;toto := p :: E ;
                                                                Type: Void
                                     Time: .067 (IN) + 9.367 (OT) = 9.433 sec

;pp : dom1 := lq ;
                                                                Type: Void
                                               Time: 6.067 (IN) = 6.067 sec

;(p =$dom1 pp)

   (37)  true
                                                             Type: Boolean
                                          Time: .1 (IN) + 7.0 (EV) = 7.1 sec

;p := p5 * p6 ;
                                                                Type: Void
                          Time: .167 (IN) + 11.0 (EV) + .1 (OT) = 11.267 sec

;q := q5 * q6 ;
                                                                Type: Void
                                           Time: .2 (IN) + .8 (EV) = 1.0 sec

;lq0 : L L I := cyclesOf$dom2 q ;
                                                                Type: Void
                                          Time: .1 (IN) + 5.1 (EV) = 5.2 sec

;lq : L L I := theCyclesOf$dom2 q ;
                                                                Type: Void
                                       Time: .2 (IN) + 5.133 (EV) = 5.333 sec

;toto := p :: E ;
                                                                Type: Void
                                              Time: 26.433 (OT) = 26.433 sec

;pp : dom1 := lq ;
                                                                Type: Void
                                     Time: 21.1 (IN) + .133 (OT) = 21.233 sec
```

```
;(p =$dom1 pp)

   (44)  true
                                                          Type: Boolean
                                        Time: .1 (IN) + 27.1 (EV) = 27.2 sec

;eval$dom1 (p,1000)

   (45)  1002
                                                          Type: Integer
                                      Time: .5 (IN) + 2.767 (OT) = 3.267 sec

;imageOf$dom2 (q,1000)

   (46)  1002
                                                          Type: Integer
                                      Time: .2 (IN) + .167 (OT) = .367 sec

;sign1$dom2 q

   (47)  - 1
                                                          Type: Integer
                                      Time: .4 (EV) + .133 (OT) = .533 sec

;sign2$dom2 q

   (48)  - 1
                                                          Type: Integer
                            Time: .067 (IN) + 5.1 (EV) + .133 (OT) = 5.3 sec

;p7 : dom1 := l7 ;
                                                            Type: Void
                                           Time: 80.433 (IN) = 80.433 sec

;p8 : dom1 := l8 ;
                                                            Type: Void
                                              Time: 81.0 (IN) = 81.0 sec

;p := p7 * p8 ;
                                                            Type: Void
                                    Time: .233 (IN) + 15.5 (EV) = 15.733 sec


;toto := p :: E ;
                                                            Type: Void
                                             Time: 107.0 (OT) = 107.0 sec

;q7 : dom2 := l7 ;
                                                            Type: Void
                                           Time: 2.067 (IN) = 2.067 sec
```

```
;q8 : dom2 := l8 ;
                                                                  Type: Void
                                                  Time: 2.0 (IN) = 2.0 sec

;q := q7 * q8 ;
                                                                  Type: Void
                                       Time: .267 (IN) + 1.733 (EV) = 2.0 sec

;lq : L L I := theCyclesOf$dom2 q ;
                                                                  Type: Void
                          Time: .167 (IN) + 10.5 (EV) + .1 (OT) = 10.767 sec

;pp : dom1 := lq ;
                                                                  Type: Void
                                             Time: 85.217 (IN) = 85.217 sec

;(p =$dom1 pp)

   (58)  true
                                                                  Type: Boolean
                            Time: .2 (IN) + 108.4 (EV) + .1 (OT) = 108.7 sec

;sign1$dom2 q

   (59)  1
                                                                  Type: Integer
                              Time: .1 (IN) + .4 (EV) + .067 (OT) = .567 sec

;sign2$dom2 q

   (60)  1
                                                                  Type: Integer
                     Time: .067 (IN) + 10.933 (EV) + .067 (OT) = 11.067 sec
```