Devoir surveillé 3 c

X 2005

Option informatique MP1, MP2 & MP3

I Génération de circuits PLA

I.1 Fonctions booléennes

- Dans ce problème, les booléens sont représentés par 0 pour la valeur faux, et 1 pour la valeur vrai. $B = \{0, 1\}$ est l'ensemble des booléens.
- Si x est un booléen, on notera $\overline{x} = 1 - x$ la négation de x.
- On écrira $x \wedge y$ pour la conjonction (le produit) de x et y.
- On écrit $x \vee y$ pour la disjonction (la somme) de x et y.
- On écrit $x \oplus y$ pour le ou exclusif, défini par $0 = 0 \oplus 0 = 1 \oplus 1$ et $1 = 0 \oplus 1 = 1 \oplus 0$.
- Soit f une fonction booléenne de n arguments booléens c'est-à-dire une fonction de B^n dans B ($n > 0$). Elle peut être définie par sa table de vérité.
  Exemple : les fonctions g et h sont définies par $g(x_0, x_1) = \overline{x_0} \vee x_0 \wedge x_1$ et $h(x_0, x_1, x_2) = x_0 \oplus x_2$. Leurs tables de vérités sont

x_0	x_1	$g(x_0, x_1)$
0	0	1
0	1	0
1	0	1
1	1	1

x_0	x_1	x_2	$h(x_0, x_1, x_2)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

- Chaque ligne $(x_0, x_1, \ldots, x_{n-1})$ de la table de vérité d'une fonction de B^n dans B peut être représentée par le produit des **littéraux** (x_i ou $\overline{x_i}$) associés aux variables de la fonction. Leur somme est la **forme normale disjonctive canonique** associée à la fonction.
  Ainsi la forme canonique de g est $\overline{x_0} \wedge \overline{x_1} \vee x_0 \wedge \overline{x_1} \vee x_0 \wedge x_1$.
- Une forme normale disjonctive (f.n.d) est une somme de produits de littéraux.

Question 1
Exprimer la forme normale disjonctive canonique de h.

Pour réduire le nombre de produits dans la f.n.d. d'une fonction, on peut utiliser l'identité $x \wedge p \vee \overline{x} \wedge p = (x \vee \overline{x}) \wedge p = 1 \wedge p = p$.
Ainsi $g(x_0, x_1) = \overline{x_0} \wedge \overline{x_1} \vee x_0 \wedge \overline{x_1} \vee x_0 \wedge x_1 = (\overline{x_0} \vee x_0) \wedge \overline{x_1} \vee x_0 \wedge x_1 = \overline{x_1} \vee x_0 \wedge x_1$.

Question 2
Réduire le nombre de produits dans la f.n.d. de h.

I.2 Utilisation des entiers

Un sous ensemble de $\{0, 1, \ldots, n-1\}$ peut être représenté par sa fonction caractéristique ; c'est la suite de booléens $x_0, x_1, \ldots, x_{n-1}$ avec $x_i = 1$ si et seulement si i appartient à l'ensemble.
Cette suite est elle-même représentable par l'entier x dont la représentation binaire sur n bits est $(x_0, x_1, \ldots, x_{n-1}) : x = \sum_{i=0}^{n-1} x_i 2^i$. Les bits de x sont les booléens $x_0, x_1, \ldots, x_{n-1}$,
x_0 est le bit le moins significatif, x_{n-1} est le bit le plus significatif.

Caml possède les opérations logiques `land`, `lor`, `lxor` et `lnot` sur les entiers positifs telles que

- `x land y` est représenté par $(x_0 \wedge y_0)2^0 + (x_1 \wedge y_1)2^1 + \ldots + (x^{n-1} \wedge y^{n-1})2^{n-1}$
- `x lor y` est représenté par $(x_0 \vee y_0)2^0 + (x_1 \vee y_1)2^1 + \ldots + (x_{n-1} \vee y_{n-1})2^{n-1}$
- `x lxor y` est représenté par $(x_0 \oplus y_0)2^0 + (x_1 \oplus y_1)2^1 + \ldots + (x^{n-1} \oplus y^{n-1})2^{n-1}$
- `lnot x` est représenté par $\overline{x_0}2^0 + \overline{x_1}2^1 + \ldots + \overline{x}_{n-1}2^{n-1}$

si x est représenté par $x_0 2^0 + x_1 2^1 + \ldots + x_{n-1}2^{n-1}$ et y par $y_0 2^0 + y_1 2^1 + \ldots + y_{n-1}2^{n-1}$.

Ces opérations s'exécutent en temps constant $O(1)$.

Question 3
Montrer que , pour $x > 0$, x est de la forme $x = 2^p$ $(p \geq 0)$ si et seulement si
`x land (x - 1) = 0` (on exclut 0 car `0` land p vaut toujours 0).

La distance de Hamming $d(x, y)$ entre deux entiers x et y est le nombre de bits dont ils diffèrent dans leur décomposition binaire.

Question 4
Écrire une fonction `aDistUn : int -> int -> bool` telle que `aDistUn x y` renvoie, en temps constant, la valeur vrai si et seulement si la distance de Hamming entre x et y est 1.

Un produit de littéraux $l_{i_1} \wedge l_{i_2} \wedge \ldots \wedge l_{i_p}$ $(0 \leq i_1 < i_2 \ldots < i_p < n-1)$
définit 3 sous-ensembles de $\{0, 1, \ldots, n-1\}$:

1. le masque $M = \{i_1, i_2, \cdots, i_p\}$, associé à l'entier m
2. P l'ensemble des indices de $i \in M$ tel que $l_i = x_i$, on dit que l_i est littéral positif, l'ensemble est associé à l'entier p
3. N l'ensemble des i dans Mtels que x_i est un littéral négatif $(l_i = \overline{x_i})$, associé à n.

On a bien sur $M = P \cup N$.

Question 5
Que valent `m land p`, `m lor p` et `m lxor p` ?

Ainsi le produit $x_0 \wedge x_2 \wedge \overline{x}_3 \wedge x_4$ est représenté par les deux entiers $p = 21$ et $m = 29$ dont les représentations binaires respectives sont $(1, 0, 1, 0, 1, 0, \ldots, 0)$ et $(1, 0, 1, 1, 1, 0, \ldots, 0)$.
On traduira un produit par un enregistrement de deux entiers :

```
type produit = {p: int; m: int};;
```

À chaque forme (normale disjonctive) définissant une fonction booléenne $f(x_0, x_1, \ldots, x_{n-1})$ on associera la suite de ses produits sous la forme d'une liste.

```
type fnd = produit list;;
```

Par exemple $f(x_0, x_1, x_2) = \overline{x_0} \wedge \overline{x_2} \vee x_1 \wedge \overline{x_2} \vee x_0 \wedge x_1$ est représenté par

```
[{m = 5; p = 0}; {m = 6; p = 2}; {p = 3; m = 3}]
```

Question 6
Écrire une fonction `unique : fnd -> fnd` qui prend en argument une forme f et qui retourne une représentation de la même forme où chacun des produits n'apparaît qu'une seule fois. Quelle est la complexité en temps en fonction la longueur ℓ de la forme f ?

I.3 Élimination de variables

Une variable x_i est dite **éliminable** entre deux produits p et q s'il existe des produits p' et p'' tels que $p = p' \wedge x_i \wedge p''$ et $q = p' \wedge \overline{x_i} \wedge p''$ ou $p = p' \wedge \overline{x_i} \wedge p''$ et $q = p' \wedge x_i \wedge p''$

Question 7
Écrire une fonction `varEliminable : produit -> produit -> bool` qui retourne la valeur `true` si et seulement si il existe une variable éliminable entre p et q.
Quelle est la complexité en temps de cette fonction ?

L'**unification** de deux produits qui admettent une variable éliminable est le produit $p' \wedge p''$ avec les notations ci-dessus. On notera que l'unification est vraie dès que $p \vee q$ est vraie

Question 8
Écrire une fonction `unifier: produit -> produit -> produit` qui retourne l'unification des deux produits passés en paramètres ; on supposera, sans avoir besoin de le vérifier, que les produits admettent une variable éliminable.

Question 9
Écrire une fonction `unifications : fnd -> fnd` qui prend en argument une forme f et qui retourne la liste de toutes les unifications possibles entre deux produits de f.
Quelle est la complexité en temps par rapport à la longueur ℓ de la forme f ?

On admet que ce processus s'arrête après un nombre fini d'itérations.
Si on part d'une forme f_0, on applique alors la fonction `unifications` à f_0, on obtient la forme f_1, on recommence avec `unifications f1` qui donne f_2 etc ; iIl existe p tel que f_p est vide.

Question 10
Donner la suite des h_i obtenus à partir de
$h_0 = \overline{x_1} \wedge \overline{x_2} \wedge x_3 \vee \overline{x_1} \wedge x_2 \wedge x_3 \vee x_1 \wedge \overline{x_2} \wedge \overline{x_3} \vee x_1 \wedge x_2 \wedge \overline{x_3}$

Question 11
Écrire une fonction `developper : fnd -> fnd list` qui prend en argument une forme f et qui retourne la liste $[f_k, \ldots, f_2, f_1, f_0]$ des formes obtenues en itérant la fonction `unifications` à partir de f_0.
Quelle est la complexité de cette fonction en fonction des longueurs ℓ_i des listes f_i ?

I.4 Réduction

Le but de cette partie est de trouver efficacement parmi toutes les réécritures d'une forme l'une de celles ayant un nombre de produits presque minimal.
On appellera FORME RÉDUITE le résultat de notre algorithme.

Pour générer une forme réduite de la fonction booléenne f de n arguments, on part d'une forme, par exemple la forme canonique définie à la section **I.1** qu'on note f_0.
On applique alors la fonction `développer` à f_0 ; le résultat de la fonction `developper` est une liste de listes, ayant beaucoup plus de termes au total que la forme de départ, et dont on va extraire les produits formant la forme réduite.

Pour cela on utilise la notion de couverture : le produit p **couvre** le produit q si et seulement les variables positives (resp. négatives) de p sont aussi des variables positives (resp. négatives) de q.

Question 12
Que vaut $p \vee q$ si p couvre q ?
Que peut-on éliminer dans une forme contenant p et q ?

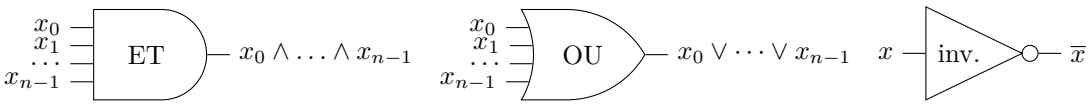
Question 13
Écrire une fonction **couvre : produit -> produit -> bool** de complexité constante qui prend en argument deux produits p et q et qui retourne la valeur `true` si p couvre par q.

Question 14
Écrire la fonction **reduire : fnd -> fnd** qui prend en argument la forme canonique `f` d'une fonction f et qui retourne une forme définissant la même fonction où il n'y a plus aucun produit couvrant un autre.
Quelle est la complexité de cette fonction par rapport aux longueurs $\ell_0, \ell_1, \dots \ell_k$ des listes a_0, a_1, …, a_k obtenues par la fonction `developper` ?

Question 15
Donner une borne supérieure de la complexité du calcul du résultat de cette forme réduite en fonction du nombre n d'arguments de la fonction f.

I.5 Circuits PLA

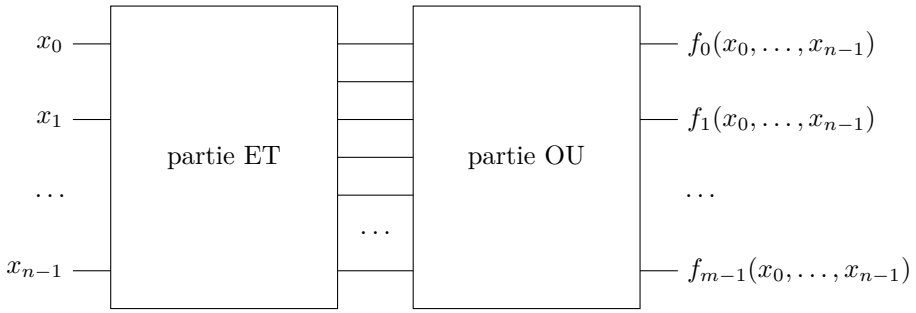
La partie combinatoire d'un circuit contient une combinaison de PORTES ET, PORTES OU et inverseurs représentés ci-dessous.



Les signaux circulant dans les fils des circuits sont assimilés aux booléens 0 et 1.
Une porte OU à n entrées $x_0, x_1, \dots x_{n-1}$ calcule la f.n.d. (disjonction) $x_0 + x_1 + \dots + x_{n-1}$.
De même une porte ET à n entrées $x_0, x_1, \dots x_{n-1}$ calcule le produit $x_0 \wedge x_1 \dots x_{n-1}$, et l'inverseur à une entrée x calcule $\overline{x}$.
Cette partie combinatoire des circuits intégrés consiste souvent à calculer une [1] fonction f de B^n dans B^m à l'aide d'un PLA (PROGRAMMABLE LOGIC ARRAY).
Un PLA, à n entrées et m sorties calcule sur chaque sortie la fonction booléenne $f_i(x_0, x_1, \dots x_{n-1})$ $(0 \leq i < m)$ en s'appuyant sur sa représentation en f.n.d.

1. c'est-à-dire m fonctions logiques.

Un PLA comporte deux parties : la partie ET ne contenant que des portes ET et des inverseurs, et la partie OU ne contenant que des [2] portes OU.



Question 16
Construire un PLA associé aux fonction g et h.

Question 17
À quoi correspond le nombre de liaisons entre les deux parties ? Peut-on le réduire ?

II Génération de circuits combinatoires

Dans cette partie, nous allons générer d'autres exemples de circuits combinatoires calculant des fonctions booléennes.
Ces circuits ne seront composés que d'inverseurs, de portes ET à 2 entrées, de portes OU à 2 entrées et de bits valant 0 ou 1.
Ils seront représentés par des arbres donnant la valeur de la sortie en fonction des valeurs des entrées, c'est-à-dire des bits figurant à leurs feuilles. Le type de ces arbres est le suivant :

```
type circuit =
    Bit of int
  | Et of circuit * circuit
  | Ou of circuit * circuit
  | Non of circuit;;
```

On ne se souciera pas du partage possible entre sous-arbres calculant la même valeur. Mais on remarquera que le temps mis par un circuit pour calculer son résultat est proportionnel à la hauteur de cet arbre.
On notera **temps de calcul** cette hauteur. Par exemple, on peut calculer en 3 unités de temps le ou-exclusif de x et y comme suit :

```
let oux x y =
  Ou (Et (x, Non y),
      Et (Non x, y));;
```

2. une porte pour chaque fonction.

Question 18
Écrire les fonctions `bitAdd : circuit -> circuit -> circuit -> circuit` et `bitRetenue : circuit -> circuit -> circuit -> circuit` qui prennent en argument trois circuits x, y et r fournissant les valeurs x', y' et r'.
La première retourne le circuit calculant le reste modulo 2 de $x' + y' + r'$.
La seconde retourne le circuit calculant le quotient de la division par 2 de $x' + y' + r'$.

La représentation binaire de l'entier x sur n bits ($n > 0$) est à présent décrite par un tableau de n circuits, dont la i-ème entrée vaut le bit x_i, comme indiqué dans la partie I.
On appellera *mot machine* de longueur n ce tableau.

```
type mot = circuit array;
```

Un additionneur série de x et y effectue l'addition successive des bits x_i et y_i en partant des bits les moins significatifs vers les bits les plus significatifs en propageant la retenue.

Question 19
Écrire la fonction `addSerie : mot -> mot -> mot` qui prend en arguments deux mots machine x et y de même longueur n et qui retourne le mot machine de longueur $n + 1$ représentant $x + y$.
Quel est le temps de calcul de ce circuit ?
Donner un ordre de grandeur du nombre de portes de ce circuit.

Question 20
Écrire la fonction `mux : circuit -> mot -> mot -> mot` qui prend en arguments un circuit s et deux mots machines de même longueur x et y et qui retourne le mot machine z tel que z_i vaut x_i si s' fourni par s vaut 1 et y_i si s' vaut 0.

On peut calculer l'addition plus rapidement en coupant les mots machine x et y à additionner en deux parties basses x_b, y_b contenant les bits les moins significatifs et deux parties hautes x_h, y_h contenant les bits les plus significatifs. Pour ne pas attendre le résultat de la retenue de $x_b + y_b$ pour calculer $x_h + y_h$, on peut précalculer les résultats de l'addition de x_h et y_h avec la retenue valant 0 et celle valant 1. Puis le véritable résultat de la retenue de $x_b + y_b$ permet de donner le résultat voulu pour $x_h + y_h + r$.

Question 21
écrire la fonction `addPar : mot -> mot -> circuit -> mot` qui prend en arguments deux mots machine x et y de longueur n et un circuit r fournissant une retenue r' et qui retourne le mot machine de longueur $n + 1$ représentant $x + y + r'$.
Quel est le temps de calcul de ce circuit ?
Donner un ordre de grandeur du nombre de portes de ce circuit.

Question 22
Pourquoi ne pas utiliser un PLA pour réaliser ces additionneurs ?
Quels auraient été les avantages/désavantages ?

Solutions

Solution de la question 1
$\overline{x_0} \wedge \overline{x_1} \wedge x_2 \vee \overline{x_0} \wedge x_1 \wedge x_2 \vee x_0 \wedge \overline{x_1} \wedge \overline{x_2} \vee x_0 \wedge x_1 \wedge \overline{x_2}$.

Solution de la question 2
$h(p,q,r) = x_1 \wedge \left(\overline{x_2} \vee x_2\right) \wedge \overline{x_3} \vee \overline{x_1} \wedge \left(\overline{x_2} \vee x_2\right) \wedge x_3 = x_1 \wedge \overline{x_3} \vee \overline{x_1} \wedge x_3$

Solution de la question 3
Si $x = 2^p$ alors il est représenté par $(x_0, \ldots, x_{n-1})$ avec $x_i = 0$ sauf $x_p = 1$. $x-1$ est représenté par $(x'_0, \ldots, x'_{n-1})$ avec $x'_i = 1$ pour $i < p$ et $x'_i = 0$ sinon. On a bien `x land (x - 1) = 0`.
Réciproquement, si $x > 0$ n'est pas une puissance de 2 alors il existe p tel que $2^p < x < 2^{p+1}$: p est le plus grand indice tel que $x_p = 1$ dans la représentation de x.
On a alors $2^p \leq x - 1$ donc on a aussi $x'_p = 1$ dans la représentation de $x' = x - 1$.
On en déduit que `x" = x land (x - 1)` vérifie $x''_p \neq 0$ donc est non nul d'où l'équivalence.

Solution de la question 4
Un bit de `x lxor y` est nul si et seulement si les bits correspondants de x et y sont égaux.
Le nombre de bits non nuls de `x lxor y` est donc égal à la distance de Hamming $d(x,y)$.
Ainsi $d(x,y) = 1$ si et seulement si `x lxor y` a un seul bit non nul c'est-à-dire est une puissance de 2 d'où

```
let aDistUn x y =
  let u = x lxor y in
  u <> 0 && u land (u-1) = 0;;
```

Solution de la question 5
`m land p = p`, `m lor p = m` et `m lxor p = n`

Solution de la question 6
On commence par enlever les occurrences de la tête dans la queue de la liste puis on applique récursivement ce processus à la queue :

```
let rec enlever x liste =
  match liste with
  |[] -> []
  |t::q -> if x = t
           then enlever x q
           else t::(enlever x q);;

let rec unique = function
  |[] -> []
  |t::q -> t::(unique (enlever t q));;
```

`enlever` a une complexité proportionnelle à la longueur de la liste et on l'appelle pour chaque queue de liste : la complexité est un $\mathcal{O}(\ell^2)$.

Solution de la question 7
Si deux produits ont une variable éliminable alors ils ont les mêmes variables qui apparaissent donc ont le même masque. De plus l'un a exactement une variable positive de plus que l'autre donc les valeurs ont une distance de Hamming de 1.
Inversement si deux produits ont le même masque alors ils ont les mêmes variables et si, de plus, les valeurs ont une distance de Hamming de 1 c'est qu'une variable unique est positive dans l'un et négative dans l'autre : cette variable est éliminable.

```
let varEliminable x y =
  (x.m = y.m)  && (aDistUn x.p y.p);;
```

La complexité est constante.

Solution de la question 8
La variable à éliminer est définie par la différence entre les valeurs, obtenu par `x.p lxor y.p = v'` donc le nouveau masque est `x.m lxor v'`. La valeur à garder est la plus petite des deux ou leur `land`.

```
let unifier x y =
  let v' = x.p lxor y.p in
  {m = x.m lxor v'; p = (min x.p y.p)};;
```

Solution de la question 9
Pour calculer toutes les unifications, on doit en tester la possibilité pour toute paire d'éléments de la forme. Pour éviter les doublons on ne teste un élément qu'avec ceux qui suivent.
On commence par les unifications d'un produit et d'une liste

```
let rec unifParUn p liste =
   match liste with
   |[] -> []
   |t::q -> if varEliminable p t
            then (unifier p t)::(unifParUn p q)
            else unifParUn p q;;
```

Il reste alors à réunir tous ces produits

```
let rec  unifications f =
   match f with
   |[] -> []
   |t::q -> (unifParUn t q)@(unifications q);;
```

On compare toutes les paires de produits : il y en a $\frac{\ell(\ell-1)}{2}$ d'où une complexité en $\mathcal{O}(\ell^2)$.
On peut n'utiliser qu'une seule fonction ; est-ce plus simple ?

```
let unifications f  =
   let rec aux premiers seconds =
   match premiers, seconds with
   |[], _ -> []
   |t1::q1, [] -> aux q1 q1
   |t1::q1, t2::q2 -> if varEliminable t1 t2
                      then (unifier t1 t2)::(aux premiers q2)
                      else aux premiers q2
   in aux f f;;
```

On compare toutes les paires de produits : il y en a $\frac{\ell(\ell-1)}{2}$ d'où une complexité en $\mathcal{O}(\ell^2)$

Solution de la question 10
On a vu $h_0 = \overline{x_1} \wedge \overline{x_2} \wedge x_3 \vee \overline{x_1} \wedge x_2 \wedge x_3 \vee x_1 \wedge \overline{x_2} \wedge \overline{x_3} \vee x_1 \wedge x_2 \wedge \overline{x_3}$.
On calcule $h_1 = \overline{x_1} \wedge x_3 \vee x_1 \wedge \overline{x_3}$, h_2 est vide

Solution de la question 11
On ajoute en tête les nouveaux produits de la tête de la liste.

```
let rec developper liste =
  match liste with
  |[] -> failwith "Erreur : liste vide"
  |t::q -> let a = nouveauxProduits t in
           if a=[]
           then liste
           else developper (a::liste);;
```

La complexité de `nouveauxProduits` a_i est en $O(\ell_i^2)$ donc la complexitÉ de la fonction `developper` par rapport aux longueurs ℓ_i est en $O\left(\sum_{i=0}^{k} \ell_i^2\right)$.
On peut diminuer la longueur des liste en remplaçant la ligne 4 par
`|t::q -> let a = unique(nouveauxProduits t) in`
La complexité de `unique` est en $O(\ell^2)$ donc cette instruction ne change pas la complexité asymptotique dans le pire des cas mais peut améliorer les tailles des listes.

Solution de la question 12
On a q admet comme littéraux tous les littéraux de p donc $q = p \wedge q'$ et $p \vee q$.
On peut donc éliminer q.

Solution de la question 13
Les conditions de couverture sont équivalentes à l'inclusion du masque de p dans celui de q et l'inclusion des variables positives. Elles s'écrivent `p.m land q.m = p.m` et `p.p land q.p = p.p`

```
let couvre p q =
  (p.m land q.m = p.m) && (p.p land q.p = p.p);;
```

Solution de la question 14
La forme f_0 est une f.n.d. de produits qui ont tous le même masque valant $2^{n+1} - 1$ car toutes les variables ont une valeur fixé dans la représentation issue de la table de vérité..
Les produits de f_1 ont un masque avec $n-1$ bits non nuls et, à chaque étape le nombre de bits du masque diminue de 1.
Ainsi un produit de f_i ne peut couvrir un produit de a_j que si $i > j$.
On va donc enlever les produits couverts à partir de f_{k-1} jusqu'à f_0 en construisant une liste des produits non couverts par adjonctions successives.

1. Extractions des produits d'une liste non couverts par un produit :

```
let rec nonCouvertsParUn liste p =
   match liste with
   |[] -> liste
   |t::ll -> if couvertPar t p
             then nonCouvertsParUn ll p
             else t::(nonCouvertsParUn ll p);;
```

2. Extractions des produits d'une f.n.d. non couverts par les produits d'une liste :

```
let rec nonCouvertsParListe liste1 liste2 =
  match liste2 with
  |[] -> liste1
  |t::ll2 -> let ll1 = nonCouvertsParUn liste1 t in
             nonCouvertsParListe ll1 ll2;;
```

3. Les réductions successives d'une liste de f.n.d.s à partir d'une liste initiale

```
let rec reductions listef.n.d.s listeProduits =
  match listef.n.d.s with
  |[] -> listeProduits
  |s::lS -> let lP = nonCouvertsParListe s listeProduits in
            reductions lS (lP@listeProduits);;
```

4. On appelle avec les conditions initiales :

```
let reduire a =
  match (developper a) with
  |[] -> failwith "Fonction sans variable"
  |ak::listeS -> reductions listeS ak;;
```

La complexité de `nonCouvertsParUn liste p` est proportionnelle à la longueur de `liste`.
Donc la complexité de `nonCouvertsParListe liste1 liste2` est proportionnelle au produit des longueurs de `liste1` et de `liste2`.
À chaque étape de `reductions` on applique `nonCouvertsParListe` à la liste a_i et à une concaténation de listes a'_i ; la liste concaténée est ensuite augmentée de la liste extraite qui a au plus autant d'éléments que a_i. Il faut ajouter à la complexité de `nonCouvertsParListe` la la complexité de la fusion des listes. La complexité totale est donc majorée par

$$\sum_{i=k-1}^{0} |a_i||a'_i| + |a_i| \text{ avec } a'_{k-1} = a_k \text{ et } |a'_{i-1}| \leq |a'_i| + |a_i| \text{ d'où } |a'_i| \leq \sum_{j=i+1}^{k} |a_j|.$$

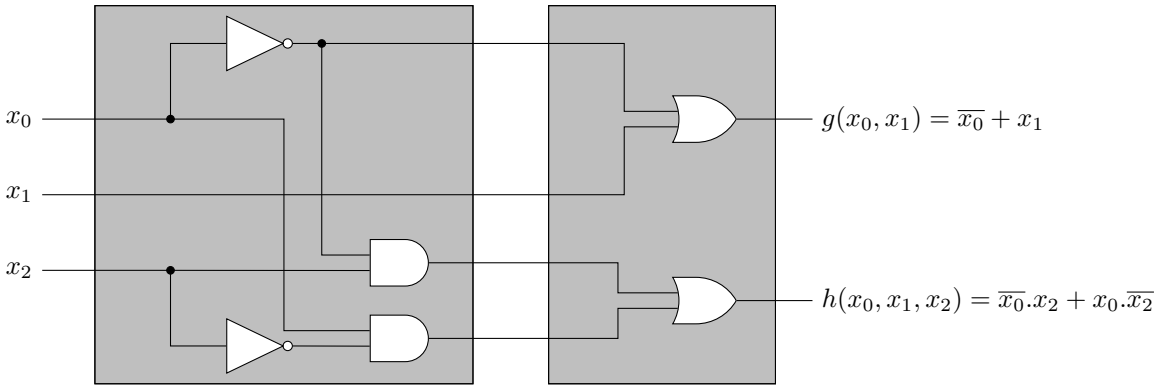
On aboutit à $C \leq \sum_{i=0}^{k-1} |a_i| \left(1 + \sum_{j=i+1}^{k} |a_j|\right) \leq \sum_{i=0}^{k-1} |a_i| \left(\sum_{j=0}^{k} |a_j|\right) \leq \left(\sum_{j=0}^{k} |a_j|\right)^2$.

Il faut aussi ajouter la complexité de `developper` : $C \leq 2\left(\sum_{i=0}^{k} |a_i|\right)^2$.

Solution de la question 15
Le cardinal de f_0 est 2^n.
Le cardinal de f_i est majoré par le nombre de produits dont le masque à un nombre de bits égal à $n-i$. Il y a $\binom{n}{n-i}$ tels masques puis, pour chaque masque 2^{n-i} valeurs possibles. Ainsi

$$C(n) \leq 2\left(\sum_{i=0}^{n} 2^{n-i}\binom{n}{n-i}\right)^2 = 2\big((1+2)^n\big)^2 = 2.9^n.$$

Solution de la question 16



Solution de la question 17
Les liaisons entre les deux parties correspondent aux produits utilisés dans les fonction f_0, ..., f_{m-1}. On peut en réduire le nombre en choisissant des formes avec peu de produits et en repérant les produit utilisés dans plusieurs fonctions.

Solution de la question 18
Le quotient et le reste se calculent avec une table de vérité :

x'	y'	r'	reste	quotient
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Le reste s'écrit $\overline{x_0}.\overline{x_1}.x_2 + \overline{x_0}.x_1.\overline{x_2} + x_0.\overline{x_1}.\overline{x_2} + x_0.x_1.x_2$
Le quotient s'écrit $\overline{x_0}.x_1.x_2 + x_0.\overline{x_1}.x_2 + x_0.x_1.\overline{x_2} + x_0.x_1.x_2$. $x_0.x_1.x_2$ permet une réduction avec chacun des 3 autres termes donc le quotient peut s'écrire $x_1.x_2 + x_0.x_2 + x_0.x_1$.
Ainsi on obtient

```
let bitAdd x y r =
  Ou(Ou(Et(Non x,Et(Non y,r)),Et(Non x,Et(y,Non r))),
     Ou(Et(x,Et(Non y,Non r)),Et(x,Et(y,r))));;

let bitRetenue x y r =
  Ou(Et(x,y),Ou(Et(x,r),Et(y,r)));;
```

Solution de la question 19
On suit les indications de l'énoncé.
Les étapes doivent être successives et à chaque étape on passe par 5 portes (au maximum) donc le temps est $5n$.
Chaque étape crée 11 portes ET, 5 portes OU et 6 portes NON alors que les données initiales ne demandent pas de portes : il y a $22n$ portes dans le circuit.

```
let addSerie x y =
  let n = vect_length x in
  let z = make_vect (n+1) (Bit 0) in
  let retenue = ref (Bit 0) in
  for i = 0 to (n-1) do
    z.(i) <- bitAdd x.(i) y.(i) !retenue;
    retenue := bitRetenue x.(i) y.(i) !retenue done;
  z.(n) <- !retenue;
  z;;
```

Solution de la question 20
L'énoncé original était encore plus flou.
On peut imaginer que, pour chaque circuit du mot, on veut un circuit qui vaut la valeur de x_i si s' vaut 1, et y si s' vaut 0. On arrive à $z_i = y_i.s + x_i.\overline{s}$.

```
let mux s x y =
  let n = vect_length x in
  let z = make_vect n (Bit 0) in
  for i = 0 to (n-1) do
    let a = Et(x.(i), Non s ) in
    let b = Et (y.(i), s ) in
    z.(i) <- Ou(a, b) done;
  z;;
```

Solution de la question 21

Voici une interprétation possible de la question, sans doute perfectible.
L'énoncé suggère une méthode diviser-pour-régner : on calcule récursivement deux couples f.n.d.-retenue, le premier s'il n'y a pas de retenue et le second avec retenue initiale. On effectue le travail récursivement en coupant en deux à chaque étape.
Il faut donc, à chaque niveau, deux mots (f.n.d. avec ous sans retenue initiale) mais il faut aussi gérer les deux retenues possibles donc deux circuits.
Ainsi une première fonction (`addition` ici) calcule récursivement ces 4 éléments.

- x_b et y_b renvoient $s_{0,b}$, $s_{1,b}$, $r_{0,b}$ et $r_{1,b}$; c'est-à-dire $x_b + y_b + 0 = s_{0,b}$ avec la retenue $r_{0,b}$ et $x_b + y_b + 1 = s_{1b}$ avec la retenue $r_{1,b}$
- De même x_h et y_h renvoient $s_{0,h}$, $s_{1,h}$, $r_{0,h}$ et $r_{1,h}$

Alors $x + y + 0$ renvoie $[s_{0,b}, ss_{0,h}]$ avec la retenue r_0 où $ss_{0,h}$ est le mux entre $s_{0,h}$ et $s_{1,h}$ en fonction de $r_{0,b}$ et r_0 est le mux entre $r_{0,h}$ et $r_{1,h}$ en fonction de $r_{0,b}$.
De même pour $x + y + 1$.
On peut ensuite additionner mais je ne comprends pas l'intérêt de la retenue.
On a besoin du mux de deux circuits :

```
let mux1 s x y =
  let a = Et(x, Non s ) in
  let b = Et (y, s ) in
  Ou(a, b);;
```

On a besoin de l'assemblage de deux tableaux :

```
let assemble t1 t2 =
  let n1 = vect_length t1 in
  let n2 = vect_length t2 in
  let t = make_vect (n1+n2) (Bit 0) in
  for i = 0 to n1 - 1 do
    t.(i) <- t1.(i) done;
  for i = 0 to n2 - 1 do
    t.(i+n1) <- t2.(i) done;
  t;;
```

On calcule le cas terminal (deux circuits à f.n.d.r) :

```
let addition1 x y =
  let sans = make_vect 1 (Bit 0) in
  let avec = make_vect 1 (Bit 0) in
  sans.(0) <- bitAdd x y (Bit 0);
  avec.(0) <- bitAdd x y (Bit 1);
  (sans, avec, bitRetenue x y (Bit 0), bitRetenue x y (Bit 1));;
```

Il reste la fonction récursive

```
let rec addition x y a b=
  if b-a = 0
  then addition1 x.(a) y.(a)
  else let c = (a+b)/2 in
       let (s0b, s1b, r0b, r1b) = addition x y a c in
       let (s0h, s1h, r0h, r1h) = addition x y (c+1) b in
       let ss0h = mux r0b s0h s1h in
       let ss1h = mux r1b s0h s1h in
       let r0 = mux1 r0b r0h r1h in
       let r1 = mux1 r1b r0h r1h in
       let s0 = assemble s0b ss0h in
       let s1 = assemble s1b ss1h in
       (s0, s1, r0, r1);;
```

Puis la fonction

```
let addPar x y =
  let n = vect_length x in
  let (s0, s1, r0, r1) = addition x y 0 (n-1) in
  assemble s0 [|r0|];;
```

Chaque étape nécessite 3 temps de plus (pour les `mux`) donc le temps de calcul pour $n = 2^p$ vaut $3p$ plus le temps de calcul de `addition1`, 5.
Pour $n = 64 = 2^6$ on trouve un temps de calcul de 23 à comparer au temps de calcul de 320 pour `addSerie`.

Le nompre de portes pour $n = 2^p$, N_p vérifie $N_{p+1} = 2N_p + 4.2^{p+1} + 2.4$ car `mux` demande 4 portes pour chaque élément.
Si on pose $C_p = 2^{-p}N_p$ on a $C_{p+1} = C_p + 4 + 8.2^{-p-1}$ donc C_p est de l'ordre de $4p + 8 + C_0$ avec $C_0 = 22$. On en déduit N_p de l'ordre $4p2^p + 30.2^p$: on augmente le nombre de portes.
Pour $n = 64$ on trouve $54n$ au lieu de des $22n$ dans le cas de l'addition série.

Solution de la question 22
L'avantage des PLA semble être la facilité de calcul des circuits. Par contre ils demandent un très grand nombre de portes.