TP Caml 2: Logique et circuits

lionel.rieg@ens-lyon.fr

30 septembre & 7 octobre 2008

Remarques sur le TP précédent :
- Les types sont **extrêmement** importants : une fonction qui n'a pas le bon type est fausse, même si son code fait ce qu'il faut. De plus, le type qu'on vous donne est, sauf erreur, le meilleur auquel vous avez droit.
- Écrivez des fonction *currifiées* chaque fois que cela est possible car elles sont plus souples (applications partielles).
- Indentez *toujours* vos fonctions, ça vous aidera à les comprendre (et moi aussi !) et pour peu que votre éditeur le fasse pour vous, toute erreur d'indentation dénote une erreur de syntaxe. De plus, cela évite le "foulage de cerveau du correcteur" (© S. Gonnord).
- Essayer autant que possible de donner des noms significatifs à vos variables car dès que le corps de la fonction fait plus de trois lignes cela aide *vraiment* à comprendre.
- S'il vaut mieux mettre trop de parenthèses que pas assez, en mettre vraiment trop peut devenir tout aussi agaçant. Par exemple, l'application de fonction ne nécessite pas de parenthèses autour de l'argument : écrire f x plutôt que f(x), sauf si cela introduit une ambiguïté, comme lorsque x est composé.
- Si vous avez du mal à voir comment écrire une fonction (une fois que vous avez compris ce qu'elle est sensée faire), commencez par écrire le **let** ... = avec tous les arguments qu'elle prend pour voir ce dont vous disposez pour l'écrire. Par exemple, l'écriture de la composition de fonctions est beaucoup plus simple une fois qu'on s'est donné un élément sur lequel appliquer la composée.

1 Formules logiques

On va s'intéresser à la synthèse d'un circuit à partir d'une formule logique. On considère le type de formule suivant :

```
type formula =
  | FVar of string
  | FNot of formula
  | FAnd of formula list
  | FOr of formula list ;;
```

On choisit d'utiliser des opérateur « et » et « ou » n-aires plutôt que simplement binaires (d'où l'utilisation de listes et non de couples dans leur constructeurs) car cela permet des simplifications plus aisées : transformer $x \wedge y \wedge z \wedge x \wedge y$ en $x \wedge y \wedge z$ est évident lorsqu'on utilise des listes mais beaucoup moins lorsqu'il faut parcourir des arbres binaires. Cependant, on ne s'attaquera pas ici au problème de la simplification de formules.

De plus, une formule logique usuelle contient souvent deux autres connecteurs, l'implication et l'équivalence même si ces connecteurs ne sont pas nécessaires. On décide donc d'avoir un type de formule étendue :

```
type extended_formula =
  | EVar of string
  | ENot of extended_formula
  | EAnd of extended_formula list
  | EOr of extended_formula list
  | EImply of extended_formula * extended_formula
  | EEquiv of extended_formula * extended_formula ;;
```

► **Question 1** Convertir une formule étendue en une formule simple. La fonction aura le type

```
ext2form : extended_formula → formula
```

À présent, on peut se limiter aux formules simples et faire la conversion avec les formules complexes.

2 Construction de circuits

2.1 Traduction directe

On s'intéresse tout d'abord à la transcription directe d'une formule booléenne en un circuit dont le type sera :

```
type AONCircuit =
  | CWire of string
  | CNot of AONCircuit
  | CAnd of AONCircuit * AONCircuit
```

Pour cela, il va nous falloir en premier lieu convertir nos « et » et nos « ou » n-aires en « et » et « ou » binaires.

► **Question 2** Écrire une fonction qui découpe une liste en deux et renvoie un couple constitué des deux moitiés. L'ordre n'a pas besoin d'être conservé. Son type est :

```
half : 'a list → 'a list * 'a list
```

► **Question 3** Écrire une fonction de traduction d'un « et » n-aire en un arbre équilibré de « et » binaires en utilisant une stratégie de diviser pour régner et la fonction de la question précédente. Faire de même pour « ou ». Leur type seront :

```
convert_and : AONCircuit list → AONCircuit
convert_or : AONCircuit list → AONCircuit
```

► **Question 4** Écrire à présent la fonction de conversion d'une formule booléenne en circuit :

```
formula2circuit : formula → AONCircuit
```

2.2 Coût de construction

On va chercher à évaluer ici les circuits en termes de performances et de coût. En supposant que toutes les portes prennent le même temps à être traversées, le temps de calcul d'un circuit est sa profondeur.

► **Question 5** Écrire une fonction qui calcule la profondeur d'un circuit. Elle sera de type :

```
depth : AONCircuit → int
```

Construire le circuit le plus rapide est intéressant mais peut être très coûteux. Par exemple, on peut toujours construire un circuit de profondeur $2\log\lfloor 2n-1\rfloor+3$ pour n variables d'entrée (exercice : comment ?) mais il peut avoir une taille exponentielle en n. Pour éviter de payer trop cher, on va s'intéresser à la taille d'un circuit, c'est à dire au nombre de portes logiques qu'il contient.

► **Question 6** Écrire la fonction qui calcule la taille d'un circuit. Son type sera :

```
size : AONCircuit → int
```

La priorité des opérateurs nous dispense de mettre des parenthèses : les fonctions (ici size) sont prioritaires sur les opérateurs infixes (ici + mais aussi ::).

3 D'autres types de circuits

3.1 Des circuits à une porte

Au moment de lancer la fabrication de nos circuits, on s'est rendu compte qu'une promotion sur les portes « Nor » et « Nand » les rendait très attractives, au point de vouloir remplacer toutes les autres par l'une ou l'autre de ces deux portes. On définit donc deux nouveaux types de circuits :

```
type norCircuit = OWire of string | Nor of norCircuit * norCircuit ;;

type nandCircuit = AWire of string | Nand of nandCircuit * nandCircuit ;;
```

Mais plutôt que de repartir des formules booléennes avec des opérateurs n-aires, on décide de convertir les circuits précédents en l'un de ces nouveaux circuits.

► **Question 7** Écrire une fonction de conversion d'un circuit de type AONCircuit en circuit de type norCircuit.

► **Question 8** Faire de même vers les circuits de type nandCircuit.

3.2 Cherchons le meilleur circuit

Comme le prix des portes « Nand » et « Nor » a remonté, il n'est plus évident de savoir quelle solution est la plus rentable. On va donc s'intéresser de façon plus précise au coût de construction et à la latence des circuits.

On distingue à présent les différents prix des portes et du fil et leur temps de latence. On se donne ainsi un type qui représente les valeurs de chaque porte, en coût ou bien en latence :

```
type gates = {Wire : float ; Not : float ; Or : float ; And : float ;
              Nor : float ; Nand : float };;
```

► **Question 9** Modifier la fonction de calcul de latence d'un circuit AONCircuit pour qu'elle prenne en compte les valeurs de chaque porte. Elle sera de type

```
AONlatency : gates → AONCircuit → float
```

de sorte que AONlatency time circuit vaudra le temps de latence du circuit circuit pour des latences de portes données dans time.

► **Question 10** Faire de même pour le coût de construction d'un circuit de type AONCircuit : AONcost costs circuit vaudra le coût total du circuit circuit pour des coûts unitaires de portes données dans costs et AONcost aura le même type qu'à la question précédente.

► **Question 11** Reprendre les deux questions précédentes pour les circuits de types norCircuit et nandCircuit.

À présent qu'on sait calculer les coûts et performances de chaque circuit, il est temps de faire un comparatif.

► **Question 12** Écrire une fonction min_cost de type

```
min_cost : gates → gates → float → extended_formula → float
```

telle que min_cost time costs ratio formula donne la valeur du meilleur circuit entre les trois types possibles, celle-ci étant calculée avec la formule suivante : $\frac{\text{ratio}}{\text{latence}\times\text{coût}}$ où la latence et le coût sont calculées sur un circuit avec les fonctions précédentes. Le terme ratio permet de jauger jusqu'à quel point on est prêt à payer cher pour avoir un circuit performant.

4 Meilleure conversion

Dans la partie précédente, on a construit les circuits utilisant uniquement les portes « Nand » et « Nor » comme des traductions de circuits utilisant les portes « et », « ou » et « non ». Cette conversion entraîne des pertes de performances : par exemple, l'implication $p \Rightarrow q$ peut se récrire $\neg p \vee q$, dont la traduction donne $((x \otimes x) \otimes (x \otimes x)) \otimes (y \otimes y)$ (où $\otimes$ dénote « Nand ») alors que $x \otimes (y \otimes y)$ suffit.

► **Question 13** Corriger ce problème, c'est à dire mettre en oeuvre l'une de deux solutions suivante :
- écrire une fonction de traduction de formule étendue vers un circuit de type nandCircuit
- trouver dans quels cas où on peut simplifier le circuit produit la méthode de la partie précédente et implémenter la simplification

► **Question 14** Faire de même pour les circuits norCircuit.

5 Solutions

▸ **Question 1** Ne pas oubliez de nommer des résultats identiques sans quoi Caml Light va les recalculer pour rien (pour ceux qui se poseraient la question, on ne peut pas toujours dire que deux appels identiques peuvent être remplacés par un seul car il peut y avoir des effets de bord dans la fonction appelée).

```
let rec ext2form = function
  | EVar v -> FVar v
  | ENot f -> FNot (ext2form f)
  | EAnd l -> FAnd (map ext2form l)
  | EOr l -> FOr (map ext2form l)
  | EImply (fg, fd) -> FOr [FNot (ext2form fg); ext2form fd]
  | EEquiv (fg, fd) ->
      let fg2 = ext2form fg and fd2 = ext2form fd in
        FAnd [FOr [FNot fg2; fd2]; FOr [FNot fd2; fg2 ]];;
```

▸ **Question 2** Une idée naturelle consiste à écrire une fonction de découpe puis de couper la liste en son milieu, ce qui donne le code suivant :

```
let rec coupe deb fin n =
  if n = 0 then (deb, fin) else
    match fin with
      | [] -> (deb, [])
      | t :: q -> coupe (t :: deb) q (n-1);;

let half l = coupe [] l ((list_length l) / 2);;
```

La solution ci-dessous a l'avantage d'être récursive terminale et de ne parcourir qu'une seule fois la liste mais elle crée complètement les deux listes alors que la première méthode partage la seconde moitié de la liste avec la liste initiale.

```
let half l =
  let rec aux left right = function
    | [] -> left, right
    | [t] -> t :: left, right
    | t1 :: t2 :: q -> aux (t1 :: left) (t2 :: right) q
  in
    aux [] [] l;;
```

Enfin la dernière est intéressante car elle ne fait qu'une ligne mais pour la comprendre vous avez besoin de savoir ce que fait it_list : c'est une sorte de list_map qui se souvient des résultats intermédaires et les passe en argument à la fonction à itérer. Plus précisément, it_list f a [b1; b2; ... ; bN] vaut f (... (f (f a b1) b2) ...) bn.

```
let half l = it_list (fun (ll, rl) elt -> (rl, elt :: ll)) ([], []) l;;
```

On ajoute chaque élément à l'une des deux listes moitiés (ici la seconde) puis on les permute pour que l'élément suivant soit ajouté sur l'autre liste. Apprendre ce que font it_list et sa cousine list_it donne souvent une façon élégante de résoudre des problèmes sur les listes.

▸ **Question 3**

```
let rec convert_and l = (* FAUX !!! *)
  let (l,r) = half l in
    CAnd(convert_and l, convert_and r);;
```

La version ci-dessus est fausse car il n'y a pas de cas d'arrêt : elle ne va faire que des appels à half sans jamais en utiliser le résultat, ce qui explique qu'elle puise prendre en argument une 'a list car elle ne se sert pas des éléments qui composent la liste pour construire le circuit. Une bonne solution doit incorporer les cas d'arrêts, à savoir celui où la liste ne contient qu'un élément (la liste vide est exclue car on suppose les connecteurs bien formés) :

```
let rec convert_and = function
  | [] → failwith "empty and"
  | [t] → t
  | l →
      let left, right = half l in
        CAnd (convert_and left, convert_and right);;

let rec convert_or = function
  | [] → failwith "empty or"
  | [t] → t
  | l →
      let left, right = half l in
        COr (convert_or left, convert_or right);;
```

Ne pas oublier de bien dé-construire l'appel à half par un **let** ... **in** et non utiliser fst et snd car cela requiert deux appels à half sans que cela soit nécessaire.

► **Question 4**

```
let rec formula2circuit = function
  | FVar x → CWire x
  | FNot f → CNot (formula2circuit f)
  | FOr l → convert_or (map formula2circuit l)
  | FAnd l → convert_and (map formula2circuit l);;
```

► **Question 5**

```
let rec depth = function
  | CWire _ → 0
  | CNot c → 1 + depth c
  | CAnd (lc, rc) → 1 + max (depth lc) (depth rc)
  | COr (lc, rc) → 1 + max (depth lc) (depth rc);;
```

► **Question 6**

```
let rec size = function
  | CWire _ → 0
  | CNot c → 1 + size c
  | CAnd (lc, rc) → 1 + size lc + size rc
  | COr (lc, rc) → 1 + size lc + size rc ;;
```

► **Question 7** Dans cette question et la suivante, il est très important de penser à nommer les résultats qui sont utilisés plusieurs fois, sans quoi la complexité de l'algorithme (en temps **et** en espace) passe de $\Theta(n)$ à $\Theta(2^n)$! (exercice : trouver pourquoi)

```
let rec toNor = function
  | CWire w → OWire w
  | CNot (COr (lc, rc)) → Nor (toNor lc, toNor rc)
  | CNot c → let nor_c = toNor c in Nor (nor_c, nor_c)
  | COr (lc, rc) →
      let nor_lc = toNor lc and nor_rc = toNor rc in
        Nor (Nor (nor_lc, nor_rc), Nor (nor_lc, nor_rc))
  | CAnd (lc, rc) →
      let nor_lc = toNor lc and nor_rc = toNor rc in
        Nor (Nor (nor_lc, nor_lc), Nor (nor_rc, nor_rc));;
```

► **Question 8** Même remarque qu'à la question précédente.

```
let rec toNand = function
  | CWire w → AWire w
  | CNot (CAnd (lc, rc)) → Nand (toNand lc, toNand rc)
  | CNot c → let nand_c = toNand c in Nand (nand_c, nand_c)
  | COr (lc, rc) →
      let nand_lc = toNand lc and nand_rc = toNand rc in
      Nand (Nand (nand_lc, nand_lc), Nand (nand_rc, nand_rc))
  | CAnd (lc, rc) →
      let nand_lc = toNand lc and nand_rc = toNand rc in
      Nand (Nand (nand_lc, nand_rc), Nand (nand_lc, nand_rc ));;
```

► **Question 9**

```
let rec AONlatency time = function
  | CWire w → time.Wire
  | CNot c → time.Not +. AONlatency time c
  | CAnd (lc, rc) → time.And +. max (AONlatency time lc) (AONlatency time rc)
  | COr (lc, rc) → time.Or +. max (AONlatency time lc) (AONlatency time rc );;
```

► **Question 10**

```
let rec AONcost costs = function
  | CWire w → costs.Wire
  | CNot c → costs.Not +. AONcost costs c
  | CAnd (lc, rc) → costs.And +. AONcost costs lc +. AONcost costs rc
  | COr (lc, rc) → costs.Or +. AONcost costs lc +. AONcost costs rc ;;
```

► **Question 11**

```
let rec Nandlatency time = function
  | AWire _ → time.Wire
  | Nand (lc, rc) →
      time.Nand +. max (Nandlatency time lc) (Nandlatency time rc );;

let rec Nandcost costs = function
  | AWire w → costs.Wire
  | Nand (lc, rc) → costs.Nand +. Nandcost costs lc +. Nandcost costs rc ;;
```

Ce sont les écritures les plus évidentes pour ces deux fonctions. Cependant, comme ces circuits ne sont constitués que d'un seul type de porte, les calculs peuvent se ramener à des calculs de taille et de profondeur des circuits. Pour le coût, il faut remarquer que dans un arbre binaire strict, le nombre de feuilles est exactement le nombre de noeuds internes plus un. L'intérêt de cette écriture vient du fait que les calculs en nombres entiers sont plus rapides et beaucoup plus précis que ceux sur des flottants.

```
let rec Norlatency time circuit =
  let rec depth = function
    | OWire _ → 0
    | Nor (lc, rc) → 1 + max (depth lc) (depth rc)
  in
    time.Nor *. float_of_int (depth circuit ) +. time.Wire;;

let rec Norcost costs circuit =
  let rec size = function
    | OWire _ → 0
    | Nor (lc, rc) → 1 + size lc + size rc
  in
    (costs.Nor +. costs.Wire) *. ( float_of_int (size circuit )) +. costs.Wire;;
```

▸ **Question 12**

```
let min_cost time cost ratio formula =
  let min3 x y = min (min x y) in
  let AONcircuit = formula2circuit (ext2form formula) in
  let NandCircuit = toNand AONcircuit in
  let NorCircuit = toNor AONcircuit in
    ratio /.
      (min3
         ((AONcost cost AONcircuit) *. (AONlatency time AONcircuit))
         ((Nandcost cost NandCircuit) *. (Nandlatency time NandCircuit))
         ((Norcost cost NorCircuit) *. (Norlatency time NorCircuit )));;
```

▸ **Question 13** On choisit évidemment la seconde solution car elle nous évite de tout refaire. Le seul cas pour lequel il y a simplification est celui d'une double négation, (*i.e.* $((x \otimes x) \otimes (x \otimes x))$) qui peut se remplacer par un fil (*i.e.* x).

```
let rec simplify_nand = function
  | AWire w → AWire w
  | Nand (x, y) →
      match simplify_nand x, simplify_nand y with
        | Nand (xx, xy), Nand (yx, yy) →
            if xx = xy && xy = yx && yx = yy
            then xx
            else Nand (Nand (xx, xy), Nand (yx, yy))
        | xx, yy → Nand (xx, yy);;
```

▸ **Question 14**

```
let rec simplify_nor = function
  | OWire w → OWire w
  | Nor (Nor (x, y), Nor (z, t))
      when x == y && y == z && z == t → simplify_nor x
  | Nor (x, y) →
      match simplify_nor x, simplify_nor y with
        | Nor (xx, xy), Nor (yx, yy) →
            if xx = xy && xy = yx && yx = yy
            then xx
            else Nor (Nor (xx, xy), Nor (yx, yy))
        | xx, yy → Nor (xx, yy);;
```

La différence avec la question précédente se trouve dans le deuxième cas de filtrage. Le == est l'égalité physique, et non l'égalité structurelle : elle vérifie que les deux objets sont les mêmes en mémoire, pas forcément qu'ils ont des champs égaux. Cela veut dire qu'elle n'a pas besoin de parcourir les structures donc s'exécute en temps constant. Cela peut être intéressant ici car les simplifications interviennent surtout lorsqu'on enchaîne des recopies des mêmes objets (d'où un autre intérêt à nommer les résultats utilisés plusieurs fois).