Circuits

Tahina RAMANANANDRO
ramanana@clipper.ens.fr

Lycée Louis-le-Grand – MPSI CAML
4 juin 2008

1 Définition

1.1 Représentation en CAML

Une *porte* est un composant élémentaire du circuit ayant une ou plusieurs entrées *ordonnées* et *une et une seule sortie*, cette sortie pouvant être toutefois reliée à des entrées de plusieurs portes. Ainsi, une porte est définie par son type de connecteur (ET, OU, NON, etc.) et ses entrées, qui peuvent être reliées à des sorties de portes, ou bien à des entrées du circuit (des « fils qui pendent » à l'entrée).

Un *circuit* est alors défini par **l'ensemble de ses portes de sortie**, c'est-à-dire, des portes dont la sortie est connectée à un « fil qui pend » à la sortie du circuit. Les sorties d'un circuit sont ordonnées.

On convient donc que les fils de sortie du circuit doivent être nécessairement passés par une porte : une entrée du circuit ne peut pas être reliée directement à une sortie du circuit.

On définit les trois types de données suivants :

```
type 'a porte = {
  numero          : int;
  connecteur      : 'a;
  entrees         : 'a entree vect
}
and 'a entree =
| EntreeCircuit of int
| SortiePorte   of 'a porte
;;

(* Circuit : l'ensemble de ses portes de sortie *)
type 'a circuit == 'a porte vect
;;
```

Ces trois types de données sont *paramétrés par un type* (noté 'a), celui des connecteurs. En effet, dans le sujet, on va considérer des circuits construits à partir d'ensembles différents de connecteurs.

L'*arité* des portes est *a priori* quelconque. L'ordre des entrées d'une porte est défini par son ordre dans le champ vecteur entrees.

Dessiner le circuit représenté par l'objet circ suivant :

```
type connecteur = ET | OU | NON
;;

let circ : connecteur circuit =
 let p0 = {numero=0; connecteur=NON; entrees= [| EntreeCircuit 0 |] } in
 let p1 = {numero=1; connecteur=OU; entrees= [| EntreeCircuit 0; SortiePorte p0 |] }
 and p2 = {numero=2; connecteur=ET; entrees= [| EntreeCircuit 0; SortiePorte p0 |] }
 in [| p1; p2 |]
;;
```

1.2 Structure

Le champ numero de la porte ne joue pas de rôle dans la logique du circuit, mais dans sa structure : on veut assurer que **le numéro d'une porte est toujours strictement plus grand que les numéros des portes connectées à ses entrées**. Ceci permet d'assurer que le circuit n'est pas *cyclique*. De plus, pour se permettre de dresser la liste des portes d'un circuit, on exigera que deux portes d'un même circuit n'aient pas le même numéro.

Écrire une fonction :

`toutes_portes : 'a circuit -> 'a porte option`[1] `vect`

telle que, si c est un circuit, alors pour tout entier i, (toutes_portes c).(i) est :

– soit Some p où p est la porte du circuit c dont le numéro est i
– soit None, si le circuit ne possède pas de porte de numéro i

2 Sémantique : comportement des circuits

On veut connaître le comportement du circuit en fonction des bits reçus par chacun de ses fils d'entrée.

Le comportement des circuits dépend des connecteurs utilisés, et du sens que l'on donne aux portes utilisant ces connecteurs : on doit disposer non seulement d'un type t donnant les connecteurs autorisés, mais aussi d'une fonction `f : t -> bool vect -> boo` qui à un connecteur de ce type et un tableau de booléens en entrée de la porte renvoie le booléen de sortie de la porte. La donnée d'un type et d'une telle fonction f constitue un *système de connecteurs*.

Écrire des fonctions f convenables pour chacun des trois ensembles de connecteurs suivants :

```
type connecteur = ET | OU | NON ;;
type nand       = NAND          ;;
type nor        = NOR           ;;
```

On conviendra que :

– une porte ET ou NOR à zéro entrée sortira toujours le booléen **true**
– une porte OU ou NAND à zéro entrée sortira toujours le booléen **false**
– une porte NON à plus d'une entrée ne tiendra compte que de son entrée 0

Écrire ensuite une fonction :

`executer : ('a -> bool vect -> bool) -> 'a circuit -> bool vect -> bool vect`

telle que, si v est un tableau contenant les bits attribués à chaque entrée du circuit c, alors (executer f c v) est un tableau contenant les bits attribués à chaque sortie du circuit c. On pourra utiliser la fonction toutes_portes et construire un tableau contenant les bits attribués à la sortie de chaque porte.

3 Un circuit arithmétique : l'additionneur

On veut construire des circuits capables d'effectuer des opérations arithmétiques sur des nombres entiers représentés en binaire, chacun de leurs bits étant représenté par un fil.

Plaçons-nous ici dans le cas de l'addition. Souvenons-nous de la façon dont on additionnait les chiffres à la main en « posant les opérations » :

$$\begin{array}{cccccc} & 1 & & 1 & & \\ & & 1 & 1 & 8 & \\ + & 1 & 9 & 5 & 7 & \\ = & 2 & 0 & 7 & 5 & \end{array}$$

« 8 et 7 font 15 : j'écris 5, je retiens 1. 1 et 5 font 6, plus la retenue 1, soit 7 : j'écris 7, je retiens 0. 9 et 1 font 10, je n'avais pas de retenue auparavant, donc j'écris 0 et je retiens 1 », etc.

On veut tout d'abord additionner deux *chiffres* binaires. Pour cela, concevoir un circuit ayant trois entrées (les deux chiffres binaires à additionner et la retenue « de l'opération précédente ») et deux sorties (le chiffre binaire somme et la retenue « à propager à l'opération suivante »), et définir l'objet CAML correspondant, sous la forme d'une fonction :

[1] Pour tout type `'b`, le type `'b option` permet de représenter la présence (`Some` *valeur*) ou l'absence (`None`) d'une valeur de type `'b`. Il est déjà prédéfini en CAML (`type 'b option = Some of 'b | None`)

```
additionneur_elem :
int -> connecteur entree -> connecteur entree -> connecteur entree -> connecteur circuit
```

prenant en paramètres le numéro de la première porte à créer et les entrées de ce sous-circuit, et renvoie les portes de sortie de ce sous-circuit. En effet, cette fonction est destinée à être utilisée pour concevoir un circuit plus grand contenant des exemplaires de ce sous-circuit.

Combien de portes nécessite-t-il ? (dans le système de connecteurs ET/OU/NON).

On veut ensuite construire un additionneur n bits, où n est un entier strictement positif quelconque. Écrire une fonction :

`additionneur : int -> connecteur circuit`

tel que additionneur n construise un circuit à 2n entrées (les n bits de chacun des deux nombres à additionner) et (n+1) sorties (les bits du résultat). Pourquoi n+1 sorties sont-elles suffisantes ? À l'inverse, que se passerait-il si on n'avait que n sorties ?

On testera pour de « petites » valeurs de n ($n < 30$).

Que faut-il modifier pour construire un soustracteur ? un multiplieur ?

4 Circuits et systèmes complets

On dit qu'un système de connecteurs t est *complet* si, et seulement si, toute fonction booléenne $h : \mathbb{B}^e \to \mathbb{B}^s$ peut être représentée par un circuit à e entrées et s sorties, dont les connecteurs sont de type t, et tel que si les entrées sont constituées du tableau v, alors les sorties sont constituées du tableau $h(v)$.

Pour montrer qu'un système de connecteurs (représenté par son type t et sa fonction `f : t -> bool vect -> bool`) est complet, il suffit de construire une fonction :

`circuit_of_func : int -> int -> (bool vect -> bool vect) -> t circuit`

qui prend en entrée une fonction booléenne et le nombre d'entrées et de sorties à prendre en compte, et qui produit le circuit correspondant.

4.1 Le système MUX

Considérons le système suivant à un seul connecteur appelé *multiplexeur* ou MUX :

```
type mux = MUX ;;
let f_mux MUX b =
 if vect_length b < 3 then false else
 if b.(0) then b.(1) else b.(2)
;;
```

Montrer que la fonction booléenne constante **true** ne peut pas être représentée par un circuit dans ce système.

La rajouter et montrer que le système obtenu est complet, en construisant une fonction CAML :

`mux_circuit_of_func : int -> int -> (bool vect -> bool vect) -> mux circuit`

éventuellement par récurrence sur le nombre d'entrées à prendre en compte.

4.2 Le système ET/OU/NON

Construire une fonction :

`connecteur_of_mux : mux circuit -> connecteur circuit`

qui à partir d'un circuit du système MUX construit un circuit du système ET/OU/NON ayant *même sémantique* (et conservant les ordres des entrées et des sorties !). En déduire que le système ET/OU/NON est complet.

Imaginons que l'on construise une fonction « réciproque » :

`mux_of_connecteur : connecteur circuit -> mux circuit`

Quel problème algorithmique surviendrait au niveau des circuits produits par une telle fonction ?

4.3 Les systèmes NAND et NOR

Construire une fonction :

`nand_of_connecteur : connecteur circuit -> nand circuit`

qui à partir d'un circuit ET/OU/NON construise un circuit NAND de même sémantique. En déduire que le système NAND est complet.

De même avec le système NOR.

5 Formules et circuits

Rappeler le type de données pour définir une formule à partir des constantes $\top$ (vrai), $\bot$ (faux) et des opérateurs $\neg$, $\wedge$, $\vee$ (on laisse de côté l'implication).

5.1 Du circuit à la formule

Écrire une fonction :

`formules_of_circuit : connecteur circuit -> formule vect`

qui à partir d'un circuit ET/OU/NON construise, pour chacune des portes de sortie du circuit, une formule représentant sa valeur de vérité en fonction des bits présents aux entrées du circuit (entrées qui seront les variables de la formule). On pourra construire le tableau des formules associées à chaque porte du circuit.

5.2 De la formule au circuit

Écrire une fonction :

`circuit_of_formule : formule -> connecteur circuit`

qui à partir d'une formule sur ces opérateurs, construise un circuit « de même sémantique », c'est-à-dire dont les entrées sont les variables de la formule, et dont la sortie calcule la valeur de vérité de la formule en fonction des valeurs des variables passées en entrée.

Une optimisation importante pourrait être faite à ce niveau : le partage des sous-expressions (c'est-à-dire, le fait de ne calculer les sous-formules qu'une seule fois), qui permet de réduire le nombre de portes du circuit et donc l'énergie consommée.