# Programmation
# Spécification et certification du logiciel

Notes de cours

P. Manoury

*2000-01*

## 1 Spécifications algébriques

Tout programmeur sait ce qu'est une *spécification* et tout programmeur a manipulé des spécifications. Celles-ci sont, en géneral, réduite à la simple expression de nom de fonctions accompagnées de leur type. Ces spécifiactions sont rassemblées dans des fichiers particuliers dits *d'interface*.

Par exemple, en faisant abstraction de certaines scories (ci-dessous le `__P`), on trouve, dans `stdio.h` les *déclarations* suivantes :

```
extern int getc __P ((FILE *));
extern int getchar __P ((void));
extern char* gets __P ((char*));
/* etc ... */
```

Ces trois déclarations régissent l'emploi des fonctions correpondantes et donnent une indication sur leur rôle. L'information contenue dans ces déclaration est une *information de type*. Par exemple, le fonction `getc` appliquée à un descripteur de fichier (argument de type `FILE *`) a pour valeur un entier (type `int`).

D'autres langages utilisent un langage de type différent qui s'apparente, extérieurement, aux formules du calcul propositionnel. Par exemple, on trouvera dans le fichier d'interface du module préchargé du langage O'CAML les déclarations :

```
val read_line : unit -> string
val read_int : unit -> int
val read_float : unit -> float
```

Ces trois déclarations aussi donnent trois fonctions de lecture sur l'entrée standard.

### Type de données abstrait

En programmation, les fichiers d'interface sont en général organisés de façon à rassembler les informations concernant un même sujet ou un même *objet*. Ces unités constituent des modules.

Toujours pour rester dans le monde ML, on trouve dans la distribution d'O'CAML les fichiers`list.mli`, `stack.mli`, `queue.mli` qui contiennent les primitives de créations et de manipulation de *structures de données* linéaires usuelles : les listes, les piles, les files d'attente.



Analysons quelques éléments du fichier `stack.mli` :

```
type 'a t

val create: unit -> 'a t

val push: 'a -> 'a t -> unit

val pop: 'a t -> 'a

val clear : 'a t -> unit

val length: 'a t -> int
```

La première ligne que nous avons reproduite introduit le nom de type `t` pour les piles (Vu de l'extérieur, le nom complet est `Stack.t`). Dans le monde de la spécification algébrique, on utilise le terme *sorte* plutôt que celui de type. Le `'a` de la déclaration indique que le type des piles est paramétré par le type indéterminé (appelé `'a`)de ses éléments. Cette première ligne permet d'énoncer les suivantes qui utiliseront le nom dit type introduit. Les quatre lignes suivantes donnent des fonctions privilégiées sur les piles permettant leur création ou leur destructuration. La dernière donne l'information de longueur. Ces quelques fonctions sont les seules dont nous disposions pour manipuler kes piles dans nos programmes. L'interface ne dit rien quand à la représentation des piles : le type `t` est *déclaré sans être défini*. L'interface `stack.mli` définit un *type abstrait*.

Néanmoins, avec ces seules informations, on peut écrire une *expression* ayant pour *valeur* une pile contenant les entiers 1, 2 3 et 4 : `push 1 (push 2 (push 3 (push 4 (create()))))`. On dira aussi que l'expression *dénote* une valeur.

L'interface telle que nous l'avons résumée ci-dessus ne dit rien non plas quand au comportement des opérateurs déclarés en dehors de leur type. Le savoir que nous en avons vient d'ailleurs. Ici, des *commentaires* qui accompagnent les déclarations :

```
(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t
        (* The type of stacks containing elements of type ['a]. *)

val create: unit -> 'a t
        (* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
val clear : 'a t -> unit
        (* Discard all elements from a stack. *)
val length: 'a t -> int
        (* Return the number of elements in a stack. *)
```

Ces commentaires sont partie intégrante de la *spécification* du module `Stack`. On les retrouvent dans la documentation qui accompagne la distribution du langage; le manuel de référence.

Ce que nous voulons faire est de donner un statut formel aux commentaires décrivant les opérateurs : remplacer l'expression en langue naturelle par une expression en langue formelle ou mathématique. Notre première approche sera celle des spécification algébrique où les opérateurs sont décrits par des *systèmes d'axiomes équationnels*.

## Spécifications équationnelles

Une spécification équationnelles est un ensemble d'identités de valeurs entre expressions. Par exemple, l'identité entre l'élément retourné par `pop` et le dernier empilé s'écrit comme l'équation :

```
(pop (push x s)) = x
```

Mais cet opérateur souffre d'un grave défaut pour se prêter facilement à la spécification algébrique : il fait deux choses à la fois ; il retourne l'élément en sommet de pile **et** le retire de la pile. La spécification algébrique se préte mal à l'expression des effets de bord, aussi considèrerons nous plutôt une *description purement fonctionnelle* des opérateurs. Chaque opération est unique et chaque symbole dénote une unique opération. Le symbole `top` désignera l'élément en sommet de pile et `pop` l'opération de retrait elle-même.

```
Sort: stack
Uses: int, elt
Symbols:
  create : → stack
  push : elt, stack → stack
  top : stack → elt
  pop : stack → stack
  clear : stack → stack
  length : stack → int
```

La figure ci-dessus décrit les premiers composants de la spécification des piles.

- la première ligne qui commence par le mot clé Sort donne le nom de la sorte (type abstrait) que l'on introduit.
- la seconde ligne qui commence par Uses donne la liste des autres sortes devant intervenir dans la spécification. Ici, les entiers (sorte `int`) et une sorte indéterminée `elt` qui joue le rôle de paramètre de sorte (le `'a` des types ML).
- les lignes suivantes, inroduites par Symbols donnent la listes des symboles composants la spécification avec leur sorte respective.

Ces éléments constituent l'en-tête de nos spécifications. Il faut maintenant en donner les corps : les équations devant être satisfaites.

```
Axioms: ∀ s:stack; x:elt
  (top (push x s)) = x
  (pop (push x s)) = s
  (length create) = 0
  (length (push x s)) = 1+(length s)
  (length (clear s)) = 0
```

Les variables servant à exprimer les axiomes sont déclarées avec leur sorte derrière le mot-clé Axioms.

Il convient de faire quelques remarques sur cet ensemble d'équations.

- les opérateurs `create` et `push` jouent un rôle particulier concernant les piles : celui de *constructeurs*. cela signifie que toute expression de sorte `stack` peut se ramener, par le jeu des èquations, à une expression utilisant uniquement `create` et `push`.
- la valeur des opérateurs `top` et `pop` n'est pas spécifiée lorsqu'on les applique à une pile vide (de longueur nulle).
- l'opérateur `length` est définit par des équations récursives.



### Des listes

Voici la structue linéaire classique des listes. Nous en donnons quelques opérateurs puis nous verrons comment déduire d'autres égalités à partir de celles posées en axiome.

```
Sort: list
Uses: elt, int
Symbols:
  nil : → list
  cons : elt, list → list
  length : list → int
  hd : list → elt
  tl : list → list
  append : list, list → list
  rev : list ->list
  rev_append : list, list ->list

Axioms: ∀ xs, ys:list; x:elt
  (length nil) = 0
  (length (cons x xs)) = 1+(length xs)
  (hd (cons x xs)) = x
  (tl (cons x xs)) = xs
  (append nil ys) = ys
  (append (cons x xs) ys) = (cons x (append xs ys))
  (rev nil) = nil
  (rev (cons x xs)) = (append (rev xs) (cons x nil))
  (rev_append nil ys) = ys
  (rev_append (cons x xs) ys) = (rev_append xs (cons x ys))
Theorems: ∀ xs, ys, zs:list
  1. (length (append xs ys)) = (length xs) + (length ys)
  2. (append (append xs ys) zs) = (append xs (append ys zs))
  3. (rev_append xs ys) = (append (rev xs) ys)
  4. (rev xs) = (rev_append xs nil)
```

L'égalité 4. est intéressante du point de vue implantation puisqu'elle donne un moyen d'obtenir le miroir d'une liste en utilisant une récursion terminale.

Le preuve de 1. est immédiate par induction sur la liste `xs` :

- si `xs = nil`, on a immédiatement `(length (append xs ys)) = (length ys)`;
- si `xs = (cons x xs')`, notre hypothèse de récurrence est que
  `(length (append xs' ys)) = (length xs') + (length ys)`.

  on a:

```
(length (append (cons x xs') ys))  =  1 + (length (append xs' ys))
                                   =  1 + (length xs') + (length ys)
                                   =  (length (cons x xs)) + (length ys)
```

La preuve de 2. qui donne l'associativité de l'opérateur de concaténation `append` est tout aussi immédiate par induction sur `xs` (en exercice).

La preuve de 3. est une induction un peu plus rusée : on montre par induction sur la liste *xs* que ∀ `ys:list. (rev_append xs ys) = (append (rev xs) ys)`. La présence de la quantification dans la formule à démontrer est primordiale car elle sera aussi présente dans l'hypothèse d'induction et en permettra l'utilisation.

- si `xs = nil`, l'égalité est immédiate;

– si xs = (cons x xs'), notre hypothèse de récurrence est que

∀ ys:list. (rev_append xs' ys) = (append (rev xs') ys)

En utilisant les axiomes équationnels et 2., on obtient à montrer :

(rev_append xs' (cons x ys)) = (append (rev xs) (cons x ys))

il suffit alors d'instancier le ys (quantifié) de l'hypothèse de récurrence par (cons x ys) pour obtenir l'égalité recherchée.

L'égalité 4. est une conséquence de 3.

## Équations conditionnelles

Une équation conditionnelles est une expression de la forme : c1, .., cn ⇒ e où les c1, .., cn sont des expressions booléennes et e une équation. Les expressions boolénnes sont appelées *préconditions* de l'équation e. On interprète ces sortes d'équation selon le sens usuel du connecteur propositionnel d'implication : si les conditions sont satisfaites (i.e. c1=true, .., cn=true) alors l'équation doit aussi l'être.

Les équations conditionnelles permettent des restreindre le domaine d'application de certaines équations. Par exemple, on peut spécificier un opérateur d'accès au ième éleément d'une liste en précisant le domaine de validité de l'indice :

```
Extends: list
Symbols:
  nth : list, int → elt
Axioms: ∀ xs:list; x:elt; i:int
  (nth (cons x xs) 0) = x
  (0 < i), (i ≤ (length xs)) ⇒ (nth xs i) = (nth (tl xs) (i-1))
```

Remarquons que nos préconditions ont pour conséquence que la liste xs n'est pas vide (puisque qu'elle est de longueur non nulle).

On a, au passage, introduit dans nos spécifications le nouveau mot clé Extends qui indique que l'on étend une sorte (ici, list).

Les équations conditionnelles servent également à exprimer des *alternatives* comme dans la spécification suivante de l'opérateur de test d'appartenance :

```
Extends: list
Symbols:
  mem : elt, list → bool
Axioms: ∀ xs:list; x,y:elt
  (mem x nil) = false
  (mem x (cons x xs)) = true
  (x≠y) ⇒ (mem y (cons x xs)) = (mem y xs)
```

### Listes ordonnées

Nous voulons exprimer ici la propriété d'ordonnancement des éléments d'une liste. Il faut pour cela disposer d'une relation d'ordre sur les éléments de la sorte indéterminée elt. C'est ce que nous faisons en introduisant l'usage des mots clés Assume: et with.

La propriété d'ordonnancement sera spécifiée par l'opérateur booléen sorted.

```
Extends: list
Assume:
  (<) : elt, elt → bool
  (≤) : elt, elt → bool
with: ∀ e,e1,e2,e3:elt
  (e < e) = true
  (e1 < e2) ⇒ (e2 < e1) = false
  (e1 < e2), (e2 < e3) ⇒ (e1 < e3) = true
  (e1 ≤ e2) = (not (e2 < e1))
Symbols:
  sorted : list → bool
Axioms: ∀ xs:list; x,y:elt
  (sorted nil) = true
  (sorted (cons x nil)) = true
  (x ≤ y) ⇒ (sorted (cons x (cons y xs))) = (sorted (cons y xs))
  (y < x) ⇒ (sorted (cons x (cons y xs))) = false
```



On peut alors utiliser le formalisme des spécifications algébriques pour décrire un algorithme de tri simple : le tri par insertion. On montrera ensuite sa correction.

```
Extends: list
Symbols:
  ins : elt, list → list
  ins_sort : list → list
Axioms: ∀ xs:list; x,y:elt
  (ins x nil) = (cons x nil)
  (x ≤ y) ⇒ (ins x (cons y xs)) = (cons x (cons y xs))
  (y < x) ⇒ (ins x (cons y xs)) = (cons y (ins x xs))
  (ins_sort nil) = nil
  (ins_sort (cons x xs)) = (ins (ins_sort xs))
Theorems: ∀ xs:list
  1. (sorted (ins_sort xs)) = true
```

Le théorème de correction de l'algorithme exprime que la valeur obtenue par l'opérateur ins_sort est une liste ordonnée. sa démonstration requiert un lemme auxiliaire qui exprime la propriété d'*invariance* de l'opérateur d'insertion ins pour la propriété d'ordonnancement. Ce lemme s'énonce :

```
Extends: list
Lemma: ∀ xs:list; x:elt
  2. (sorted xs) ⇒ (sorted (ins x xs)) = true
```

Pour démontrer ce lemme d'invariance, on utilise un autre lemme auxiliaire correspondant à l'un des cas d'induction de la démonstration de 2. :

```
Extends: list
Lemma: ∀ xs:list; x,y:elt
  2.1 (x ≤ y), (sorted (cons x xs)) ⇒ (sorted (cons x (ins y xs))) = true
```

Preuve de 2.1: on montre

∀ x,y:elt. (x ≤ y), (sorted (cons x xs)) ⇒ (sorted (cons x (ins y xs))) = true

par induction sur la liste xs.

– si xs = nil, le résultat est immédiat.
– si xs = (cons z zs), on a les hypothèses suivantes :
  HR: ∀ x,y:elt. (x ≤ y), (sorted (cons x zs)) ⇒ (sorted (cons x (ins y zs))) = true
  H1: (x ≤ y) = true
  H2: (sorted (cons x (cons z zs))) = true
  Notons que de H2, on tire
  H21: (x ≤ z) = true et
  H22: (sorted (cons z zs)) = true

Il faut montrer qu'alors: `(sorted (cons x (ins y (cons z zs)))) = true`.

Pour cela, on raisonne par cas sur la valeur de `(y ≤ z)` :

- si `(y ≤ z) = true`, il faut montrer que `(sorted (cons x (cons y (cons z zs)))) = true`. Les hypothèses nous donnent que `x ≤ y ≤ z` ce qui permet d'obtenir le résultat recherché par application de la définiton de `sorted`.
- sinon, `(z < y) = true` et il faut montrer que `(sorted (cons x (cons z (ins y zs)))) = true`.
  En utilisant H21, il reste à montrer que `(cons z (ins y zs))) = true`. Notons que, comme on est dans le cas où `(z < y) = true`, on a aussi que H3: `(z ≤ y) = true`. On utilise alors H3 et H22 tirer notre résultat de l'hypothèse de récurrence HR où `x` est instancié en `z`.

## 2 Assertions

Les spécifications algébriques permettent de spécifier des données a priori peu fonctionnelles telles que les tableaux.

En fait, on peu avoir une idée de ce que pourrait être la spécification d'une sorte `vect` en consultant l'interface du module `Array` d'Objective Caml :

```
Module Array: array operations

val length : 'a array -> int

      Return the length (number of elements) of the given array.

val get: 'a array -> int -> 'a

      Array.get a n returns the element number n of array a. The first
      element has number 0. The last element has number Array.length a
      - 1. Raise Invalid_argument "Array.get" if n is outside the
      range 0 to (Array.length a - 1). You can also write a.(n)
      instead of Array.get a n.

val set: 'a array -> int -> 'a -> unit

      Array.set a n x modifies array a in place, replacing element
      number n with x. Raise Invalid_argument "Array.set" if n is
      outside the range 0 to Array.length a - 1. You can also write
      a.(n) <- x instead of Array.set a n x.

val make: int -> 'a -> 'a array
val create: int -> 'a -> 'a array

      Array.make n x returns a fresh array of length n, initialized
      with x. All the elements of this new array are initially
      physically equal to x (in the sense of the ==
      predicate). Consequently, if x is mutable, it is shared among
      all elements of the array, and modifying x through one of the
      array entries will modify all other entries at the same time.
```

Il s'en dégage essentiellement une opération de création, une opération d'accès, une opération de modification et une information de longueur ou cardinalité. Traduisons cela dans notre formalisme :

```
Sort: vect
Uses: int, elt
Symbols:
  length : vect → int
  get : vect, int → elt
  set : vect, int, elt → vect
  make : int, elt → vect
Axioms: ∀ v:vect; e:elt; n,i,j:int
  (length (make n e)) = n
  (length (set v i e)) = (length v)
  (0 ≤ i), (i < (length v)) ⇒ (get (make n e) i) = e
  (0 ≤ i), (i < (length v)) ⇒ (get (set v i e) i) = e
  (0 ≤ i), (i < (length v)), (0 ≤ j), (j < (length v)), (i ≠ j)
    ⇒ (get (set v i e) j) = (get v j)
```

On a essentiellemnt abstrait la vision de tableaux en termes de ses constructeurs : `make` et `set`. On a restreint la pertinance de l'opération d'accès `set` par des équations conditionnelles. Un vecteur, du point de vue algébrique est l'histoire de sa création et de ses modifications.

Pour alléger l'écriture, on définit l'appartenance au domaine d'indice des vecteurs :

```
Extends: vect
Symbols:
  indom : int, vect → bool
Axioms: ∀ v:vect; i:int
  (indom i v) = (and (0 ≤ i) (i < (length v)))
```

#### Le plus petit indice de l'élément maximal d'un tableau

Pour finir donnons un dernier exemple de spécification d'algorithme : la recherche du plus petit indice de lélément maximal d'un tableau.

La spécification suivante définit complétement abstraitement ce qu'est ce plus petit indice (`imax`), puis donne une fonction de calcul dont on pourra montrer qu'elle calcule cette valeur (`find_imax`).

```
Extends: vect
Assume:
  (<) : elt, elt → bool
  (≤) : elt, elt → bool
with: ∀ e,e1,e2,e3:elt
  (e < e) = true
  (e1 < e2) ⇒ (e2 < e1) = false
  (e1 < e2), (e2 < e3) ⇒ (e1 < e3) = true
  (e1 ≤ e2) = (not (e2 < e1))
Symbols:
  imax : vect → int
  loop : int, int, vect → int
  find_imax : vect → int
Axioms: ∀ v:vect; i,m:int; e:elt
  /* IMAX.1 */
  (indom (imax v) v) = true
  /* IMAX.2 */
  (indom i v) ⇒ (get v i) ≤ (get v (imax v))
  /* IMAX.3 */
  (indom i v), ((get v (imax v)) = (get v i)) ⇒ (imax v) ≤ i
  (loop m (length v) v) = m
  (0 ≤ i), (i <(length v)), (indom m v), (get(v,m) <get(v,i))
    ⇒ (loop m i v) = (loop i i+1 v)
```

```
  (0 ≤ i), (i <(length v)), (indom m v), (get(v,i) ≤ get(v,m))
    ⇒ (loop m i v) = (loop m i+1 v)
  ((length v)≠ 0) ⇒ (find_max v) = (loop 0 1 v)
Theorems: ∀ v:vect
  ((length v)≠ 0) ⇒ (find_max v) = (imax v)
```

La démonstration du théorème de correction de `find_max` réclame, en fait, la correction de l'appel à `loop` :

```
Theorems: ∀ v:vect
  ((length v)≠ 0) ⇒ (loop 0 1 v) = (imax v)
```

Pour obtenir cette correction, il faut observer qu'en fait, ce n'est pas `loop` elle-même qui est correcte, mais son emploi à partir de bonnes valeurs. Par exemple (en utilisant la notation OCAML des tableau) `(loop 1 2 [|5;4;3;2;1;0|])` ne donne pas le résultat escompté. En revanche, si le premier argument contient la valeur du plus petit indice de l'élément maximal parmi les indices déjà explorés, on obtiendra bien le résultat escompté en poursuivant la recherche.

Pour exprimer, le lemme qui nous donnera notre théorème, nous avons besoin d'une variante renforcée de la spécification de `imax` : l'opérateur `bimax` qui donne le plus petit indice de l'élément maximal des éléments contenus dans le tableau avant un certain indice :

```
Extends: vect
Symbols:
  bimax : int → vect → int

Axioms: ∀ v:vect; i,j,m:int
  (0 < i) ⇒ (indom (bimax i v) v) = true
  (0 ≤ j), (j < i) ⇒ (get v j) ≤ (get v (bimax i v))
  (0 ≤ j), (j < i), ((get v (bimax i v)) = (get v j))
    ⇒ (bimax i v) ≤ j
Lemma: ∀ v:vect; i,m:int
  ((length v) ≠ 0), (0 ≤ i), (i ≤ (length v))
    ⇒ (loop (bimax i v) i v) = (imax v)
```

L'induction permettant de mener à bien cette preuve est la même que celle qui permet d'établir la terminaison de `loop` : la distance de l'indice `i` au dernier indice de `v` décroît à chaque appel récursif.

Soit `n=(length v)`, on prouve notre lemme par induction sur `n-i` :

- si `n-i`= 0, alors `i`=`n` et on a que `(loop (bimax n v) n v)`=`(bimax n v)`=`(imax v)`.
- sinon, posons comme hypothèse de récurrence: `(loop (bimax i+1 v) i+1 v) = (imax v)` (ce qui est légitime car `n-(i+1)`=`(n-i)-1`) et montrons `(loop (bimax i v) i v) = (imax v)`.
  On raisonne alors par cas selon que `(get v i) ≤ (get v (bimax i v))` ou non.
  - si `(get v i) ≤ get v bimax i v)` alors
    ```
    (loop (bimax i v) i v)  =  (loop (bimax i v) i+1 v)
                            =  (loop (bimax i+1 v) i+1 v)
                            =  (imax v)
    ```
  - si `get v (bimax i v)) < (get v i)` alors
    ```
    (loop (bimax i v) i v)  =  (loop i i+1 v)
                            =  (loop (bimax i+1 v) i+1 v)
                            =  (imax v)
    ```

## Assertions

Nous allons à présent utiliser la spécification ci-dessus pour établir la correction d'un programme impératif calculant l'indice de l'élément maximal.



On peut donner en Objective Caml une version impérative de notre fonction de recherche de l'indice de l'élément maximal :

```
let find_max v =
 let m = ref 0 in
 let i = ref 1 in
  while !i < Array.lenth v do
   if v.(!m) < v.(!i) then m := i;
   incr i
  done;
  !m
;;
```

Affirmer la correction de ce programme. c'est affirmer que la valeur retournée par `(find_max v)` est celle de `(imax v)`. On peut indiquer cette égalité sous forme d'un commentaire inséré dans le texte du programme :

```
let find_max v =
 let m = ref 0 in
 let i = ref 1 in
  while !i < Array.lenth v do
   if v.(!m) < v.(!i) then m := i;
   incr i
  done;
  !m (* !m = (imax v) *)
;;
```

De tels commentaires s'appellent des *assertions*. Un assertion affirme que la formule qu'elle contient est vraie *à l'endroit* du programme où elle est insérée. Cette spacialité des assertions vient du style impératif de programmation : un programme impératif est une suite d'instructions.

Intuitivement, ce programme est correct car la boucle `while` conserve la propriété d'invariance que `m` contient toujours l'indice de l'élément maximal parmi ceux explorés. En d'autres termes, on a, tout au long du programme `!m = (bimax !i v)`.

On peut alors préciser la raison de la correction en détaillant les assertions vérifiées à chaque étape du calcul :

```
let find_max v =
 let m = ref 0 in
 let i = ref 1 in
  (* a1 : !m = (bimax !i v) *)
  while !i < Array.length v do
   (* a2 : (!i < Array.length v) & (!m = (bimax !i v)) *)
   if v.(!m) < v.(!i) then m := i;
   (* a3 : !m = (bimax !i+1 v) *)
   incr i
   (* a4 : !m = (bimax !i v) *)
  done;
  (* a5 : (!i = Array.lenth v) & (!m = (bimax !i v)) *)
  !m
  (* a6 : !m = (imax v) *)
;;
```

Examinons brièvement chacune de nos assertions :

- `a6` la dernière assertion affirme la correction générale du programme. Ce devra être une conséquence des assertions précédentes;
- `a1` affirme qu'avant de rentrer dans la boucle `while`, la propriété d'invariance est établie;

a2 affirme qu'à chaque nouveau passage dans la boucle, on a l'invariance;

a3 affirme que cette propriété est devenue vrai pour l'indice suivant. Les assertions a2 et a3 tiennent lieu de schéma de récurrence où a2 est l'hypothèse de récurrence.

a4 permet de «retrouver» l'hypothèse de récurrence si l'on revient dans la boucle;

a5 donne la valeurs de l'indice de parcourt en sortie de boucle et la valeur obtenue pour m. C'est de ces deux valeur que l'on déduit la validité de a6.

## 3 Théorie axiomatique des programmes

On peut se convaincre intuitivement que la fonction `find_max` satisfait l'ensemble des assertions qui commentent son code. Mais pourquoi se contenter de si peu lorsque l'on peut *formaliser* la relation entre le *langage de programmation* et le *langage de spécification*?

Une formalisation de ce rapport est connu sur le nom de *logique de Hoare*. Les énoncés de cette logique sont des triplets de la forme $[P]\ C\ [Q]$ où $P$ et $Q$ sont des formules du langage de spécification et $C$ une expression du langage de programmation. La formule $P$ est appelée *précondition* et $Q$, *postcondition*. Intuitivement, les formules $P$ et $Q$ expriment les contraintes et propriétés que doivent satisfairent les variables du programme $C$. La précondition $P$ décrit *l'état* des variables *avant* exécution de $C$ et $Q$, l'état ou, le résultat, obtenu *aprés* l'exécution de $C$. La relation entre pré- et post- condition est définie par induction sur les règles de construction du langage de programmation.

Nous considérons un tout petit noyau des langages de programmation impératifs :

| | |
|---|---|
| Instruction vide | `SKIP` |
| Affectation | $x$ `:=` $e$ |
| Séquence | $C_1$ `;` $C_2$ |
| Conditionnelle | `If` $e$ `then` $C_1$ `else` $C_2$ |
| Boucle | `While` $e$ `do` $C$ |

On présente les définitions sous forme de règles de déduction :

$$\frac{}{[P]\ \texttt{SKIP}\ [P]}$$

$$\frac{}{[P[e/x]]\ x\ \texttt{:=}\ e\ [P]}$$

$$\frac{[P]\ C_1\ [Q] \qquad [Q]\ C_2\ [R]}{[P]\ C_1\ \texttt{;}\ C_2\ [R]}$$

$$\frac{[P \wedge e = \texttt{true}]\ C_1\ [Q] \qquad [P \wedge e = \texttt{false}]\ C_2\ [Q]}{[P]\ \texttt{If}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ [Q]}$$

$$\frac{[P \wedge e = \texttt{true}]\ C\ [P]}{[P]\ \texttt{While}\ e\ \texttt{do}\ C\ [P \wedge e = \texttt{false}]}$$

L'ensemble de ces règles décrivant le comportement des instruction du langage de programmation vis-à-vis des états exprimés par les formules est complété par une dernière règle autorisant des transformations purement logiques des pré- et post- conditions :

$$\frac{P \Rightarrow P' \qquad [P']\ C\ [Q'] \qquad Q' \Rightarrow Q}{[P]\ C\ [Q]}$$



En scindant cette règle en deux, on obtient les deux règles dérivées suivantes :

$$\frac{P \Rightarrow P' \qquad [P']\ C\ [Q]}{[P]\ C\ [Q]}\ \textit{Affaiblissement de la précondition}$$

$$\frac{[P]\ C\ [Q'] \qquad Q' \Rightarrow Q}{[P]\ C\ [Q]}\ \textit{Renforcement de la postcondition}$$

### `find_max` revisité

Reformulons dans le langage axiomatisé ci-dessus la fonction `find_max` :

```
m := 0;
i := 1;
While i < (length v) do begin
 If v(m) < v(i) then m := i else SKIP;
 i := i+1
end
```

Nous voulons montrer que si `v` est un tableau non vide alors, aprés exécution de `find_max`, la variable `m` contient la valeur de `(imax v)`. C'est ce qu'exprime le triplet :

$$[\texttt{(length v)} \neq 0]\ \texttt{find\_max}\ [\texttt{m} = \texttt{(imax v)}]$$

Le programme, simple, `find_max` est constitué d'une initialisation ($C_0$) suivie d'une boucle (`While` $e$ `do` $C$). Nous voulons établir $[P]\ C_0\texttt{;}\ \texttt{While}\ e\ \texttt{do}\ C\ [Q]$. À ce shéma de programme correspond le schéma de preuve suivant :

$$\frac{[P]\ C_0\ [I] \qquad \dfrac{I \wedge \neg e \Rightarrow Q \qquad \dfrac{[I \wedge e]\ C\ [I]}{[I]\ \texttt{While}\ e\ \texttt{do}\ C\ [I \wedge \neg e]}}{[I]\ \texttt{While}\ e\ \texttt{do}\ C\ [Q]}}{[P]\ C_0\texttt{;}\ \texttt{While}\ e\ \texttt{do}\ C\ [Q]}$$

Suivant ce schéma de preuve, pour établir la correction de notre programme, il nous faut donc trouver une formule $I$ (dit *invariant*) tel que :

1. $[P]\ C_0\ [I]$
2. $I \wedge \neg e \Rightarrow Q$
3. $[I \wedge e]\ C\ [I]$

#### Preuve de correction (partielle) de `find_max`

Nous avons déjà vu l'invariant nécessaire à notre preuve, il s'agit de l'égalité `m = (bimax i v)`. Passons donc aux trois étapes définies ci-dessus :

1. Il faut montrer que :
$$[\texttt{(length v)} \neq 0]\ \texttt{m:=0; i:=1}\ [\texttt{m = (bimax i v)}]$$
Il est facile de vérifier que $0 =$ `(bimax 1 v)` est vrai. On a donc trivialement l'implication :
$$\texttt{(length v)} \neq 0 \Rightarrow 0 = \texttt{(bimax 1 v)}$$
En utilisant la règle d'affaiblissement de la précondition, on obtient la séquence d'initialisation avec : $[0 = \texttt{(bimax 1 v)}]$ `m := 0` $[\texttt{m = (bimax 1 v)}]$ et $[\texttt{m = (bimax 1 v)}]$ `i := 1` $[\texttt{m = (bimax i v)}]$
CQFD

2. En utilisant les équations de `bimax`, on peut montrer que si `i` $\geq$ `(length v)` alors `(bimax i v)` = `(imax v)`. D'où

$$\texttt{m = (bimax i v)} \wedge \neg(\texttt{i < (length v)}) \Rightarrow \texttt{m = (imax v)}$$

3. Vient maintenant le gros morceau : montrer que l'invariant est conservé par le corps de la boucle. C'est à dire (en écrivant es triplets verticalement) :

```
[ m = (bimax i v) ∧ i < (length v) ]
If v(m) < v(i) then m := i else SKIP;i := i+1
[ m = (bimax i v) ]
```

Ou encore, par les règles de la séquence et de l'affectation :

```
[ m = (bimax i v) ∧ i < (length v) ]
If v(m) < v(i) then m := i else SKIP;
[ m = (bimax i+1 v) ]
i := i+1
[ m = (bimax i v) ]
```

La seconde partie de la séquence est immédiate, reste à voir :

```
[ m = (bimax i v) ∧ i < (length v) ]
If v(m) < v(i) then m := i else SKIP;
[ m = (bimax i+1 v) ]
```

Par la règle de l'alternative, il faut montrer que (a)

```
[ m = (bimax i v) ∧ i < (length v) ∧ v(m) < v(i)]
m := i
[ m = (bimax i+1 v) ]
```

et que (b)

```
[ m = (bimax i v) ∧ i < (length v) ∧ v(m) ≥ v(i)]
SKIP;
[ m = (bimax i+1 v) ]
```

(a) de `m = (bimax i v)` et `v(m) < v(i)`, on peut déduire que `i = (bimax i+1 v)`. Et, par règle de l'affectation, on a

$$[\texttt{i = (bimax i+1 v)}]\ \texttt{m := i}\ [\texttt{m = (bimax i+1 v)}]$$

On obtient le résulat attendu par affaiblissement de la précondition.

(b) de `m = (bimax i v)` et `v(m)` $\geq$ `v(i)`, on peut déduire que `m = (bimax i+1 v)`. On a alors le résultat attendu par la règle du `SKIP` et affaiblissement de la précondition.

CQFD

# 4 Raffinement

L'axiomatique de Hoare-Floyd permet de vérifier *a posteriori* qu'un programme donné statisfait une spécification exprimée en terme de pré et post conditions. L'idée du raffinement est d'obtenir la correction des programmes *a priori* en régentant leur construction à partir de leur spécification. La dérivation des programmes obéit à un certain nombre de règles qui permettent d'introduire petit-à-petit les constructions usuelles des langages de programmation. On élimine ainsi progressivement toutes les formulations non calculatoires des spécifications pour obtenir un code exécutable. Chaque règle est conçue de façon à garantir la correction au sens de Hoare-Floyd (nous verrons comment ci-dessous).

La discipline du raffinement conçoit en fait les programmes comme un classe particulière de spécification. On obtient ainsi un langage mélangeant des spécifications sous forme de pré et post conditions (notées $[P, Q]$) et des constructions des langages de programmations. Ainsi,on peut obtenir des expressions comme :

$$\texttt{If}\ B\ \texttt{then}\ [P_1, Q_1]\ \texttt{else}\ [P_2, Q_2]$$



On voit donc les structures de controle des langages de programmations comme des *combinateurs* de spécifications. De façon générale, on définit l'ensemble des spécifications :

- si $P$ et $Q$ sont des formules alors $[P, Q]$ est une spécification.
- si $S_1$ et $S_2$ sont des spécifications alors $S_1\ ;\ S_2$ est une spécification.
- si $B$ est une formule et si $S_1$ et $S_2$ sont des spécifications alors $\texttt{If}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ aussi.
- si $B$ est une formule et $S$ une spécification alors $\texttt{While}\ B\ \texttt{do}\ S$ est une spécification.
- si $S$ est une spécification et $x$ est une variable alors $\texttt{Var}\ x.S$ est une spécification.
- si $x$ est une variable et $e$ une expression alors $x\ \texttt{:=}\ e$ est une spécification.
- `SKIP` est une spécification.

La relation de raffinement etre deux spécification $S_1$ est $S_2$ est notée :

$$S_1 \supseteq S_2$$

Cette relation devra-t-être monotone par rapport aux combinateurs : si $S_1 \supseteq S_1'$ et $S_2 \supseteq S_2'$ alors

- $S_1\ ;\ S_2 \supseteq S_1'\ ;\ S_2'$ ;
- $\texttt{If}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \supseteq \texttt{If}\ B\ \texttt{then}\ S_1'\ \texttt{else}\ S_2'$ ;
- $\texttt{While}\ B\ \texttt{do}\ S_1 \supseteq \texttt{While}\ B\ \texttt{do}\ S_1'$ ;
- $\texttt{Var}\ x.S_1 \supseteq \texttt{Var}\ x.S_1'$.

La monotonicité permet un raffinememnt modulaire des spécifications.

### Règles de raffinement

**Ne rien faire** ($Sk$)

$$[P, P] \supseteq \texttt{SKIP}$$

**Affectation** ($As$)

$$[Q[e/x], Q] \supseteq x\ \texttt{:=}\ e$$

**Séquence** ($Sq$)

$$[P, Q] \supseteq [P, R]\ ;\ [R, Q]$$

**Variable locale** ($Vr$)

$$[P, Q] \supseteq \{\texttt{Var}\ x_1..x_n\,.\,[P, Q]\}$$

où $x_1..x_n$ n'apparaissent ni dans $P$ ni dans $Q$.

Combinant ces trois dernières règles, on peut dériver le programme d'échange de deux valeurs comme suit :

Soient $N$ et $M$ deux valeurs entières et $x$, $y$ deux variables

$$
\begin{array}{lll}
[x = N \wedge y = M,\ x = M \wedge y = N] & \supseteq & \{\ \texttt{Var}\ z.[x = N \wedge y = M,\ y = N \wedge x = M]\ \} \\
 & \supseteq & \{\ \texttt{Var}\ z. \\
 & & \quad [x = N \wedge y = M,\ z = N \wedge y = M]\ ; \\
 & & \quad [z = N \wedge y = M,\ y = N \wedge x = M]\ \} \\
 & \supseteq & \{\ \texttt{Var}\ z. \\
 & & \quad z\ \texttt{:=}\ x\ ; \\
 & & \quad [z = N \wedge y = M,\ y = N \wedge x = M]\ \} \\
 & \supseteq & \{\ \texttt{Var}\ z. \\
 & & \quad z\ \texttt{:=}\ x\ ; \\
 & & \quad [z = N \wedge y = M,\ z = N \wedge x = M]\ ; \\
 & & \quad [z = N \wedge x = M,\ y = N \wedge x = M]\ \} \\
 & \supseteq & \{\ \texttt{Var}\ z. \\
 & & \quad z\ \texttt{:=}\ x\ ; \\
 & & \quad x\ \texttt{:=}\ y\ ; \\
 & & \quad [z = N \wedge x = M,\ y = N \wedge x = M]\ \} \\
 & \supseteq & \{\ \texttt{Var}\ z. \\
 & & \quad z\ \texttt{:=}\ x\ ; \\
 & & \quad x\ \texttt{:=}\ y\ ; \\
 & & \quad y\ \texttt{:=}\ z\ \}
\end{array}
$$

Les deux règles suivantes permettent du raffinememnt purement logique, sans contrepartie algorithmique.

**Précondition** $(Wk)$

$$[P, Q] \supseteq [R, Q] \text{ si } P \Rightarrow R$$

**Postcondition** $(St)$

$$[P, Q] \supseteq [P, R] \text{ si } R \Rightarrow Q$$

L'application des règles d'affaiblissemnt de la précondition et de renforcement de la postcondition est soumise à une condition : vérifier la validité d'une formule. On appelle de telles conditions des *obligations de preuve*.

**Affectation bis** $(Ab)$ Losrque la précondition implique la postcondition modulo une substitution, on obtient, par affaiblissement une deuxième règle pour l'affectation :

$$[P, Q] \supseteq x\ \texttt{:=}\ e \text{ si } P \Rightarrow Q[e/x]$$

En effet

$$
\begin{array}{llll}
[P, Q] & \supseteq & [Q[e/x], Q] & (Wk) \\
 & \supseteq & x\ \texttt{:=}\ e & (As)
\end{array}
$$

**Initialisation** $(Ai)$ En utilisant $(Wk)$, $(As)$ et $(Sq)$ on obtient une autre règle dérivée $(Ai)$ permettant l'initialisation d'une variable :

$$[P, Q] \supseteq x\ \texttt{:=}\ e\ ;\ [P \wedge x = e,\ Q]$$

En effet :

$$
\begin{array}{llll}
[P, Q] & \supseteq & [P \wedge e = e,\ Q] & (Wk) \\
 & \supseteq & [P \wedge e = e,\ P \wedge x = e]\ ; & (Sq) \\
 & & [P \wedge x = e,\ Q] & \\
 & \supseteq & x\ \texttt{:=}\ e\ ; & (As) \\
 & & [P \wedge x = e,\ Q] &
\end{array}
$$



On complète notre jeu de règle par les structures de contrôles de base.

**Conditionnelle** $(If)$

$$[P, Q] \supseteq \texttt{If}\ B\ \texttt{then}\ [P \wedge B, Q]\ \texttt{else}\ [P \wedge \neg B, Q]$$

**Boucle** $(Wh)$

$$[I, I \wedge \neg B] \supseteq \texttt{While}\ B\ \texttt{do}\ [P \wedge B \wedge e = n, I \wedge e < n] \text{ si } P \wedge B \Rightarrow e \geq 0$$

Dans cette dernière règle, $n$ est une valeur entière et $e$ une expression qui soint là pour garantir la terminaison de la boucle.

## Exemple

Voici comment on peut dériver une boucle calculant la division euclideinne de deux entiers. On sait que si $X$ et $Y$ sont deux entiers leur quotient est $Q$ et leur reste $R$ si l'on a : $X = (Y \times Q) + R$. Il faut de surcroît que $R \leq Y$ et, comme précondition, que $Y > 0$.

$$
\begin{array}{llll}
[Y > 0,\ X = (Y \times Q) + R \wedge R \leq Y] & \supseteq & R\ \texttt{:=}\ X\ ; & (Ai) \\
 & & [Y > 0 \wedge R = X,\ X = (Y \times Q) + R \wedge R \leq Y] & \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & (Ai) \\
 & & [Y > 0 \wedge R = X \wedge Q = 0,\ X = (Y \times Q) + R \wedge R \leq Y] & \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & & [Y > 0 \wedge X = (Y \times Q) + R,\ X = (Y \times Q) + R \wedge R \leq Y] & (Wk) \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & & [Y > 0 \wedge X = (Y \times Q) + R,\ X = (Y \times Q) + R \wedge \neg(Y \leq R)] & (St) \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & & \texttt{While}\ (Y \leq R)\ \texttt{do} & (Wh) \\
 & & \quad [Y > 0 \wedge X = (Y \times Q) + R \wedge Y \leq R \wedge R = n, & \\
 & & \quad\ Y > 0 \wedge X = (Y \times Q) + R \wedge R < n] & \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & & \texttt{While}\ (Y \leq R)\ \texttt{do} & \\
 & & \quad [Y > 0 \wedge X = (Y \times (Q+1) + (R - Y) \wedge Y \leq R \wedge R = n, & (Wk) \\
 & & \quad\ Y > 0 \wedge X = (Y \times Q) + R \wedge R < n] & \\
 & \supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & & \texttt{While}\ (Y \leq R)\ \texttt{do} & \\
 & & \quad [Y > 0 \wedge X = (Y \times (Q+1) + (R - Y) \wedge Y \leq R \wedge R = n, & \\
 & & \quad\ Y > 0 \wedge X = (Y \times Q) + (R - Y) \wedge Y \leq R \wedge R = n] & (Sq) \\
 & & \quad [Y > 0 \wedge X = (Y \times Q) + (R - Y) \wedge Y \leq R \wedge R = n, & \\
 & & \quad\ Y > 0 \wedge X = (Y \times Q) + R \wedge R < n] &
\end{array}
$$

$$
\begin{array}{lll}
\supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & \texttt{While}\ (Y \leq R)\ \texttt{do} & \\
 & \quad Q\ \texttt{:=}\ Q + 1\ ; & (As) \\
 & \quad [Y > 0 \wedge X = (Y \times Q) + (R - Y) \wedge Y \leq R \wedge R = n, & \\
 & \quad\ Y > 0 \wedge X = (Y \times Q) + R \wedge R < n] & \\
\supseteq & R\ \texttt{:=}\ X\ ;\ Q\ \texttt{:=}\ 0\ ; & \\
 & \texttt{While}\ (Y \leq R)\ \texttt{do} & \\
 & \quad Q\ \texttt{:=}\ Q + 1\ ;\ R = R - Y & (Ab)
\end{array}
$$

Nous donnons ci-dessous la liste des obligations de preuves (résumées) dans l'ordre de leur apparition au cours de la dérivation :

1. $X = R \wedge Q = 0 \Rightarrow X = (Y \times Q) + R$

2. $R \leq Y \Rightarrow \neg(Y \leq R)$
3. $X = (Y \times Q) + R \Rightarrow X = (Y \times (Q+1)) + (R - Y)$
4. $Y > 0 \wedge X = (Y \times Q) + (R - Y) \wedge R = n \Rightarrow X = (Y \times Q) + (R - Y) \wedge R - Y < n$

## 4.1 Hoare-Floyd et le raffinement

On peut justifier la correction des règles basiques de raffinement en terme de logique de Hoare-Floyd. Pour cela, on interprète une spécification comme l'ensemble des programmes qui la satisfont, au sens de Hoare-Floyd. Pour l'écrire, on pose :

$$[P,Q] = \{\ C \mid\ \vdash [P]\ C\ [Q]\ \}$$

où $\vdash [P]\ C\ [Q]$ signifie que le triplet $[P]\ C\ [Q]$ est dérivable en logique de Hoare.

L'interprétation passe aux combinateurs de la façon suivante :

– $S_1\ \texttt{;}\ S_2 = \{\ C_1\ \texttt{;}\ C_2 \mid C_1 \in S_1 \wedge C_2 \in S_2\ \}$
– $\texttt{If}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 = \{\ \texttt{If}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid C_1 \in S_1 \wedge C_2 \in S_2\ \}$
– $\texttt{While}\ B\ \texttt{do}\ S = \{\ \texttt{While}\ B\ \texttt{do}\ C \mid C \in S\ \}$
– $\texttt{Var}\ x.S = \{\ \texttt{Var}\ x.C \mid C \in S\ \}$
– $x\ \texttt{:=}\ e = \{\ x\ \texttt{:=}\ e\ \}$
– $\texttt{SKIP} = \{\ \texttt{SKIP}\ \}$

On montre alors que si $S$ est un raffinement de $[P,Q]$ alors pour tout programme $C$ appartenant à (l'interprétation) de $S$, le triplet $[P]\ C\ [Q]$ est dérivable en logique de Hoare. C'est à dire :

$$[P,Q] \supseteq S \Rightarrow \forall C \in S. \vdash [P]\ C\ [Q]$$

On montre l'implication en raisonnant par cas sur la règle de raffinememnt appliquée :

**Ne rien faire** on a directement $[P]\ \texttt{SKIP}\ [P]$.
**Affectation** on a aussi $[Q[e/x]]\ x\ \texttt{:=}\ e\ [Q]$.
**Séquence** soit $C_1\ \texttt{;}\ C_2 \in [P,R]\ \texttt{;}\ [R,Q]$, on a par définition, $C_1 \in [P,R]$ et $C_2 \in [R,Q]$. D'où, $\vdash [P]\ C_1\ [R]$ et $\vdash [R]\ C_2\ [Q]$, et donc $\vdash [P]\ C_1\ \texttt{;}\ C_2\ [Q]$.
**Variable** soit $\texttt{Var}\ x.C \in \texttt{Var}\ x.[P,Q]$, on a, par définition $C \in [P,Q]$, et donc $\vdash [P]\ C\ [Q]$, par défintion. On a donc $\vdash [P]\ \texttt{Var}\ x.C\ [Q]$, puisque $x$ ne n'apparaîît pas dans $P$ ni dans $Q$.
**Conditionnelle** soit $\texttt{If}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \in \texttt{If}\ B\ \texttt{then}\ [P \wedge B, Q]\ \texttt{else}\ [P \wedge \neg B, Q]$. On a, par définition que $C_1 \in [P \wedge B, Q]$ et $C_2 \in [P \wedge \neg B, Q]$. D'où $\vdash [P \wedge B]\ C_1\ [Q]$ et $\vdash [P \wedge \neg B]\ C_2\ [Q]$. Et donc $\vdash [P]\ \texttt{If}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ [Q]$.
**Boucle** soit $\texttt{While}\ B\ \texttt{do}\ C \in \texttt{While}\ B\ \texttt{do}\ [P \wedge B \wedge n = e, P \wedge e < n]$. On a $C \in [P \wedge B \wedge n = e, P \wedge e < n]$, c'est-à-dire $\vdash [P \wedge B \wedge n = e]\ C\ [P \wedge e < n]$. D'où $\vdash [P]\ \texttt{While}\ B\ \texttt{do}\ C\ [P \wedge \neg B]$.

# 5 Spécification ensembliste

Une alternative à la formalisation en termes de spécification équationnelle est l'utilisation de la richesse et de la généralité de la *théorie des ensembles*.

Il existe des présentations axiomatiques rigoureuse de la théorie des ensembles. Néanmoins, nous pourrons nous contenter ici d'une présentation intuitive.

### Langage ensembliste

La relation de base entre ensembles est la *relation d'appartenance* notée

$$x \in y$$



pour « *$x$ appartient à $y$* » ou « *$x$ est une élément de $y$* ».

Sur la base de cette relation primitive, on exprime l'égalité entre ensembles en posant *l'axiome*, dit *d'extensionnalité*, suivant :

$$x = y \Leftrightarrow (\forall z. z \in x \Leftrightarrow z \in y)$$

Et d'autres termes, deux ensemble sont égaux si et seulement si ils possèdent exactement les mêmes éléments.

D'autres notions ensemblistes utiles, comme *l'inclusion* peuvent être définies au sens usuel :

$$x \subseteq y \mathrel{\widehat{=}} (\forall z. z \in x \Rightarrow z \in y)$$

Cette façon usuelle de définir s'oppose à la définition axiomatique de l'égalité en ce sens que la formule $x \subseteq y$ *syntaxiquement* équivalente à sa définition et peut être remplacée à la manière dont un préprocesseur remplace une macro, alors que l'égalité $x = y$ est donnée comme *logiquement* équivalente à une formule. On ne peut remplacer une égalité, mais on peut raisonner dessus en utilisant la formule équivalente.

Nous rappelons ci-dessous quelques constructions ensemblistes utiles en donnant, pour chacune, un axiome qui les caractérise :

| | notation | axiome |
|---|---|---|
| Ensemble vide : | $\emptyset$ | $\forall x. x \notin \emptyset$ |
| Couples | $(x,y)$ | $(x,y) = (x',y') \Leftrightarrow x = x' \wedge y = y'$ |
| Produit | $X \times Y$ | $z \in X \times Y \Leftrightarrow \exists x \in X. \exists y \in Y. z = (x,y)$ |
| Union | $X \cup Y$ | $z \in X \cup Y \Leftrightarrow z \in X \vee z \in Y$ |
| Ensemble des parties | $\mathcal{P}(X)$ | $z \in \mathcal{P}(X) \Leftrightarrow z \subseteq X$ |
| Schéma de compréhension | $\{x \in X \mid \varphi\}$ | $y \in \{x \in X \mid \varphi\} \Leftrightarrow y \in X \wedge \varphi[y/x]$ |

Enfin, d'autres notions sont définies en utilisant le schéma de compréhension :

| | notation | définition |
|---|---|---|
| Intersection | $X \cap Y$ | $\{z \in X \mid z \in Y\}$ |
| Différence | $X \backslash Y$ | $\{x \in X \mid z \notin Y\}$ |

**Relations binaires et fonctions**

Une relation binaire entre un ensemble $X$ et un ensemble $Y$ associe des éléments de $X$ avec des élément de $Y$. Cette association peut être représentée comme un couple $(x,y)$ où $x \in X$ et $y \in Y$. On peut alors définir *l'ensemble des relations binaires* entre deux ensembles :

$$X \leftrightarrow Y \mathrel{\widehat{=}} \mathcal{P}(X \times Y)$$

On peu ainsi parler d'une relation $R$ entre (éléments de) $X$ et $Y$ comme un élément de $X \leftrightarrow Y$ et noter $R \in X \leftrightarrow Y$. Si deux éléments $x$ et $y$ sont dans la relation $R$, on se donne la notation infixe usuelle en posant :

$$x\ \underline{R}\ y \mathrel{\widehat{=}} (x,y) \in R$$

Rappelons ce que sont les *domaine* et *codomaine* d'une relation :

$$\mathrm{dom}(R) \mathrel{\widehat{=}} \{x \in X \mid \exists y \in Y . x \mathrel{\underline{R}} y\}$$
$$\mathrm{ran}(R) \mathrel{\widehat{=}} \{y \in Y \mid \exists x \in X . x \mathrel{\underline{R}} y\}$$

Les fonctions sont une catégorie particulière de relations binaires : celles qui associe au plus un élément du codomaine au élément de leur domaine. En tant que relations, on pourra aussi parler de *l'ensemble des fonctions* entre $X$ et $Y$ en distinguant les fonctions *partielles* (notées $X \nrightarrow Y$) des fonction *totales* (notées $X \rightarrow Y$). Voici comment on définit ces deux ensembles :

$$\begin{array}{lcl} X \nrightarrow Y & \widehat{=} & \{f \in X \leftrightarrow Y \mid \forall x \in \mathrm{dom}(f) . \exists ! y \in Y . x \mathrel{\underline{f}} y\} \\ X \rightarrow Y & \widehat{=} & \{f \in X \nrightarrow Y \mid \mathrm{dom}(f) = X\} \end{array}$$

Si $f$ est une fonction, on se donne la notation usuelle d'application en posant l'axiome :

$$f(x) = y \Leftrightarrow (x, y) \in f$$

Ce ne peut être une simple définition, car l'expression $f(x) = y$ suppose l'existence de $y$.

On définit, sur les relations biniare un certain nombre d'opérateurs utiles dans le cadre de la spécification.

- restriction du domaine :
$$Z \lhd R \mathrel{\widehat{=}} \{(x, y) \in X \times Y \mid x \in Z \wedge x \mathrel{\underline{R}} y\}$$
- exclusion du domaine :
$$Z \mathrel{⩤} R \mathrel{\widehat{=}} (X \backslash Z) \lhd R$$
- restriction du codomaine :
$$R \rhd Z \mathrel{\widehat{=}} \{(x, y) \in X \times Y \mid y \in Z \wedge x \mathrel{\underline{R}} y\}$$
- mise-à-jour (de la relation $R_1$ par la relation $R_2$) :
$$R_1 \oplus R_2 \mathrel{\widehat{=}} (R_2 \mathrel{⩤} R1) \cup R_2$$

Cette dernière opération est massivement utilisée lorsque l'on veut modifier en un point la valeur d'une fonction. En notant $x \mapsto y$ le couple $(x, y)$, on a :

$$\left\{ \begin{array}{lcll} f \oplus \{x \mapsto y\}(x) & = & y & \\ f \oplus \{x \mapsto y\}(z) & = & f(z) & \text{si } z \neq x \end{array} \right.$$

### Arithmétique

On se donne l'arithmétique (c'est possible).

### 5.0.1 Les suites

Avec notre langage ensembliste, on peut définir une *structure linéaire générique*, modèle mathématique de structures de données que sont les listes, les tableaux, les files d'attentes, etc.

Une *suite* d'éléments de $X$, notée seq $X$, est une fonction partielle d'un intervalle $1..n$ dans $x$ :

$$\mathrm{seq}\ X \mathrel{\widehat{=}} \{S : I\!N_1 \nrightarrow X \mid \exists n : I\!N . \mathrm{dom} S = 1..n\}$$

Le ième élément de $S$ est simplement $S(i)$ (*i.e* l'image de $i$ par $S$). Notez que si $i \notin \mathrm{dom} S$ cet élément n'est pas défini.

On note :

| | |
|---|---|
| $\#S$ | la longueur (*i.e.* le nombre d'éléments) de la suite $S$ ; |
| $\langle\rangle$ | la suite vide (*i.e.* l'ensemble vide) ; |
| $\langle a_1, .., a_n \rangle$ | la suite composées des éléments $a_1, .., a_n$ (*i.e.* la fonction $\{1 \mapsto a_1, .., n \mapsto a_n\}$). |



Il sera souvent utile, on définit l'ensemble des suites non vides par :

$$\mathrm{seq}_1\ X \mathrel{\widehat{=}} \{s : \mathrm{seq}\ X \mid s \neq \langle\rangle\}$$

## Le langage Z

Le langage Z se base sur le langage ensembliste pour offrir un *format* de spécification de fonctions ou d'opérations.

### Défintions axiomatiques

Les fonctions sont définies de façon axiomatiques : on introduit un symbole et on donne une formule énonçant les propriétés essentielles de ce symbole. Le format de définition axiomatique est :

$$\begin{array}{|l} \mathit{nom} : \mathit{type} \\ \hline \mathit{formule} \end{array}$$

Voici quelques exemples de fonctions manipulant des suites.

Concaténation :

$$\begin{array}{|l} \frown : \mathrm{seq}\ X \times \mathrm{seq}\ X \rightarrow \mathrm{seq}\ X \\ \hline \forall s_1, s_2 \in \mathrm{seq}\ X . \forall i \in I\!N_1 . \\ \quad (i \leq \#s_1 \Rightarrow s_1 \frown s_2(i) = s_1(i)) \wedge \\ \quad (i > \#s_1 \Rightarrow s_1 \frown s_2(i) = s_2(i - \#s_1)) \end{array}$$

Sous suite :

$$\begin{array}{|l} \mathit{sub} : \mathrm{seq}\ X \times I\!N \times I\!N \rightarrow \mathrm{seq}\ X \\ \hline \forall s \in \mathrm{seq}\ X . \forall i, j \in I\!N . \forall k \in \mathrm{dom} S \cap i..j . \\ \quad \mathit{sub}(s, i, j)(k) = s(k) \end{array}$$

Suites vues comme des listes :

$$\begin{array}{|l} \mathit{head} : \mathrm{seq}_1\ X \rightarrow X \\ \hline \forall s \in \mathrm{seq}_1\ X . \\ \quad \mathit{head}(s) = s(1) \end{array} \qquad \begin{array}{|l} \mathit{tail} : \mathrm{seq}_1\ X \rightarrow \mathrm{seq}\ X \\ \hline \forall s \in \mathrm{seq}_1\ X . \\ \quad \mathit{tail}(s) = \mathit{sub}(s, 2, \#s) \end{array}$$

Suites vues comme des files d'attentes :

$$\begin{array}{|l} \mathit{last} : \mathrm{seq}_1\ X \rightarrow X \\ \hline \forall s \in \mathrm{seq}_1\ X . \\ \quad \mathit{last}(s) = s(\#s) \end{array} \qquad \begin{array}{|l} \mathit{front} : \mathrm{seq}_1\ X \rightarrow \mathrm{seq}\ X \\ \hline \forall s \in \mathrm{seq}_1\ X . \\ \quad \mathit{front}(s) = \mathit{sub}(s, 1, \#s - 1) \end{array}$$

### Schémas d'opérations

Un schéma d'opération est, à la base un triplet constitué d'un nom, d'un ensemble de déclarations et d'une formule. Le format général d'un schéma d'opération est le suivant :

$$\begin{array}{|l} \mathit{Nom} \\ \hline \mathit{déclarations} \\ \hline \mathit{formule} \\ \hline \end{array}$$

Les déclarations sont de la forme $x : X$ où $x$ est une variable et $X$ une expression ensembliste.

Ainsi considéré, un schéma représente un *état* : n-uplet de valeurs (les variables déclarées) possédant certaines propriétés (la formule). Cette formule peut exprimer des relations entre les variables constituant l'état. C'est pourquoi un schéma d'opération s'utilise aussi pour établir une relation entre un état *avant* (l'opération) et un état *aprés* (l'opération).

Par exemple, on définira l'opération de retournement d'une liste comme une opération $Rev$ reliant une suite $s$ à une suite $s'$ oú $s'$ est le miroir de $s$ :

$$\begin{array}{|l}
\multicolumn{1}{l}{Rev} \\ \hline
s, s' : \text{seq } X \\ \hline
\mathrm{dom}(s) = \mathrm{dom}(s') \wedge \\
\forall i \in \mathrm{dom}(s'), s'(i) = s(\#s - i + 1) \\ \hline
\end{array}$$

Cette façon de noter un état *avant* et *aprés* avec une apostrophe est devenue l'usage aussi l'a-t-on généralisée aux schemas.

Si le schéma $S$ est
$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
x : T \\ \hline
\varphi \\ \hline
\end{array}$$
alors le schéma $S'$ est
$$\begin{array}{|l}
\multicolumn{1}{l}{S'} \\ \hline
x' : T \\ \hline
\varphi[x'/x] \\ \hline
\end{array}$$

Un schéma peut se résumer à de simples déclarations auquel cas on laisse vide la section $formule$ :

$$\begin{array}{|l}
\multicolumn{1}{l}{Nom} \\ \hline
\textit{déclarations} \\ \hline
\\ \hline
\end{array}$$

Les schémas peuvent se combiner de plusieurs manières :

– par inclusion : si $S_1$ est
$$\begin{array}{|l}
\multicolumn{1}{l}{S_1} \\ \hline
x_1 : T_1 \\ \hline
\varphi_1 \\ \hline
\end{array}$$
on a
$$\begin{array}{|l}
\multicolumn{1}{l}{S_2} \\ \hline
S_1 \\
x_2 : T_2 \\ \hline
\varphi_2 \\ \hline
\end{array}
\;\widehat{=}\;
\begin{array}{|l}
\multicolumn{1}{l}{S_2} \\ \hline
x_1 : T_1 \\
x_2 : T_2 \\ \hline
\varphi_1 \wedge \varphi_2 \\ \hline
\end{array}$$

– par conjonction : si $S_2$ est
$$\begin{array}{|l}
\multicolumn{1}{l}{S_2} \\ \hline
x_2 : T_2 \\ \hline
\varphi_2 \\ \hline
\end{array}$$
on a
$$\begin{array}{|l}
\multicolumn{1}{l}{S_1 \wedge S_2} \\ \hline
x_1 : T_1 \\
x_2 : T_2 \\ \hline
\varphi_1 \wedge \varphi_2 \\ \hline
\end{array}$$



– etc.

Et utilisant la combinaison par conjonction, on définit une schéma général de relation *avant-aprés* :

$$\Delta S \;\widehat{=}\; S \wedge S'$$

### Un exemple

Les éléments de spécification qui suivent concernent la partie l'API de CICS qui permet de gérer les files d'attentes temporaires *Temporary Storage Queue* lors d'échanges de données entre divers sites d'un système d'information. L'ensemble de la spécification est donnée par schémas. Cet exemple est tiré de « Specifying the IBM CICS Application Programming Interface », par Steve King, *in* Specification Case Studies, Ian Hayes ed., Prentice Hall, deuxième édition, 1993.

**Les données** Nous manipulerons des files d'attentes contenant des suites d'octets. On pose :

$$\begin{array}{rcl}
BYTE & == & 0..255 \\
TSElem & == & \text{seq } BYTE
\end{array}$$

Les files d'attentes du système ($TSQ$) sont modélisées par des suites ($ar$) et un pointeur (l'entier $p$) sur le dernier élément de la file ayant fait l'objet d'une opération de lecture ou d'écriture (voir *infra*) :

$$\begin{array}{|l}
\multicolumn{1}{l}{TSQ} \\ \hline
ar : \text{seq } TSElem \\
p : IN \\ \hline
p \leq \#ar \\ \hline
\end{array}$$

On définit l'état initial d'une file d'attente :

$$\begin{array}{|l}
\multicolumn{1}{l}{TSQ_Initial} \\ \hline
TSQ \\ \hline
ar = \langle\rangle \wedge p = 0 \\ \hline
\end{array}$$

Notez que la conjonction $p \leq \#ar \wedge p = 0$, qui vient de l'inclusion de $TSQ$, est cohérente.

**Les opérations** On spécifie les opérations d'ajout et de retrait d'un élément par les schémas $Append0$ et $Remove0$

$$\begin{array}{|l}
\multicolumn{1}{l}{Append0} \\ \hline
\Delta TSQ \\
from? : TSElem \\
item! : IN \\ \hline
ar' = ar \frown \langle from? \rangle \wedge \\
item! = \#ar' \wedge \\
p' = p \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Remove0} \\ \hline
\Delta TSQ \\
item! : TSElem \\ \hline
p < \#ar \wedge \\
p' = p + 1 \wedge \\
into! = ar(p') \wedge \\
ar' = ar \\ \hline
\end{array}$$

Le ? de $from$ et le ! de $item!$ sont conventionnels. Ils indiquent les « entrées-sorties » du schéma. C'est à dire les valeurs dont il faut disposer pour l'opération (ici, l'éléent à ajouter $from?$) et les valeurs données à l'issue de l'opération (ici, le nouveau nombre d'élément $item!$). Ces marques doivent être considérées comme des commentaires pour une implentation future. Ils n'ont pas de sens au niveau logique.

L'opération de retrait $Remove0$ a une *précondition* : le pointeur $p$ ne doit pas être en fin de file. C'est ce qu'exprime $p < \#ar$. En fin d'opération, il aura été incrémenté : $p' = p + 1$.

Les deux opérations suivantes utilisent les files comme des tableaux dont on peut modifier ($Write0$) ou consulter ($Read0$) une case :

$$\begin{array}{|l}
\multicolumn{1}{l}{Write0} \\
\hline
\Delta TSQ \\
item? : IN \\
from? : TSElem \\
\hline
item? \in 1..\#ar \land \\
ar' = ar \oplus \{item? \mapsto from?\} \land \\
p' = p \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Read0} \\
\hline
\Delta TSQ \\
item? : IN \\
into! : TSElem \\
\hline
item? \in 1..\#ar \land \\
into! = ar(item?) \land \\
p' = item? \land \\
ar' = ar \\
\hline
\end{array}$$

**Les erreurs** Nous avons vu qu'une opération comme $Remove0$ supposait une précondition. Si celle-ci n'est pas réalisée, il faut que l'opération le signale en positionnant un staut d'exécution. Pour ce, on se donne l'ensemble énuméré suivant :

$$OpStatus = \{Success, ItemErr, NoSpace\}$$

dont les éléments sont des constantes abstraites.

On donne dans un schéma les éléments communs à toute opération sur le statut d'exécution :

$$\begin{array}{|l}
\multicolumn{1}{l}{ERROR} \\
\hline
\Delta TSQ \\
report! : OpStatus \\
\hline
\theta TSQ' = \theta TSQ \\
\hline
\end{array}$$

L'opérateur $\theta$ appliqué à un schéma donne le n-uplet de tous les noms de variables que contient le schéma. On l'utilise ici pour signifier que toutes des variables de $TSQ$ doivent êtres égales à celle de $TSQ'$ (*i.e.* $ar' = ar$ et $p' = p$).

Pour chaque statut possible, on donne une opération spécifique :

$$\begin{array}{|l}
\multicolumn{1}{l}{NoneLeft} \\
\hline
ERROR \\
\hline
p = \#ar \land \\
report! = ItemErr \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{OutOfBounds} \\
\hline
ERROR \\
item? : IN \\
\hline
item? \notin 1..\#ar \land \\
report! = ItemErr \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{OutOfSpace} \\
\hline
ERROR \\
\hline
report! = NoSpace \\
\hline
\end{array}$$



$$\begin{array}{|l}
\multicolumn{1}{l}{Successful} \\
\hline
report! : OpStatus \\
\hline
report! = Success \\
\hline
\end{array}$$

Remarquez que les opérations $NoneLeft$ et $OutOfBounds$ expriment une condition pour que $report!$ prenne une valeur donnée alors que $OutOfSpace$ ne le fait pas. En effet, dans l'état actuel de la spécification, on ne dispose d'aucune information sur la taille de l'espace disponible pour stocker les files d'attente. La seule chose que l'on puisse faire, c'est de prévoir la possibilité d'un débordement de l'espace de stockage disponible.

**Redéfinitions avec gestion d'erreurs** Maintenant que les erreurs prévisibles ont été répertoriées et spécifiées, on redéfinit les opérations sur les files d'attente en intégrant la gestion du statut d'exécution. On obtient ainsi par simple combinaison logique des schémas les opérations :

$$\boxed{\begin{array}{lcl}
Append & \widehat{=} & (Append0 \land Successful) \lor OutOfSpace \\
Remove & \widehat{=} & (Remove0 \land Successful) \lor NoneLeft \\
Write & \widehat{=} & (Write0 \land Successful) \lor OutOfBound \lor OutOfSpace \\
Read & \widehat{=} & (Read0 \land Successful) \lor OutOfBound
\end{array}}$$

Les schémas obtenus se lisent simplement : ou l'opération a réussi et son satut est $Success$, ou l'opération a échoué avec l'un des statut d'erreur associé.

# 6 La méthode B

La *méthode B* définit, comme Z, un format de spécification : la *machine abstraite*. Elle reprend le concept de raffinement en l'adaptant aux machines abstraites. Mais contrairement à ce que nous avons présenté, B ne fournit pas un jeu prédéfini de règles de raffinemen *a priori* correctes, mais un mécanisme de vérification *a posteriori* engendrant des *obligations de preuves*. Enfin, le langage de spécification, s'il est basé sur un langage ensembliste, enrichi celui-ci d'une construction dédiée à la spécification de programmes : les *substitutions généralisées*.

## Substitutions généralisées

Les substitutions généralisées sont, en fait un langage d'instrutions (au sens usuel des langages impératifs) plus ou moins abstraites. En tant que substitutions, on peut les appliquer à un terme, une formule, voire une substitution, pour obtenir un nouveau terme, une nouvelle formule, voire substitution...

On note $[S]\Psi$ l'application de la substitution $S$ à l'expression, la formule ou la substitution $\Psi$. L'application des substitution est définie par induction sur $S$. Lorsque $S$ est une *substitution simple* (voir ci-dessous), son application est définie par induction sur $\Psi$.

**Substitution simple** La substitution de base est la substitution usuelle notée :

$$x \;:=\; E$$

L'application de la substitution simple $x := E$ est définie de façon usuelle sur les expressions et les formules :

$$\begin{array}{lcll}
[x := E]x & \widehat{=} & E & \\
[x := E]y & \widehat{=} & y & \text{si } x \neq y \\
[x := E]F(E_1, \ldots, E_n) & \widehat{=} & F([x := E]E_1, \ldots, [x := E]E_n) & \\
[x := E]\{x \in X \mid \varphi\} & \widehat{=} & \{x \in X \mid \varphi\} &
\end{array}$$

**Substitution parallèle** Si $S_1$ et $S_2$ sont deux substitutions *simples*, on note

$$S_1 \| S_2$$

la substitution appliquant *en même temps* et de façon indépendante, $S_1$ et $S_2$.

On généralise la substitution parallèle en *substitution multiple*

$$x_1, \ldots, x_n := E_1, \ldots, E_n$$

Soient $z_2, \ldots, z_n$ des variables toutes différentes entre elles, différentes des $x_1, \ldots, x_n$ et n'apparaissant dans aucunes des $E_1, \ldots, E_n$, $F$ on pose :

$$[x_1, \ldots, x_n := E_1, \ldots, E_n]F \quad \widehat{=} \quad [z_n := E_n] \ldots [z_2 := E_2][x_1 := E_1][x_2 := z_2] \ldots [x_n := z_n]F$$

Pour faire court, on dira que les $z_i$ sont des *nouvelles variables*. On a recours au *renommage* pour éviter que les $x_i$ pouvant apparaître dans les $E_2$, ..., $E_n$ ne soient affectées par la mise en séquence.

On s'autorisera à combiner des substitutions multiples avec l'opérateur $\|$. Par exemple :

$$x_1 := E_1 \| x_2, x_3 := E_2, E_3 \;\widehat{=}\; x_1, x_2, x_3 := E_1, E_2, E_3$$

**Substitution préconditionnée** Si $\varphi$ est une formule et $S$ une substitution, on note

$$\text{PRE } \varphi \text{ THEN } S$$

la substitution imposant que $\varphi$ soit satisfaite pour appliquer $S$ :

$$[\text{PRE } \varphi \text{ THEN } S]F \quad \widehat{=} \quad \varphi \wedge [S]F$$

**Substitution indéterminée** Si $x_1, \ldots, x_n$ sont des variables, $\varphi$ un formule et $S$ une substitution, on note

$$\text{ANY } x_1, \ldots, x_n \text{ WHERE } \varphi \text{ THEN } S$$

la substitution qui consiste à choisir n'importe quels $x_1, \ldots, x_n$ qui satisfont $\varphi$ pour appliquer $S$ :

$$[\text{ANY } x_1, \ldots, x_n \text{ WHERE } \varphi \text{ THEN } S]F \quad \widehat{=} \quad \forall z_1 \ldots z_n . (\varphi' \Rightarrow [S']F)$$

avec $z_1, \ldots, z_n$ nouvelles variables, $\varphi' = [x_1, \ldots, x_n := z_1, \ldots, z_n]\varphi$ et $S' = [x_1, \ldots, x_n := z_1, \ldots, z_n]S$. Le renommage est rendu nécessaire à cause de l'introduction du quantificateur universel.

## Machines abstraites

Intuitivement, une machine abstraite peut être comparée à un un *objet* comprenant un état interne (les VARIABLES) et des moyens d'actions sur cet état (les OPERATIONS). Comme une machine est un élément de spécification, elle contiendra également des *commentaires logiques* exprimant ses propriétés (l'INVARIANT).

Une machine est constituée de plusieurs rubriques jouant chacune un rôle dans la description des spécifications. Chacune de ces rubriques est identifiée par un mot clef. Voici quelles sont les rubriques essentielles d'une machine.

MACHINE cette rubrique ne contient qu'un seul élément : le nom de la machine.



SETS cette rubrique contient la déclaration des *ensembles* dont se servira la machine. On peut déclarer un ensemble soit comme un simple identificateur (auquel cas son contenu reste abstrait) soit par extension (comme un ensemble énuméré) Tous ces ensembles sont finis (explicitement ou implicitement)

VARIABLES cette rubrique contient la déclaration des variables qu'utilise la machine. L'ensemble de ces variables constitue l'*état* interne de la machine.

INVARIANT cette rubrique contient une formule. C'est un élément trés important de la spécification. L'invariant contient la propriété globale que doit satisfaire la machine. Nous reviendrons ultérieurement sur ce point.

INITIALIZATION cette rubrique contient une susbtitution (généralisée) qui définit la valeur initiale des variables.

OPERATIONS cette rubrique contient la définition d'une liste d'objets appelés *opérations*. Une opération peut modifier l'état de la machine, prendre des arguments ou (inclusif) renvoyer une valeur. Les opérations sont définies en termes de substitutions généralisées.

On peut rajouter encore deux rubriques permetant d'introduire des constantes :

CONSTANTS cette rubrique contient les déclarations des noms des constantes.

CONSTRAINTS cette rubrique contient les axiomes caratérisant les constantes.

## L'exemple

Nous allons utiliser tout au long de notre étude de la méthode B un exemple tiré d'une petite étude de cas que J.-Y. Chauvet a publiée dans *1st Conference on the B method, Proceedings, ed. Henri Habrias, Nantes 1996* : le *CARREFOUR*.

### Spécification informelle

> La circulation d'un carrefour est réglée par deux feux tricolores dont les couleurs sont vert, orange ou rouge. Sur chacun des feux une seule des couleurs est active à la fois. Le système de feux du carrefour peut être en service ou hors service. Lorsque le système est hors service, les deux feux sont oranges. Lorsque le système est en service, la couleur de chacun des feux change suivant le cycle : orange puis rouge puis vert puis orange, etc ...
> Un véhicule ne peut s'engager sur une voie que si le feu n'est pas rouge. Lorsque le système est en service, les feux doivent être réglés de façon à ce que deux véhicules venant de voies différentes ne se trouvent pas en même temps sur le carrefour.
> On désire obtenir un système assurant la mise en route du carrefour et la gestion du changement de couleur des feux.

De cette spécification informelle, il faut extraire les composantes essentielles et les traduire en objets formels.

**Pré-spécification formelle** Il est utile pour pouvoir valider la pertinance de l'analyse de la spécification formelle, de garder trace de l'origine des éléments formels proposés.

1. « les couleurs sont vert, orange ou rouge » :

$$Couleurs = \{vert, orange, rouge\}$$

2. « deux feux tricolores » :

$$feuA \in Couleurs, \; feuB \in Couleurs$$

3. « cycle : orange puis rouge puis vert » :

$$Succ \in Couleurs \to Couleurs$$

$$orange \mapsto rouge$$

$$rouge \mapsto vert$$

$$vert \mapsto orange$$

4. « en service ou hors service » :

$$Etats = \{hs, es\}$$

$$etat \in Etats$$

5. « Lorsque le système est hors service, les deux feux sont oranges » :

$$(etat = hs) \Rightarrow (feuA = orange \land feuB = orange)$$

6. « Lorsque le système est en service, les feux doivent être réglés de façon à ce que deux véhicules venant de voies différentes ne se trouvent pas en même temps sur le carrefour » :

$$(etat = es) \Rightarrow ((feuA = rouge \lor feuB = rouge) \land feuA \neq feuB)$$

Notez que notre formule dit plus que ce que réclamait la spécification informelle. Nous avons ajouté à la propriété réclamée de *sécurité*, une propriété de *disponibilité* garantissant que l'un des feux permet le passage des véhicules.

**Spécification formelle** Il faut maintenant réunir les éléments formels retenus dans le cadre des machines abstraites.

```
MACHINE
  CARREFOUR

SETS
  Etats = {hs, es}
  Couleurs = {vert, orange, rouge}

CONSTANTS
  Succ

CONSTRAINTS
  Succ ∈ Couleurs → Couleurs ∧
  Succ(orange) = rouge ∧
  Succ(rouge) = vert ∧
  Succ(vert) = orange

VARIABLES
  etat, feuA, feuB

INVARIANT
  etat ∈ Etats ∧
  feuA ∈ Couleurs ∧
  feuB ∈ Couleurs ∧

  (etat = hs) ⇒ (feuA = orange ∧ feuB = orange) ∧
  (etat = es) ⇒ ((feuA = rouge ∨ feuB = rouge) ∧ feuA ≠ feuB)

INITIALIZATION
  etat, feuA, feuB := hs, orange, orange

OPERATIONS
```



```
MiseEnService ≙
  PRE etat = hs THEN
    ANY f1, f2 WHERE
      f1 ∈ Couleurs ∧ f2 ∈ Couleurs ∧
      (f1 = rouge ∨ f2 = rouge) ∧ f1 ≠ f2
    THEN
      etat, feuA, feuB := es, f1, f2

Changement ≙
  PRE etat = es THEN
    ANY f1, f2 WHERE
      f1 ∈ Couleurs ∧ f2 ∈ Couleurs ∧
      (f1 = rouge ∨ f2 = rouge) ∧ f1 ≠ f2 ∧
      (f1 = Succ(feuA)) ∨ (f2 = Succ(feuB))
    THEN
      feuA, feuB := f1, f2
```

Remarquez que dans l'opération de changement de couleur, on a rajouté une propriété de *vivacité* : la formule $f_1 = Succ(feuA)) \lor (f_2 = Succ(feuB)$ énonce que la couleur d'un des deux feux au moins change effectivement.

### Correction des machines abstraites

La communauté B a l'habitude de décrire une machine abstraite comme un systéme possédant un état initial et offrant à son utilisateur une série de *boutons* (les opérations) permettant d'agir sur cet état. La correcion d'une machine est établit vis-à-vis de l'invariant :

1. L'initialisation doit *établir* l'invariant.
2. Chaque opération doit *conserver* l'invariant.

Soit le schéma de machine suivant :

```
MACHINE
  M
SETS
  X
VARIABLES
  x
INVARIANT
  I
INITIALIZATION
  S0
OPERATIONS
  O1 ≙
    PRE φ THEN S1
```

Puisque l'initialisation et les opérations sont définies comme des substitution, la correction de la machine $M$ est exprimée par les formules obtenues n appliqunt ces substitutions à la formule de l'inivariant :

1. $[S_0]I$
2. $I \land \varphi \Rightarrow [S_1]I$

**Correction de $CARREFOUR$**

Décomposons la formule de l'invariant de $CARREFOUR$ en trois conjonctions : $I_1 \wedge I_2 \wedge I_3$ avec

$I_1 \mathrel{\hat=} etat \in Etats \wedge feuA \in Couleurs \wedge feuB \in Couleurs$
$I_2 \mathrel{\hat=} (etat = hs) \Rightarrow (feuA = orange \wedge feuB = orange)$
$I_3 \mathrel{\hat=} (etat = es) \Rightarrow ((feuA = rouge \vee feuB = rouge) \wedge feuA \neq feuB)$

**Initialisation** Posons

$$S_0 \mathrel{\hat=} etat, feuA, feuB \ := \ hs, orange, orange$$

L'obligation de preuve de correction de l'initialisation est la formule obtenue en développant l'application

$$[S_0](I_1 \wedge I_2 \wedge I_3)$$

Comme dans la substitution multiple $S_0$, aucune des variables apparaissant à gauche du signe $:=$ n'apparaît à sa droite, on peut traiter cette substitution multiple comme une substitution simple et affirmer que notre obligation de preuve est logiquement équivalente à la formule obtenue en développant

$$[S_0]I_1 \wedge [S_0]I_2 \wedge [S_0]I_3$$

Nous utiliserons par la suite systématiquement cette équivalence pour simplifier le développement des applications de substitutions multiples.

Il faut donc montrer la validité des trois formules $[S_0]I_1$, $[S_0]I_2$ et $[S_0]I_3$.

1. $[S_0]I_1 \mathrel{\hat=} es \in Etats \wedge orange \in Couleurs \wedge orange \in Couleurs$
   qui est trivialement vraie par définition des ensembles $Etats$ et $Couleurs$.
2. $[S_0]I_2 \mathrel{\hat=} (hs = hs) \Rightarrow (orange = orange \wedge orange = orange)$
   qui est on ne peut plus trivialement vraie puisque le vrai entraîne la vrai.
3. $[S_0]I_2 \mathrel{\hat=} (es = hs) \Rightarrow ((orange = rouge \vee orange = rouge) \wedge orange \neq orange)$
   qui est tout aussi trivialement vraie puisque le faux $(es = hs)$ entraîne n'importe quoi.

**MiseEnRoute** Soit $S_1$ la substitution de mise en route, posons

$$S_1 \mathrel{\hat=} \text{PRE } \varphi_1 \text{ THEN } S'_1$$

avec

$$\varphi_1 \mathrel{\hat=} etat = hs$$
$$S'_1 \mathrel{\hat=} \text{ANY } f_1, f_2 \text{ WHERE } \psi_1^1 \wedge \psi_1^2 \text{ THEN } S''_1$$

et

$$\psi_1^1 \mathrel{\hat=} f_1 \in Couleurs \wedge f_2 \in Couleurs$$
$$\psi_1^2 \mathrel{\hat=} (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2$$
$$S''_1 \mathrel{\hat=} etat, feuA, feuB \ := \ es, f_1, f_2$$

Il faut montrer que $I \wedge \varphi_1 \Rightarrow [S'_1]I$, c'est-à-dire, $I \wedge \varphi_1 \Rightarrow \forall f_1, f_2.(\psi_1^1 \wedge \psi_1^2 \Rightarrow [S''_1]I)$. Pour ce, on suppose $I$ et $\varphi_1$ et on se donne $f_1$ et $f_2$ telles que $\psi_1^1$ (ie $f_1 \in Couleurs \wedge f_2 \in Couleurs$) et $\psi_1^2$ (ie $(f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2$) ; reste à montrer $[S''_1]I$, c'est-à-dire $[S''_1]I_1$, $[S''_1]I_2$ et $[S''_1]I_3$.

1. $[S''_1]I_1 \mathrel{\hat=} es \in Etats \wedge f_1 \in Couleurs \wedge f_2 \in Couleurs$.
   On a $es \in Etats$ par définition de l'ensemble $Etats$ ; et $f_1 \in Couleurs$ et $f_2 \in Couleurs$ par hypothèse ($\psi_1^1$).
2. $[S''_1]I_2 \mathrel{\hat=} (es = hs) \Rightarrow (f_1 = orange \wedge f_2 = orange)$
   Ce qui est trivialement vrai puisque $es \neq hs$.
3. $[S''_1]I_3 \mathrel{\hat=} (es = es) \Rightarrow ((f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2)$.
   Ce qui est rendu vrai par l'hypothèse $\psi_1^2$.



**Changement** Soit $S_2$ la substitution définissant le changement de couleur des feux, posons

$$S_2 \mathrel{\hat=} \text{PRE } \varphi_2 \text{ THEN } S'_2$$

avec

$$\varphi_2 \mathrel{\hat=} etat = hs$$
$$S'_2 \mathrel{\hat=} \text{ANY } f_1, f_2 \text{ WHERE } \psi_2^1 \wedge \psi_2^2 \wedge \psi_2^3 \text{ THEN } S''_2$$

et

$$\psi_2^1 \mathrel{\hat=} f_1 \in Couleurs \wedge f_2 \in Couleurs$$
$$\psi_2^2 \mathrel{\hat=} (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2$$
$$\psi_2^3 \mathrel{\hat=} (f_1 = Succ(feuA)) \vee (f_2 = Succ(feuB))$$
$$S''_2 \mathrel{\hat=} feuA, feuB \ := \ f_1, f_2$$

Il faut montrer que $I \wedge \varphi_2 \Rightarrow [S'_2]I$, c'est-à-dire, $I \wedge \varphi_2 \Rightarrow \forall f_1, f_2.(\psi_2^1 \wedge \psi_2^2 \wedge \psi_2^3 \Rightarrow [S''_2]I)$. Pour ce, on suppose $I$ et $\varphi_2$ et on se donne $f_1$ et $f_2$ telles que $\psi_2^1$ (ie $f_1 \in Couleurs \wedge f_2 \in Couleurs$), $\psi_2^2$ (ie $(f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2$) et $\psi_2^3$ (ie $(f_1 = Succ(feuA)) \vee (f_2 = Succ(feuB))$) ; reste à montrer $[S''_2]I$, c'est-à-dire $[S''_2]I_1$, $[S''_2]I_2$ et $[S''_2]I_3$.

1. $[S''_2]I_1 \mathrel{\hat=} etat \in Etats \wedge f_1 \in Couleurs \wedge f_2 \in Couleurs$.
   On a $etat \in Etats$ par l'hypothèse $I$ et $f_1 \in Couleurs$ et $f_2 \in Couleurs$ par $\psi_2^1$.
2. $[S''_2]I_2 \mathrel{\hat=} (etat = hs) \Rightarrow (f_1 = orange \wedge f_2 = orange)$.
   Par l'hypothèse $\varphi_2$ (la précondition), on a que $etat = es$. Ce qui rend trivialement vraie l'implication recherchée.
3. $[S''_2]I_3 \mathrel{\hat=} (etat = es) \Rightarrow ((f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2)$.
   Ce qui nous est donné par $I$ et $\psi_2^2$

**Commentaire sur cette première machine abstraite** Les deux opérations de la machine $CARREFOUR$ peuvent se paraphraser par : « *n'importe quelles valeurs qui satisfasse au moins l'invariant* ». Dés lors, la correction est facilement acquise.

Néanmoins, dans les preuves de correction, il faut noter comment la précondition intervient, rendant caduque l'examen de certains cas de l'invariant (ici, $etat = hs$, par exemple).

## Raffinement

On peut raffiner une machine sur deux points : les données ou(inclusif) les opérations (c'est à dire, les substitutions). Le raffinement des substitutions porte à son tour sur deux points : affaiblissement des préconditions ou levée de l'indéterminisme. Il va de soit que dans la plus part des cas, le raffinement de données imlique un raffinement d'opération. Une machine et son raffinement définissent les *mêmes opérérations*, seuls peuvent changer les variables, les ensembles et les substitutions définissant les opérations.

Lorsque l'on passe d'une machine à son raffinement, on change d'espace de variables. Pour pouvoir contrôler la légitimité du raffinement, il faut établir le lien entre l'état interne (ie les variables) de la machine originale et celui de son raffinement. On exprime ce lien comme une sous-formule de l'invariant du raffinement que l'on appelle *invariant de liaison*. Cette sous-formule utilise aussi bien les variables de la machine originale que celles du raffinement.

Un raffinement n'est un raffinement correct que s'il conserve la validité des propriétés de la machine d'origine. La méthode B permet de s'assurer de la légitimité d'un raffinement en engendrant des *obligations de preuves* énonçant cette propriété de conservation.

### Obligations de preuve

Supposons donc une machine $M$ dont l'invariant est $I$, l'initialisation est $S0$ et n'ayant qu'une seule opération définie par la substitution conditionnelle $\text{PRE } \varphi \text{ THEN } S1$. Supposons également que cette machine est correcte. C'est à dire que l'on a effectivement démontré que les substitution $S0$ et PRE $\varphi$ THEN $S1$ conservent l'invariant $I$ qui exprime les propriétés statiques du système que l'on a en tête.

Soit alors, une machine $R$ dont l'invariant (de liaison) est $J$, l'initialisation est $T0$ et définissant la même opération que $M$, mais par la substitution PRE $\psi$ THEN $T1$. Pour établir que $R$ est un raffinement de $M$, il faut s'assurer que celle-là vérifie encore les propriétés que vérifie celle-ci. En particulier, $R$ doit satisfaire, en quelque sorte, $I$. Mais puisque $M$ et $R$ sont deux machines distinctes, elles agissent sur des variables différentes. La machine $R$ ne peut donc directement établir $I$. Pour, depuis la machine $R$ pouvoir parler des variables de $M$, on utilise le bien nommé invariant de liaison $J$ dont le rôle est justement de *mettre en relation* les variables de $M$ et de $R$.

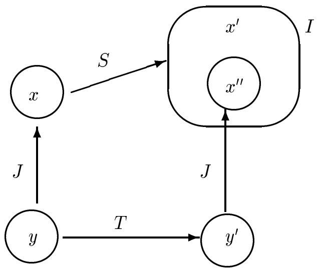
Formellement, les obligations de preuve engendrées par le raffinement de $M$ par $R$ sont

1. $[T0]\neg[S0]\neg J$ et
2. $I \wedge J \wedge \varphi \Rightarrow \psi \wedge [T1]\neg[S1]\neg J$

Pour comprendre intuitivement ce que signifient ces formules nous allons supposer que $M$ travaille sur une variable $x$ appartenant à un ensemble $X$, et $R$, sur une variable $y$ appartenant à un ensemble $Y$. L'invariant de liaison $J$ exprime une relation (au sens mathématique du terme) entre $X$ et $Y$. Une susbtitution peut également être vue comme une relation : la relation qui à chaque valeur initiale d'une variable associe la valeur obtenue par application de la substitution. Soit alors une substitution $S$ de $M$ et $T$ une substitution de $R$. Si nous appelons $x'$ l'ensemble des valeurs associées à $x \in X$ par $S$ et $y'$ l'ensemble des valeurs associées à $y$ par $T$, si nous appelons $x''$ l'ensemble des valeurs que $J$ associe à $y'$ dans $X$ alors la formule $[T]\neg[S]\neg J$ nous dit que :

*il n'est pas possible que $x''$ ne soit pas une partie de $x'$*

La figure ci-dessous illustre cet énoncé.




Puisque l'on a supposé que $M$ est correcte, ses substitutions établissent $I$. L'ensemble de valeurs $x'$ satisfait donc $I$. En démontrant l'obligation de preuve de raffinement $[T]\neg[S]\neg J$ on obtient donc bien que $x''$ satisfait également $I$.

Dans l'obligation de preuve concernant les opérations apparaissent explicitement leur préconditions ; celle de l'opération de $M$ en antécédant et celle de l'opération de $R$ en conséquent. Il faudra donc prouver, en particulier, que $I \wedge J \wedge \varphi \Rightarrow \psi$. Ce qui nous donnera que $\psi$ (la précondition du raffinement) est satisfaite chaque fois que $\varphi$ (la précondition originelle) est satisfaite. On pourra alors utiliser de façon licite, à la fois les substitutions $T1$ et $S1$.

Nous allons à présent illustrer successivement comment utiliser nos deux formes de raffinements en revenant à l'exemple de gestion des feux d'un carrefour.

## Raffinement d'opération

Nous allons préciser de faç définitive l'opération de mise en service et de façon encore transitoire l'opération de changement de couleur.



**Substitutions généralisées en plus**

Pour notre seconde opération, nous allons utiliser deux nouvelles substitutions généralisées.

**Substitution gardée** Comme la substitution conditionnelle, la substitution gradée est une substitution soumise à la satisfaction d'une certaine condition, ou, plus précisément, un certain *test*. La différence est que dans le cas de la substitution conditionnelle, c'est à l'*utilisateur extérieur* de vérifier la validité de la condition avant d'utiliser la substitution, alors que dans le cas de la substitution gardée, c'est la *substitution elle-même* qui doit vérifier la validité du test. Cette différence est manifeste lorsque l'on regarde comment une substitution gardée s'applique à une formule.

Si $\varphi$ est une formule et $S$ une substitution, la substitution gardée s'écrit

WHEN $\varphi$ THEN $S$

On définit l'application de la substitution gardée par

$$[\text{WHEN } \varphi \text{ THEN } S]F \mathrel{\widehat{=}} \varphi \Rightarrow [S]F$$

Alors que l'application d'une substitution conditionnelle engendrait une conjonction, nous avons ici une implication. Dans le cas de la substitution gardée, la formule de garde $\psi$ décrit un état interne pouvant ne pas apparaître. Dans ce cas, on ne « déclanchera » pas la substitution $S$. On n'a donc pas d'obligation de toujours satisfaire $\varphi$. Il suffit de s'assurer que chaque fois que $\varphi$ est vraie, $S$ établit bien $F$. C'est bien ce qu'exprime le connecteur propositionnel $\Rightarrow$.

**Substitution à choix borné** La substitution à choix borné est une autre sorte de substitution indéterminée. Elle permet de pouvoir utiliser au choix un nombre fini de substitutions. Elle s'écrit

CHOICE $S_1$ OR ... OR $S_n$

Chacune des substitutions proposées par un choix borné doit pouvoir être utilisée de façon indifférente. On définit donc l'application d'une substituiton à choix borné par une conjonction :

$$[\text{CHOICE } S_1 \text{ OR} \ldots \text{OR } S_n]F \mathrel{\widehat{=}} [S_1]F \wedge \ldots \wedge [S_n]F$$

**Le raffinement $CARREFOUR1$**

Pour raffiner l'opération de mise en service, nous allons choisir arbitrairement de mettre un feu au vert et l'autre au rouge.
L'idée du raffinement de l'opération de changement de couleur est de spécifier la fonction *Succ* comme substitution décrivant les divers cas de figures déterminés par les équations qui définissent la fonction *Succ*. C'est une première étape vers l'*implémentation* de la fonction mathématique *Succ*.

Une machine en raffinant une autre n'est pas nommée par la rubrique MACHINE, mais par la rubrique REFINEMENT. De plus, elle doit obligatoirement contenir une rubrique additionnelle REFINES indiquant quelle machine elle raffine.

REFINEMENT
  $CARREFOUR1$

REFINES
  $CARREFOUR$

SETS
  $Etats = \{hs, es\}$

$Couleurs = \{vert, orange, rouge\}$

VARIABLES
$etat_1, feuA_1, feuB_1$

INVARIANT
$etat_1 = etat \wedge$
$feuA_1 = feuA \wedge$
$feuB_1 = feuB$

INITIALIZATION
$etat_1, feuA_1, feuB_1$ := $hs, orange, orange$

OPERATIONS
MiseEnService $\widehat{=}$
PRE $etat_1 = hs$ THEN
$etat_1, feuA_1, feuB_1$ := $es, rouge, vert$

Changement $\widehat{=}$
PRE $etat_1 = es$ THEN
CHOICE
WHEN $feuA_1 = vert$ THEN $feuA_1$ := $orange$
OR
WHEN $feuA_1 = orange$ THEN $feuA_1, feuB_1$ := $rouge, vert$
OR
WHEN $feuA_1 = rouge \wedge feuB_1 = vert$ THEN $feuB_1$ := $orange$
OR
WHEN $feuA_1 = rouge \wedge feuB_1 = orange$ THEN $feuA_1, feuB_1$ := $vert, rouge$

### Obligations de preuves

Nous donnons, sans tout le détail de leur calcul, les obligations de preuves engendrées.

**Initialisation** On obtient la formule triviale

$$\neg\neg(hs = hs \wedge orange = orange \wedge orange = orange)$$

**Mise en service** Appelons $J$ l'invariant de $CARREFOUR1$. On obtient la formule

$$\begin{array}{l} I \wedge J \wedge etat = hs \Rightarrow \\ \quad etat_1 = hs \wedge \\ \quad \neg(\forall f_1, f_2.(f_1 \in Couleurs \wedge f_2 \in Couleurs \wedge (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2 \Rightarrow \\ \qquad\qquad \neg(es = es \wedge f_1 = rouge \wedge f_2 = vert))) \end{array}$$

Ce qui donne, en simplifiant les négations

$$\begin{array}{l} I \wedge J \wedge etat = hs \Rightarrow \\ \quad etat_1 = hs \wedge \\ \quad \exists f_1, f_2.(f_1 \in Couleurs \wedge f_2 \in Couleurs \wedge (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2 \wedge \\ \qquad\qquad es = es \wedge f_1 = rouge \wedge f_2 = vert) \end{array}$$

On obtient $etat_1 = hs$ de $etat_1 = etat$ (*cf* $J$) et l'hopothèse $etat = hs$.
On obtient le second membre de la conjonction en prenant $rouge$ et $vert$ pour valeurs respectives de $f_1$ et $f_2$.



Il convient de noter ici comment la quatification universlle de ls substitution déterminée est devenue une quantification existentielle par le jeu de la négation qu'introduit l'obligation de preuve de raffinement. Pour montrer la validité d'une formule existentielle, il nous faut exhiber au moins une valeur satisfaisante (ou, à tout le moins montrer qu'on ne peut pas supposer qu'il n'en existe pas, si l'on raisonne par l'absurde). Ainsi l'obligation de preuve du raffinement rajoute de l'information par rapport à la correction de la machine originelle.

**Changement de couleur** Soit $S_2$ la substitution définissant le changement de couleur des feux de la machine initiale $CARREFOUR$, posons

$$S_2 \widehat{=} \text{PRE } \varphi_2 \text{ THEN ANY } f_1, f_2 \text{ WHERE } \psi_2 \text{ THEN } S_2'$$

Calculons, dans un premier temps $\neg[S2]\neg J$. En simplifiant les négations, on obtient

$$\exists f_1, f_2.(\psi_2 \wedge [S_2']J)$$

Soit maintenant $T_2$ la substitution définissant l'opération de changement de couleur dans le raffinement $CARREFOUR1$. Posons

$$T_2 \widehat{=} \text{PRE } \varphi_2 \text{ THEN CHOICE } T_1' \text{ OR } T_2' \text{ OR } T_3' \text{ OR } T_4'$$

Avec, pour chaque $i \in [1..4]$,

$$T_i' \widehat{=} \text{WHEN } \gamma_i \text{ THEN } T_i''$$

L'application $[T_2]\exists f_1, f_2.(\gamma_2 \wedge [S_2']J)$ nous donne la conjonction

$$\begin{array}{l} (\gamma_1 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_1''][S_2']J)) \wedge \\ (\gamma_2 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_2''][S_2']J)) \wedge \\ (\gamma_3 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_3''][S_2']J)) \wedge \\ (\gamma_4 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_4''][S_2']J)) \end{array}$$

Au final, l'obligation de preuve concernant l'opération de changement de couleur s'écrit

$$\begin{array}{l} I \wedge J \wedge etat = es \Rightarrow \\ \quad (etat_1 = es) \wedge \\ \quad (\gamma_1 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_1''][S_2']J)) \wedge \\ \quad (\gamma_2 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_2''][S_2']J)) \wedge \\ \quad (\gamma_3 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_3''][S_2']J)) \wedge \\ \quad (\gamma_4 \Rightarrow \exists f_1, f_2.(\psi_2 \wedge [T_4''][S_2']J)) \end{array}$$

On obtient $etat_1 = es$ comme précédemment.
Traitons le premier des quatre autres membres de la conjonction et laissons les derniers en exercice. Rappelons que

$$\gamma_1 \widehat{=} feuA_1 = vert$$

que

$$\begin{array}{ll} \psi_2 \widehat{=} & f_1 \in Couleurs \wedge f_2 \in Couleurs \wedge \\ & (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2 \wedge \\ & (f_1 = Succ(feuA)) \vee (f_2 = Succ(feuB)) \end{array}$$

et que

$$[T_1''][S_2']J \widehat{=} etat_1 = es \wedge orange = f_1 \wedge feuB_1 = f_2$$

Il faut donc montrer que sous les hypothèses $I$, $J$, $etat = es$ et $feuA_1 = vert$, on peut trouver deux valeurs pour $f_1$ et $f_2$ telles que l'on ait la conjonction

$$\begin{array}{l} f_1 \in Couleurs \wedge f_2 \in Couleurs \wedge \\ (f_1 = rouge \vee f_2 = rouge) \wedge f_1 \neq f_2 \wedge \\ (f_1 = Succ(feuA)) \vee (f_2 = Succ(feuB)) \wedge \\ etat_1 = es \wedge orange = f_1 \wedge feuB_1 = f_2 \end{array}$$

Le choix de $f_1$ est simple, puisqu'il faut satisfaire que $orange = f_1$. Pour ce qui est de $f_2$, on peut faire le choix minimaliste en prenant $f_2 = feuB$. Notre obligation de preuve se résume alors à la vérification des neufs formules suivantes.

1. $orange \in Couleurs$
   Trivial par définition de $Couleurs$.
2. $feuB \in Couleurs$
   On l'a par hypothèse $I$.
3. $orange = rouge \vee feuB = rouge$
   On ne pourra pas avoir $orange = rouge$, montrons donc $feuB = rouge$ : On a l'hypothèse $feuA_1 = vert$ ainsi que (par $J$) $feuA_1 = feuA$. On a donc que $feuA = vert$. Or $I$ nous dit que $feuA = rouge \vee feuB = rouge$, comme $feuA = vert \neq rouge$, il faut bien que $feuB = rouge$.
4. $orange \neq feuB$
   On l'obtient en montrant comme ci-dessus qu'en fait $feuB = rouge$.
5. $orange = Succ(feuA) \vee feuB = Succ(feuB)$
   Trivial en utilisant l'hypothèse $feuA = vert$.
6. $etat_1 = es$
   On l'a déjà montré...
7. $orange = orange$
   Sans commentaire...
8. $feuB_1 = feuB$
   On l'a par l'hypothèse $J$.

## Raffinement de données

Une spécification manipule en général des données abstraites : des ensembles. Une étape importante du raffinement est donc d'obtenir une repréntation concrètes des données.

Nous illustrons cette catégorie particulière de raffinement sur notre exemple en introduisant l'usage de valeurs booléennes en place des états et couleurs. Ceci implique restructuration de la représentation des feux qui à son tour implique une reformulation des opérations. Cependant, pour minimiser l'impact de la modification de représentation des données, nous conserverons la structure *logique* des substitutions définissant les opérations.

La relation entre les variables de la machine originale et de son raffinement est exprimée dans l'invariant du raffinement.

### Le raffinement $CARREFOUR2$

Puisque l'état du système est à valeur dans un ensemble à deux élément, on peut utiliser des booléens. De plus, on peut représenter chaque feux par un triplet de booléens dont, suivant l'intuition, chaque composante correspond à une lampe, éteinte ou allumée, d'une couleur donnée. Chaque lampe pouvant prendre deux valeurs, on utilise ici encore les booléens.

L'ensemble des booléens est supposé (pré)défini par : $BOOL = \{true, false\}$.

REFINEMENT
$CARREFOUR2$

REFINES
$CARREFOUR1$

VARIABLES
$etat_2, Av, Ao, Ar, Bv, Bo, Br$



INVARIANT
$etat_2 \in BOOL \wedge$
$Av, Ao, Ar \in BOOL \times BOOL \times BOOL \wedge$
$Bv, Bo, Br \in BOOL \times BOOL \times BOOL \wedge$

$(etat_2 = true \Leftrightarrow etat_1 = es) \wedge$
$(feuA_1 = vert \Leftrightarrow Av = true \wedge Ao = false \wedge Ar = false) \wedge$
$(feuA_1 = orange \Leftrightarrow Av = false \wedge Ao = true \wedge Ar = false) \wedge$
$(feuA_1 = rouge \Leftrightarrow Av = false \wedge Ao = false \wedge Ar = true) \wedge$
$(feuB_1 = vert \Leftrightarrow Bv = true \wedge Bo = false \wedge Br = false) \wedge$
$(feuB_1 = orange \Leftrightarrow Bv = false \wedge Bo = true \wedge Br = false) \wedge$
$(feuB_1 = rouge \Leftrightarrow Bv = false \wedge Bo = false \wedge Br = true)$

INITIALIZATION
$etat_2 := false \parallel$
$Av, Ao, Ar := false, true, false \parallel$
$Bv, Bo, Br := false, true, false$

OPERATIONS
MiseEnService $\widehat{=}$
PRE $etat_2 = false$ THEN
$etat_2 := true \parallel$
$Ao, Ar := false, true \parallel$
$Bv, Bo := true, false$

Changement $\widehat{=}$
PRE $etat_2 = true$ THEN
CHOICE
WHEN $Av = true$ THEN $Av, Ao := false, true$
OR
WHEN $Ao = true$ THEN $Ao, Ar := false, true \parallel Bv, Br := true, false$
OR
WHEN $Ar = true \wedge Bv = true$ THEN $Bo, Bv := true, false$
OR
WHEN $Ar = true \wedge Bo = true$ THEN $Ar, Av := false, true \parallel Bo, Br := false, true$

### Obligations de preuves

Les obligations de preuves vont ici essentiellement contrôler la cohérence de l'usage de la nouvelle représentation des données dans les opérations vis-à-vis de la façon dont l'invariant définit cette représentation.

**Initialisation** On obtient, en simplifiant la double négation,

$$\begin{array}{l}
false \in BOOL \wedge \\
false, true, false \in BOOL \times BOOL \times BOOL \wedge \\
false, true, false \in BOOL \times BOOL \times BOOL \wedge \\
\\
(false = true \Leftrightarrow hs = es) \wedge \\
(orange = vert \Leftrightarrow false = true \ \wedge\ true = false \ \wedge\ false = false) \wedge \\
(orange = orange \Leftrightarrow false = false \ \wedge\ true = true \ \wedge\ false = false) \wedge \\
(orange = rouge \Leftrightarrow false = false \ \wedge\ true = false \ \wedge\ false = true) \wedge \\
(orange = vert \Leftrightarrow false = true \ \wedge\ true = false \ \wedge\ false = false) \wedge \\
(orange = orange \Leftrightarrow false = false \ \wedge\ true = true \ \wedge\ false = false) \wedge \\
(orange = rouge \Leftrightarrow false = false \ \wedge\ true = false \ \wedge\ false = true)
\end{array}$$

On vérifie facilement la validité de la formule obtenue sachant que deux propositions sont équivalentes lorsqu'elles ont même valeur de vérité; soit vrai, soit faux.

**Mise en service** Nous avons présenté les obligations de preuve de correction du raffinement dans le cadre simplifié d'une machine originale et d'un raffinement de celle-ci. Ici, Nous sommes en présence de trois machines dans une relation de raffinement mutuelle: $CARREFOUR \supseteq CARREFOUR1 \supseteq CARREFOUR2$ (pour reprendre la notation de 4). Les obligation de preuves établissant que $CARREFOUR1 \supseteq CARREFOUR2$ doivent prendre en compte l'invariant de $CARREFOUR$ puisqu'en fin de compte, c'est toujours cette machine originale que nous raffinons. D'ailleurs, une formule ne faisant intervenir que les invariant de $CARREFOUR2$ et $CARREFOUR1$ aurait peu de sens puisque l'invariant (de liaison) de $CARREFOUR1$ fait référence aux variables de $CARREFOUR$.

Si $K$ est l'invariant de $CARREFOUR2$, si on écrit son opération de mise en service sous la forme

$$\text{PRE } etat_2 = false \text{ THEN } U_1$$

si l'on écrit l'opération de mise en service de $CARREFOUR1$ sous la forme

$$\text{PRE } etat_1 = hs \text{ THEN } T_1$$

alors la formule à calculer pour obtenir l'obligation de preuve est

$$I \wedge J \wedge K \wedge etat = hs \wedge etat_1 = hs \Rightarrow etat_2 = false \wedge [U_1]\neg[T_1]\neg K$$

Ce qui donne

$$\begin{array}{l}
I \wedge J \wedge K \wedge etat_1 = hs \Rightarrow \\
\quad etat_2 = false \wedge \\
\quad true \in BOOL \wedge \\
\quad Av, false, true \in BOOL \times BOOL \times BOOL \wedge \\
\quad true, false, Br \in BOOL \times BOOL \times BOOL \wedge \\
\quad (true = true \Leftrightarrow es = es) \wedge \\
\quad (rouge = vert \Leftrightarrow Av = true \ \wedge\ false = false \ \wedge\ true = false) \wedge \\
\quad (rouge = orange \Leftrightarrow Av = false \ \wedge\ false = true \ \wedge\ true = false) \wedge \\
\quad (rouge = rouge \Leftrightarrow Av = false \ \wedge\ false = false \ \wedge\ true = true) \wedge \\
\quad (vert = vert \Leftrightarrow true = true \ \wedge\ false = false \ \wedge\ Br = false) \wedge \\
\quad (vert = orange \Leftrightarrow true = false \ \wedge\ false = true \ \wedge\ Br = false) \wedge \\
\quad (vert = rouge \Leftrightarrow true = false \ \wedge\ false = false \ \wedge\ Br = true)
\end{array}$$

Seules deux formules méritent ici un peu d'attention

$$rouge = rouge \Leftrightarrow Av = false \ \wedge\ false = false \ \wedge\ true = true$$



et

$$vert = vert \Leftrightarrow true = true \ \wedge\ false = false \ \wedge\ Br = false$$

Pour en montrer la validité, il faut obtenir que $Av = false$ et que $Br = false$ à partir des hypothèses $I$, $J$, $K$, $etat = hs$ et $etat_1 = hs$. Le raisonnement est le suivant: de $J$ et de $etat_1 = hs$, on tire $etat = hs$; de $I$ et $etat = hs$, on tire que $feuA$ et $feuB$ valent $orange$; en utilisant les invariants de liaison $J$ et $K$, on obtient les deux égalités recherchées.

**Changement de couleur** Bien qu'un peu fastidieuse, il est intéressant d'étudier l'obligation de preuve du raffinement de l'opération de changement de couleur pour voir comment se distribuent les alternatives des substituions à choix borné.

Nous reprenons la décomposition de la substitution définissant l'opération de changement de couleur de $CARREFOUR1$ et introduisons une décomposition analogue pour celle de $CARREFOUR2$:

$$U_2 \mathrel{\widehat{=}} \text{PRE } \varphi_3 \text{ THEN CHOICE } U_1' \text{ OR } U_2' \text{ OR } U_3' \text{ OR } U_4'$$

Avec, pour chaque $i \in [1..4]$,

$$U_i' \mathrel{\widehat{=}} \text{WHEN } \delta_i \text{ THEN } U_i''$$

Le calcul de $[U_2]\neg[T_2]\neg K$ donne, en simplifiant les négations, la conjonction suivante:

$$\begin{array}{l}
(\delta_1 \Rightarrow ((\gamma_1 \wedge [U_1''][T_1'']K) \vee (\gamma_2 \wedge [U_1''][T_2'']K) \vee (\gamma_3 \wedge [U_1''][T_3'']K) \vee (\gamma_4 \wedge [U_1''][T_4'']K))) \wedge \\
(\delta_2 \Rightarrow ((\gamma_1 \wedge [U_2''][T_1'']K) \vee (\gamma_2 \wedge [U_2''][T_2'']K) \vee (\gamma_3 \wedge [U_2''][T_3'']K) \vee (\gamma_4 \wedge [U_2''][T_4'']K))) \wedge \\
(\delta_3 \Rightarrow ((\gamma_1 \wedge [U_3''][T_1'']K) \vee (\gamma_2 \wedge [U_3''][T_2'']K) \vee (\gamma_3 \wedge [U_3''][T_3'']K) \vee (\gamma_4 \wedge [U_3''][T_4'']K))) \wedge \\
(\delta_4 \Rightarrow ((\gamma_1 \wedge [U_4''][T_1'']K) \vee (\gamma_2 \wedge [U_4''][T_2'']K) \vee (\gamma_3 \wedge [U_4''][T_3'']K) \vee (\gamma_4 \wedge [U_4''][T_4'']K))) \wedge
\end{array}$$

Notez comment, dans chaque membre de cette conjonction, la négation a introduit une disjonction en place de la conjonction que donnait l'application de $T_2$.

Que le lecteur se rassure, nous n'allons pas traiter exhaustivement cette obligation de preuve¿Nous nous contenterons de décrire comment, sous les hypothèses $I$, $J$, $K$, $etat = es$ et $etat_1 = es$, on obtient le premier membre de cette conjonction. Les autres se traitent de façon similaire.

Une analyse rapide de la forme obtenue pour $[U_2]\neg[T_2]\neg K$ nous porte à penser que pour chaque $\delta_i$, un seul des membres de la disjonction correspondante doit être pertinent: celui de la forme $\gamma_i \wedge [U_i''][T_i'']K$. C'est ce que nous allons établir pour $i = 1$; les autres cas se traiteraient de façon similaire..

Supposons donc $\delta_1$ (ie $Av = true$) et montrons que $\gamma_1$ (ie $feuA1 = vert$) ainsi que $[U_1''][T_1'']K$.
On obtient $feuA1 = vert$ de $Av = true$ et $K$ qui dit que le seul cas où $Av$ a la valeur $true$, c'est quand $feuA1$ a la valeur $vert$.
L'application $[U_1''][T_1'']K$ se développe en

$$\begin{array}{l}
etat_2 \in BOOL \wedge \\
false, true, Ar \in BOOL \times BOOL \times BOOL \wedge \\
Bv, Bo, Br \in BOOL \times BOOL \times BOOL \wedge \\
(etat_2 = true \Leftrightarrow etat_1 = es) \wedge \\
(orange = vert \Leftrightarrow false = true \ \wedge\ true = false \ \wedge\ Ar = false) \wedge \\
(orange = orange \Leftrightarrow false = false \ \wedge\ true = true \ \wedge\ Ar = false) \wedge \\
(orange = rouge \Leftrightarrow false = false \ \wedge\ true = false \ \wedge\ Ar = true) \wedge \\
(feuB_1 = vert \Leftrightarrow Bv = true \ \wedge\ Bo = false \ \wedge\ Br = false) \wedge \\
(feuB_1 = orange \Leftrightarrow Bv = false \ \wedge\ Bo = true \ \wedge\ Br = false) \wedge \\
(feuB_1 = rouge \Leftrightarrow Bv = false \ \wedge\ Bo = false \ \wedge\ Br = true)
\end{array}$$

La plus par des élément de cette conjonction sont données par hypothèse ou directement en utilisant que faux est équivalent à faux. Le seul cas non trivial est celui de l'équivalence

$$orange = orange \Leftrightarrow false = false \ \wedge\ true = true \ \wedge\ Ar = false$$

ou il faut obtenir $Ar = false$. Ce que l'on obtient en utilisant le fait (démontré ci-dessus) que $Ar = false$ implique que $feuA1 = vert$. On utilise alors ce résultat et $K$ pour obtenir $Ar = false$.

## L'implantation

L'ultime étape de raffinement reste dans le formalisme des machines abstraites, mais elle est soumise à un certain nombre de restrictions. Nous ne suivrons pas ici complètement *la lettre* de B, mais plutôt son esprit en nous contentant de les principales de ces restrictions :

- les opérations ne peuvent être définies qu'en utilisant un jeu restreint de substitutions correspondant en fait à des instructions d'un langage de programmation impératif ;
- tous les ensembles utilisés doivent correspondre à des représentation concrètes de valeurs.

### Substitutions exécutables

**Affectation** L'affectation est tout simplement la substitution simple $x := E$ en restreignant l'expression $E$ à n'utiliser que des opérateurs connus d'un langage de programmation.

**Séquence** La séquence des langages de programmations correspond à la composition des substitutions :

$$[S_1 ;\ S_2]\Psi \mathrel{\widehat{=}} [S_2][S_1]\Psi$$

**Conditionnelles** La conditionnelle des langages de programmation est définie comme combinaison de deux substitutions gardées et d'une substitution à choix borné :

$$\textsc{if}\ B\ \textsc{then}\ S_1\ \textsc{else}\ S_2 \mathrel{\widehat{=}} \textsc{choice}\ (\textsc{when}\ B\ \textsc{then}\ S_1)\ \textsc{or}\ (\textsc{when}\ \neg B\ \textsc{then}\ S_1)$$

La condition $B$ doit être exprimée en n'utilisant que les connecteurs propositionnels correspondant aux opérateurs booléens usuels.

**L'instruction vide** On a la toujours utile SKIPque l'on définit comme la substitution identitée qui remplace chaque variable par elle-même.

### L'ultime raffinement

En tant que machine particulière, le raffinement correspondant à une implantation est introduit par la clause IMPLEMENTATION en place de REFINEMENT.

IMPLEMENTATION
  $CARREFOUR\_SYSTEM$

REFINES
  $CARREFOUR2$

VARIABLES
  $a_1, a_2, a_3, b_1, b_2, b_3$

INVARIANT
  $a1 \in BOOL \land a2 \in BOOL \land a3 \in BOOL \land$
  $b1 \in BOOL \land b2 \in BOOL \land b3 \in BOOL \land$

  $a1 = Av \land a2 = Ao \land a3 = Ar \land$
  $b1 = Bv \land b2 = Bo \land b3 = Br$

INITIALIZATION



  $a1 := false;\ a2 := true;\ a3 := false;$
  $b1 := false;\ b2 := true;\ b3 := false$

OPERATIONS
  MiseEnService $\widehat{=}$
    $a2 := false;\ a3 := true;$
    $b1 := true;\ b2 := false$

  Changement $\widehat{=}$
    IF $a1 = true$ THEN
      $a1 := false;\ a2 := true$
    ELSE IF $a2 = true$ THEN
      $a2 := false;\ a3 := true;$
      $b1 := true;\ b3 := false$
    ELSE IF $b1 = true$ THEN
      $b1 := false;\ b2 := true$
    ELSE
      $a1 := true;\ a3 := false;$
      $b2 := false;\ b3 := true$

Notez que les préconditions ont disparues des définitions des opérations. En effet, une précondition n'est pas destinée à être vérifiée par le code du programme à chaque exécution, mais plutôt, *a priori* par l'utilisateur des opérations fournies par une machine. Une précondition est un pur élément de spécification.

### Obligations de preuves

L'invariant de l'implantation est trivial. Nous ne traiterons donc pas l'initialisation ni l'opération de mise en route.

L'obligation de preuve de l'opération de changement de couleurs va nous permettre de vérifier la cohérence de l'analyse de cas spécifiée dans l'implantation vis-à-vis de celle définie précédemment en terme de choix entre substitutions gardées. En particulier, nous allons pouvoir nous assurer de l'exhaustivité des choix proposés.

Appelons $V_2$ la substitution définissant l'opération de changement de couleur de l'implantation et réécrivonssous la sous la forme :

IF $B_1$ THEN$V_1'$
ELSE IF $B_2$ THEN $V_2'$
ELSE IF $B_3$ THEN $V_3'$
ELSE $V_4'$

Soit $L$ l'invariant de l'implantation, en reprenant $U_2$, la substitution définissant le changement de couleur dans $CARREFOUR2$, en développant $\neg[U_2]\neg L$, on obtient :

$$(\delta_1 \land [U_1'']L) \lor (\delta_2 \land [U_2'']L) \lor (\delta_3 \land [U_3'']L) \lor (\delta_4 \land [U_4'']L)$$

que nous écrirons plus simplement

$$\bigvee_{i=1}^{4} ((\delta_i \land [U_i'']L)$$

Comme nous l'avions remarqué pour les substitutions multiples, on peut vérifier que pour chacune des substitutions séquentielles $V_j'$ avec $j \in [1..4]$ l'application $[V_j'] \bigvee_{i=1}^{4} ((\delta_i \land [U_i'']L)$ donne une formule logiquement équivalente à $\bigvee_{i=1}^{4} ((\delta_i \land [V_j'][U_i'']L)$. Posons, pour simplifier les écritures

$$\Phi_j \mathrel{\widehat{=}} \bigvee_{i=1}^{4} ((\delta_i \land [V_j'][U_i'']L)$$

Tenant compte de cette équivalence, le développement $[V_2]\neg[[U_2]\neg L$ est lui-même équivalent à la formule :

$$(B_1 \Rightarrow \Phi_1) \wedge (\neg B_1 \Rightarrow ((B_2 \Rightarrow \Phi_2) \wedge (\neg B_2 \Rightarrow ((B_3 \Rightarrow \Phi_3) \wedge (\neg B_3 \Rightarrow \Phi_4)))))$$

L'implication se distribuant sur la conjonction, et la proposition $A \Rightarrow (B \Rightarrow C)$ étant équivalente à $(A \wedge B) \Rightarrow C$, notre obligation de preuve est elle-même équivalente à la conjonction

$$\begin{array}{l} (B_1 \Rightarrow \Phi_1) \wedge \\ (\neg B_1 \wedge B_2 \Rightarrow \Phi_2) \wedge \\ (\neg B_1 \wedge \neg B_2 \wedge B_3 \Rightarrow \Phi_3) \wedge \\ (\neg B_1 \wedge \neg B_2 \wedge \neg B_3 \Rightarrow \Phi_4) \end{array}$$

Les différents cas de cette obligation de preuve se traitent selon la technique appliquée lors de la preuve de correction du raffinement $CARREFOUR2$ : on vérifie le membre « pertinent » de la disjonction ; celui ou les indices $j$ et $i$ coïncident.
La nouveauté ici est que les tests contiennent plus d'implicite que les gardes. Traitons le dernier cas, qui est celui où l'implicite culmine.

Rappelons nous que nous travaillons sous l'hypothèse que les invariants $I$, $J$ et $K$ sont satisfaits et que le système est en service. Nous allons, dans un premier déduire de $\neg B_1 \wedge \neg B_2 \wedge \neg B_3$ la couleur des deux feux. Intuitivement : puisque chaque feu doit avoir une couleur, si ni $a1$ ni $a2$ sont à vrai alors nécessairement $a3 = true$. C'est-à-dire que le premier feu ($feuA$) est au rouge. Le second ($feuB$) est donc soit $vert$ soit $orange$. Comme $b1$ (la lampe verte) n'est pas à vrai, c'est que le second feu est à l'orange.
De ce résultat, il est facile de déduire que $\delta_4$ et $[V_4'][U_4'']L$ sont valides. Ce qui nous donne $\Phi_4$.

### Quelques mots pour finir

Le travail réalisé sur ce petit exemple peut sembler bien démesuré en regard du résultat atteint : trois petites fonctions (l'initialisation et les deux opérations) dont l'intuition nous était venue dés l'énoncé du problème et que nous aurions donc pu tout aussi bien écrire directement.

À cette objection, il convient de faire deux réponses :

– le problème étudié ici est un *exemple* et comme tout exemple, sa valeur est dans sa simplicité. Mais la vie réelle a rarement cette simplicité et alors le travail d'élaboration de la spécification qui permet une première formulation triviale des opération (substitution indéterminée) puis la décomposition contrôlée par les obligations de preuves de chacune des étapes de raffinement prennent tout leur sens. Même sur ce petit exemple, on peut voir comment la méthode permet d'obtenir une formulation assez concise de l'opération de changement de couleur qui maîtrise un compléxité non négligeable : les $2^6$ combinaisons booléennes *a priori* possibles des variables du système.

– le système simple développé ici peut n'être qu'un composant d'un plus vaste système et la défaillance de notre système simple peut entraîner la défaillance du système plus vaste. Aussi est-il nécessaire de s'assurer de la fiabilité du composant. Mais nous avons ici obtenu plus. Non seulement, nous nous sommes assurés de la fiabilité des opérations de manipulations des feux, mais nous sommes également en mesure de fournir un *certificat de garantie* de cette fiabilité sous la forme des étapes successives du raffinement accompagnées chacune de la démonstration de leurs obligations de preuves.