Université Pierre & Marie Curie Maîtrise d'informatique

Programmation
Spécification et certification du logiciel

Notes de cours

P. Manoury

2001-02

1 Spécifications algébriques

Tout programmeur sait ce qu'est une *spécification* et tout programmeur a manipulé des spécifications. Celles-ci sont, en géneral, réduites à la simple expression de nom de fonctions accompagnés de leur type. Ces spécifications 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 préter 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. On ne retiendra pas l'opérateur `clear` qui a peu de sens dans le monde fonctionnel.

Sort: `stack`
Uses: `int, elt`
Symbols:
  `create :` $\rightarrow$ `stack`
  `push : elt, stack` $\rightarrow$ `stack`
  `top : stack` $\rightarrow$ `elt`
  `pop : stack` $\rightarrow$ `stack`
  `length : stack` $\rightarrow$ `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 le corps : les équations devant être satisfaites.

Axioms: $\forall$ `s:stack; x:elt.`
  $[A1]$ `(top (push x s)) = x`
  $[A2]$ `(pop (push x s)) = s`
  $[A3]$ `(length create) = 0`
  $[A4]$ `(length (push x s)) = 1+(length s)`

Les variables servant à exprimer les axiomes sont déclarées avec leur sorte derrière le mot-clé Axioms. Chacune des équations est implicitement universellement quantifiée (i.e. si l'on était moins fainéants, il faudrait répéter $\forall$ `s:stack; x:elt.` devant toutes les équations – sauf la troisième qui ne contient pas de variables).

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

– les opérateurs `create` et `push` jouent un rôle particulier concernant les piles : celui de *constructeurs*. Ce sont des primitives abstraites dont aucune équation ne définit la valeur (il n'y a pas d'équation de la forme `(push ...) = ...`). On a de plus 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 défini par des équations récursives.

Résumé

En rassemblant les éléments déclaratifs et descriptifs de la spécification des piles, on obtient le *module de spécification* complet suivant :

```
Sort: stack
Uses: int, elt
Symbols:
  create : → stack
  push : elt, stack → stack
  top : stack → elt
  pop : stack → stack
  length : stack → int
Axioms: ∀ s:stack; x:elt.
  [A1] (top (push x s)) = x
  [A2] (pop (push x s)) = s
  [A3] (length create) = 0
  [A4] (length (push x s)) = 1+(length s)
```

Langage des spécifications algébriques (1)

On donne sous forme de BNF une première grammaire des modules de spécification.

Caractères réservés : `:` `→` `,` `;` `(` `)` `∀` `[` `]`

Mots clef : Sort: Uses: Symbols: Axioms:

Autres unités lexicales : *ident num inop*

SpecMod	::=	Header DecPart AxPart
Header	::=	Sort: *ident*
	|	Sort: *ident* Uses: IdentList
DecPart	::=	Symbols: SigList
SigList	::=	*ident* : SymSort
	|	*ident* : SymSort SigList
SymSort	::=	→ BasicSort
	|	FunSort
BasicSort	::=	*ident*
FunSort	::=	ArgSortList → BasicSort
ArgSortList	::=	ArgSort
	|	ArgSort , ArgSortList
ArgSort	::=	BasicSort
	|	FunSort

AxPart	::=	Axioms: EqList
	|	Axioms: VarBinding EqList
VarBinding	::=	∀ VarDecList .
VarDecList	::=	VarDec
	|	VarDec ; VarDecList
VarDec	::=	VarList : VarSort
VarList	::=	*ident*
	|	*ident* , VarList
VarSort	::=	BasicSort
	|	FunSort
EqList	::=	Eq
	|	Eq EqList
Eq	::=	[*num*] Term = Term
Term	::=	*ident*
	|	Term Term
	|	(Term)
	|	Term *inop* Term

Outre leur correction syntaxique, les termes doivent être correctement typés (i.e. « *sortés* ») :

si σ est une sorte, un *terme de sorte* σ est soit un symbole seul de sorte $\to \sigma$, soit une application de la forme (s s_1 .. s_n) si s est de sorte $\sigma_1, .., \sigma_n \to \sigma$ et s_1 est de sorte σ_1, ... et s_n est de sorte σ_n, soit une application de la forme s_1 *inop* s_2 si *inop* est un symbole infixe de sorte $\sigma_1, \sigma_2 \to \sigma$.

Des listes

Voici la structure 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
  [A5] (length nil) = 0
  [A6] (length (cons x xs)) = 1+(length xs)
  [A7] (hd (cons x xs)) = x
  [A8] (tl (cons x xs)) = xs
```

[$A9$] `(append nil ys) = ys`
[$A10$] `(append (cons x xs) ys) = (cons x (append xs ys))`
[$A11$] `(rev nil) = nil`
[$A12$] `(rev (cons x xs)) = (append (rev xs) (cons x nil))`
[$A13$] `(rev_append nil ys) = ys`
[$A14$] `(rev_append (cons x xs) ys) = (rev_append xs (cons x ys))`

Theorems: ∀ `xs, ys, zs:list`
[$T1$] `(append xs nil) = xs`
[$T2$] `(length (append xs ys)) = (length xs) + (length ys)`
[$T3$] `(append (append xs ys) zs) = (append xs (append ys zs))`
[$T4$] `(rev_append xs ys) = (append (rev xs) ys)`
[$T5$] `(rev xs) = (rev_append xs nil)`

L'égalité $T5$ est intéressante du point de vue implantation puisqu'elle donne un moyen d'obtenir le miroir d'une liste en utilisant une récursion terminale.

La preuve de $T1$ est immédiate par induction sur la liste `xs` :

- si `xs = nil`, on a immédiatement `(append nil nil) = nil`;
- si `xs = (cons x xs')`, notre hypothèse de récurrence est que `(append xs' nil) = xs'` et il faut monter que

  `(append (cons x xs') nil) = (cons x xs')`

  on a:

`(append (cons x xs') nil)`	=	`(cons x (append xs') nil)`	par définition de `append`
	=	`(cons x xs')`	par hypothèse de récurence

La preuve de $T2$ est aussi 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 $T3$ qui donne l'associativité de l'opérateur de concaténation `append` est encore immédiate par induction sur `xs` (en exercice).

La preuve de $T4$ 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)`

  Il faut montrer, pour un `ys:list` quelconque, que

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

  C'est à dire, en utilisant les axiomes équationnels et l'associativité de `append` :

  `(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 $T5$ est une conséquence de $T4$ et $T1$

Équations conditionnelles

Une équation conditionnelles est une expression de la forme : `c1, .., cn` $\Rightarrow$ `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
  [A15] (nth (cons x xs) 0) = x
  [A16] (0 < i), (i ≤ (length xs)) ⇒ (nth (cons x xs) i) = (nth 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
  [A17] (mem x nil) = false
  [A18] (mem x (cons x xs)) = true
  [A19] (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) = false
  (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`
[$A20$] `(sorted nil) = true`
[$A21$] `(sorted (cons x nil)) = true`
[$A22$] `(x` $\leq$ `y)` $\Rightarrow$ `(sorted (cons x (cons y xs))) = (sorted (cons y xs))`
[$A23$] `(y < x)` $\Rightarrow$ `(sorted (cons x (cons y xs))) = false`

On énonce, sur les listes triées deux petits résultats qui serviront par la suite :

Extends: `list`
Theorems: ∀ `x,y:elt; ys:list.`
[$T6$] `(sorted (cons x (cons y ys)))` $\Rightarrow$ `(x` $\leq$ `y) = true`
[$T7$] `(sorted (cons y ys))` $\Rightarrow$ `(sorted ys) = true`

On a immédiatement 6. en remarquant que sa contraposée est exactement la quatrième clause de la définition de `sorted`.

Pour obtenir 7., on raisonne par cas sur `ys` :

- si `ys = nil`, on a, par définition, que `(sorted nil) = true`. Ce qui suffit à établir l'implication;
- si `ys = (cons z zs)`, on raisonne par l'absurde, en supposant que ($H1$) `(sorted (cons x (cons z zs))) = true` et que ($H2$) `(sorted (cons z zs)) = false`.
  De $H1$ et de 6., on obtient que `(x` $\leq$ `z) = true`. On peut alors utiliser la définition de `sorted` (troisième clause) pour tirer de $H1$ que `(sorted (cons z zs)) = true`, ce qui contredit $H2$.

Opérateur de tri. On utilise maintenant 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` $\rightarrow$ `list`
`ins_sort : list` $\rightarrow$ `list`
Axioms: ∀ `xs:list; x,y:elt`
[$A24$] `(ins x nil) = (cons x nil)`
[$A25$] `(x` $\leq$ `y)` $\Rightarrow$ `(ins x (cons y xs)) = (cons x (cons y xs))`
[$A26$] `(y < x)` $\Rightarrow$ `(ins x (cons y xs)) = (cons y (ins x xs))`
[$A27$] `(ins_sort nil) = nil`
[$A28$] `(ins_sort (cons x xs)) = (ins (ins_sort xs))`
Theorems: ∀ `xs:list`
[$T8$] `(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`
9. `(sorted xs)` $\Rightarrow$ `(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 9. :

Extends: `list`
Lemma: ∀ `xs:list; x,y:elt`
9.1 `(x` $\leq$ `y), (sorted (cons x xs))` $\Rightarrow$ `(sorted (cons x (ins y xs))) = true`

Preuve de 9.1: 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 pour tirer notre résultat de l'hypothèse de récurrence HR où `x` est instancié en `z`.

Langage des spécifications algébriques (2)

On résume les extensions au langage des spécifications.

SpecMod	::=	...
Header	::=	Title Using Assuming
Title	::=	Sort: *ident*
	|	Extends: *ident*
Using	::=	
	|	Uses: IdentList
Assuming	::=	
	|	Assume: SigList with VarBinding EqList
	⋮	
Eq	::=	[*num*] Term = Term
	|	[*num*] PremList ⇒ Term = Term
PremList	::=	Prem
	|	Prem , PremList
Prem	::=	Term
	|	Term = Term
	|	Term ≠ Term
	⋮	

Modélisation des vecteurs

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` $\rightarrow$ `int`
  `get : vect, int` $\rightarrow$ `elt`
  `set : vect, int, elt` $\rightarrow$ `vect`
  `make : int, elt` $\rightarrow$ `vect`
Axioms: $\forall$ `v:vect; e:elt; n,i,j:int`
  $[A29]$ `(length (make n e)) = n`
  $[A30]$ `(length (set v i e)) = (length v)`
  $[A31]$ `(0` $\leq$ `i), (i < n)` $\Rightarrow$ `(get (make n e) i) = e`
  $[A32]$ `(0` $\leq$ `i), (i < (length v))` $\Rightarrow$ `(get (set v i e) i) = e`
  $[A33]$ `(0` $\leq$ `i), (i < (length v)), (0` $\leq$ `j), (j < (length v)), (i` $\neq$ `j)`
     $\Rightarrow$ `(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
Uses: bool
Symbols:
  indom : int, vect → bool
Axioms: ∀ v:vect; i:int
  [A34] (0 ≤ i), (i < (length v)) ⇒ (indom i v) = true
  [A35] (indom i v) ⇒ (0 ≤ i) = true
  [A36] (indom i v) ⇒ (i < (length v)) = true
```

Ces trois équations signifient que `(indom i v)` est égal à la conjonction (booléenne) de `(0 ≤ i)` et `(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) = false
  (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
  /* DÉFINTION DE IMAX */
  [A37] (indom (imax v) v) = true
  [A38] (indom i v) ⇒ (get v i) ≤ (get v (imax v))
  [A39] (indom i v), ((get v (imax v)) = (get v i)) ⇒ (imax v) ≤ i
  /* FONCTION DE CALCUL: FIND_MAX */
  [A40] (loop m (length v) v) = m
  [A41] (0 ≤ i), (i <(length v)), (indom m v), (get(v,m) <get(v,i))
        ⇒ (loop m i v) = (loop i i+1 v)
  [A42] (0 ≤ i), (i <(length v)), (indom m v), (get(v,i) ≤ get(v,m))
        ⇒ (loop m i v) = (loop m i+1 v)
  [A43] ((length v)≠ 0) ⇒ (find_max v) = (loop 0 1 v)
Theorems: ∀ v:vect
  [T9] ((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` :

```
∀ 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
  [A44] (0 < i) ⇒ (indom (bimax i v) v) = true
  [A45] (0 ≤ j), (j < i) ⇒ (get v j) ≤ (get v (bimax i v))
  [A46] (0 ≤ j), (j < i), ((get v (bimax i v)) = (get v j))
        ⇒ (bimax i v) ≤ j
Lemma: ∀ v:vect; i,m:int
  [T10] ((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)` $\leq$ `(get v (bimax i v))` ou non.
  - si `(get v i)` $\leq$ `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)
    ```

Préfixe d'une liste

Définition récursive

Extends: `list`
Symbols:
  `pref : list, list` $\rightarrow$ `bool`
Axioms: $\forall$ `x, y:elt; xs1, xs2, xs: list.`
  [$A47$] `(pref nil xs) = true`
  [$A48$] `(pref (cons x xs1) (cons x xs)) = (pref xs1 xs)`
  [$A49$] `(x` $\neq$ `y)` $\Rightarrow$ `(pref (cons x xs1) (cons y xs)) = false`
  [$A50$] `(pref (cons x xs1) nil) = false`
Theorems: $\forall$ `xs1, xs2: list.`
  [$T11$] `1 (pref xs1 (append xs1 xs2)) = true`

Preuve de [$T1$] par induction sur `xs1`

- si `xs1 = nil`, immédiat par [$A1$].
- si `xs1 = (cons x xs1')`, HR: `(pref xs1' (append xs1' xs2)) = true`.
  On veut `(pref (cons x xs1') (append (cons x xs1') xs2)) = true`. C'est à dire, par [$A2$] et déf. append, `(pref xs1' (append xs1' xs2)) = true`.

Préfixe et concaténation

Extends: `list`
Theorems: $\forall$ `xs1, xs: list.`
  [$T2$] `(pref xs1 xs)` $\Rightarrow$ $\exists$ `xs2:list. xs = (append xs1 xs2)`

Preuve de [$T2$] on montre $\forall$ `xs: list. (pref xs1 xs)` $\Rightarrow$ $\exists$ `xs2:list. xs = (append xs1 xs2)` par induction sur `xs1`

- si `xs1 = nil`, on prend `xs2 = xs`.
- si `xs1 = (cons x xs1')`,
  HR: $\forall$ `xs: list. (pref xs1' xs)` $\Rightarrow$ $\exists$ `xs2:list. xs = (append xs1' xs2)`
  On montre `(pref (cons x xs1') xs)` $\Rightarrow$ $\exists$ `xs2:list. xs = (append (cons x xs1') xs2)` par cas sur `xs`.
  - si `xs = nil`, on veut `(pref (cons x xs1') nil)` $\Rightarrow$ $\exists$ `xs2:list. nil = (append (cons x xs1') xs2)`. Trivial, sachant [$A4$].
  - si `xs = (cons y xs')`, on suppose H1: `(pref (cons x xs1') (cons y xs')) = true` et on veut $\exists$ `xs2:list. (cons y xs') = (append (cons x xs1') xs2)`.
    Par H1 et [$A3$] (contrap), on a `x = y`. D'où, par H1 et [$A2$], `(pref xs1' xs') = true`. D'où, par HR, il existe `xs2` tel que `xs' = (append xs1' xs2)`.
    C'est à dire, `xs = (cons x xs') = (append (cons x xs1') xs2)`.

Calcul d'un préfixe de longueur donnée

Extends: `list`
Symbols:
  `n_pref : nat, list` $\rightarrow$ `list`
Axioms: $\forall$ `n:nat; x:elt; xs:list.`
  [$A5$] `(n_pref 0 xs) = nil`
  [$A6$] `(n > 0)` $\Rightarrow$ `(n_pref n (cons x xs)) = (cons x (n_pref (n-1) xs))`

[$A7$] `(n_pref n nil) = nil`
Theorems: ∀ `n:nat; xs:list.`
[$T3$] `(pref (n_pref n xs) xs) = true`

Preuve de [$T3$] on montre ∀ `xs:list. (pref (n_pref n xs) xs) = true` par induction sur `n`

- si `n = 0`, immédiat par [$A5$] et [$A1$].
- si `n = n'+1`, HR: ∀ `xs:list. (pref (n_pref n' xs) xs) = true`.
  On montre ∀ `xs:list. (pref (n_pref (n'+1) xs) xs) = true` par cas sur `xs`
  - si `xs = nil`, immédiat par [$A7$] et [$A1$].
  - si `xs = (cons x xs')`, on veut `(pref (n_pref (n'+1) (cons x xs)) (cons x xs)) = true`.
    C'est-à-dire, par [$A6$], `(pref (cons x (n_pref n' xs') (cons x xs')) = true`.
    Immédiat par [$A2$] et HR.

Tri par arbre binaire de recherche

Les arbres binaires

Sort: `btree`
Uses: `elt`
Symbols:
`Lf :` → `btree`
`Br : btree, elt, btree` → `btree`
`root : btree` → `elt`
Axioms: ∀ `x:elt; a1, a2:btree.`
[$A51$] `(root (Br a1 x a2)) = x`

Les arbres binaires de recherche

Extends: `btree`
Uses: `elt`
Assumes:
`[ORDERED elt]`
Symbols:
`verify : (elt, elt` → `bool), elt, btree` → `bool`
`maximize : elt, btree` → `bool`
`minimize : elt, btree` → `bool`
`abr : btree` → `bool`
Axioms: ∀ `r:elt, elt` → `bool; x, y:elt; a1, a2:btree.`
[$A52$] `(verify r x Lf) = true`
[$A53$] `(verify r x a1), (verify r x a2), (r x y)` ⇒ `(verify r x (Br y a1 a2)`
[$A54$] `minimize = (verify` ≤`)`
[$A55$] `maximize = (verify` ≥`)`
[$A56$] `(abr Lf) = true`
[$A57$] `(abr a1), (abr a2), (maximize x a1), (minimize x a2)` ⇒ `(abr (Br a1 x a2))`

Le tri

Extends: `list`
Uses: `elt, abr`
Symbols:
`ins : elt, btree` → `btree`

```
to-abr : list → btree
to-list : btree → list
to-list-bis : btree → list
abr-sort : list → list
```
Axioms: ∀ x, y:elt; xs:list; a, a1, a2, a3:btree.
```
[A58] (ins x a) = (Br Lf x Lf)
[A59] (x ≤ y) ⇒ (ins x (Br a1 y a2)) = (Br (ins x a1) y a2)
[A60] (x ≥ y) ⇒ (ins x (Br a1 y a2)) = (Br a1 y (ins x a2))
[A61] (to-abr nil) = Lf
[A62] (to-abr (cons x xs)) = (ins x (to-abr xs))
[A63] (to-list Lf) = nil
[A64] (to-list (Br a1 x a2)) = (append (to-list a1) (cons x (to-list a2)))
[A65] (to-list-bis Lf) = nil
[A66] (to-list-bis (Br Lf x a2)) = (cons x (to-list-bis a2))
[A67] (to-list-bis (Br (Br a1 x a2) y a3)) = (to-list-bis (Br a1 x (Br a2 y a3)))
```
Theorems ∀ x:elt; xs:list; a:btree.
```
(sorted (abr-sort xs)) = true
(abr (to-abr xs)) = true
(abr a) ⇒ (abr (ins x a))
(abr a) ⇒ (sorted (to-list a))
```

Graphes 1

Les couples

Sort: pair
Uses: elt1, elt2
Symbols:
```
mkpair : elt1, elt2 → pair
fst : pair → elt1
snd : pair → elt2
```
Axioms: ∀ x1:elt1; x2:elt2.
```
[A68] (fst (mkpair x1 x2)) = x1
[A69] (snd (mkpair x1 x2)) = x2
```
Theorems: ∀ x:pair.
```
(mkpair (fst x) (snd x)) = x
```

Listes d'associations

Extends: list
Uses: pair
Symbols:
```
assoc : elt1, list → elt2
add-assoc : elt1, elt2, list → list
```
Axioms: ∀ x, x':elt1; y, y':elt2; ys, xys:list.
```
[A70] (assoc x (cons (mkpair x y) xys)) = y
[A71] (x ≠ x') ⇒ (assoc x (cons (mkpair x' y) xys)) = (assoc x xys)
[A72] (add-assoc x y nil) = (cons (mkpair x y) nil)
[A73] (add-assoc x y (cons (mkpair x ys) xys)) = (cons (mkpair x (cons y ys)) xys)
[A74] (x ≠ x') ⇒ (add-assoc x y (cons (mkpair x' ys) xys)) = (cons (mkpair x'
ys) (add-assoc x y xys))
```

Theorems: ∀ `x:elt1; y:alt2; xys:list.`
`(mem y (assoc x (add-assoc x y xys))) = true`

Liste d'adjacences

Où l'on voit la faiblesse du typage avec paramètres implicites.

Extends: `list`
Uses: `pair`
Symbols:
`to-adj : list ->list`
Axioms: ∀
$[A75]$ `(to-adj nil) = nil`
$[A76]$ `(to-adj (cons (mkpair x y) xys)) = (add-assoc x y (to-adj xys))`
Theorems: ∀ `x:elt1; y:elt2; g:list.`
`(mem (mkpair x y) g)` ⇒ `(mem y (assoc x (to-adj g))) = true`
`(mem y (assoc x (to-adj g)))` ⇒ `(mem (mkpair x y) g) = true`

2 Preuve de programmes impératifs

Assertions

Nous allons à présent utiliser la spécification de l'opérateur `imax` (et de son alter-ego `bimax`) pour établir la correction d'un programme impératif calculant l'indice de l'élément maximal d'un tableau.

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*. Une 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`.

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	$\mathtt{SKIP}$
Affectation	$x\ \mathtt{:=}\ e$
Séquence	$C_1\ \mathtt{;}\ C_2$
Conditionnelle	$\mathtt{If}\ e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$
Boucle	$\mathtt{While}\ e\ \mathtt{do}\ C$

On présente les définitions sous forme de règles de déduction :

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

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

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

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

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

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 :

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

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

`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 $[P]\ C_0$`;` `While` e `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 de renforcement 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 $\texttt{i} \geq \texttt{(length v)}$ alors $\texttt{(bimax i v)} = \texttt{(imax v)}$. D'où

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

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) :

$[\,\texttt{m} = \texttt{(bimax i v)} \wedge \texttt{i} < \texttt{(length v)}\,]$
`If v(m) < v(i) then m := i else SKIP;i := i+1`
$[\,\texttt{m} = \texttt{(bimax i v)}\,]$

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

$[\,\texttt{m} = \texttt{(bimax i v)} \wedge \texttt{i} < \texttt{(length v)}\,]$
`If v(m) < v(i) then m := i else SKIP;`
$[\,\texttt{m} = \texttt{(bimax i+1 v)}\,]$
`i := i+1`
$[\,\texttt{m} = \texttt{(bimax i v)}\,]$

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

$[\,\texttt{m} = \texttt{(bimax i v)} \wedge \texttt{i} < \texttt{(length v)}\,]$
`If v(m) < v(i) then m := i else SKIP;`
$[\,\texttt{m} = \texttt{(bimax i+1 v)}\,]$

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

$[\,\texttt{m} = \texttt{(bimax i v)} \wedge \texttt{i} < \texttt{(length v)} \wedge \texttt{v(m)} < \texttt{v(i)}]$
`m := i`
$[\,\texttt{m} = \texttt{(bimax i+1 v)}\,]$

et que (b)

$[\,\texttt{m} = \texttt{(bimax i v)} \wedge \texttt{i} < \texttt{(length v)} \wedge \texttt{v(m)} \geq \texttt{v(i)}]$
`SKIP;`
$[\,\texttt{m} = \texttt{(bimax i+1 v)}\,]$

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

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

On obtient le résulat attendu par renforcement de la précondition.

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

CQFD

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 `If` B `then` S_1 `else` S_2 aussi.
- si B est une formule et S une spécification alors `While` B `do` S est une spécification.
- si S est une spécification et x est une variable alors `Var` $x.S$ est une spécification.
- si x est une variable et e une expression alors x `:=` e est une spécification.
- `SKIP` est une spécification.

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

$$S_1 \supseteq S_2$$

Les combinateurs devront être monotones par rapport à la relation de raffinement : si $S_1 \supseteq S_1'$ et $S_2 \supseteq S_2'$ alors

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

La monotonie 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)

$$[P, Q] \supseteq [P, R] \texttt{ ; } [R, Q]$$

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

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

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

Précondition (St)

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

Postcondition (Wk)

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

L'application des règles de renforcement de la précondition et de affaiblissement 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) Lorsque la précondition implique la postcondition modulo une substitution, on obtient, par renforcement une deuxième règle pour l'affectation :

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

En effet

$$
\begin{array}{rcll}
[P, Q] & \supseteq & [Q[e/x], Q] & (St) \\
& \supseteq & x\ \mathtt{:=}\ e & (As)
\end{array}
$$

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

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

En effet :

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

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 $Y > 0$.

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

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

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

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 :

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

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 $C_1\ ;\ C_2 \in [P, R]\ ;\ [R, Q]$, 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 $\vdash [P]\ C_1\ ;\ C_2\ [Q]$.

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 définition. On a donc $\vdash [P]\ \texttt{Var}\ x.C\ [Q]$, puisque x n'apparaît 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 $\vdash [P]\ \texttt{While}\ B\ \texttt{do}\ C\ [P \wedge \neg B]$.

Miroir d'une liste en impératif

On utilise la spécification des listes donnée page ??.

On donne le code impératif avec assertions d'un programme calculant la fonction `rev`. Les variables sont `xs, ys:list` et L est une constante donnant la valeur initiale de `xs`.

```
(* xs = L *)
ys := nil;
(* (append (rev xs) ys) = (rev L) *)
While xs ≠ nil do
  (* xs ≠ nil ∧ (append (rev xs) ys) = (rev L) *)
  ys := (cons (hd xs) ys);
  (* xs ≠ nil ∧ (append (rev (tl xs)) ys) = (rev L) *)
  xs := (tl xs)
  (* (append (rev xs) ys) = (rev L) *)
done.
(* ys = (rev L) *)
```

Preuve par Hoare-Floyd

On veut

```
[ xs = L ]
ys := nil;
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  xs := (tl xs)
done.
[ ys = (rev L) ]
```

On a

```
xs = L
⇒ (rev xs) = (rev L)
⇒ (append (rev xs) nil) = (rev L) (par [T1] )
```

On veut donc (par aff. précondition)

```
[ (append (rev xs) nil) = (rev L) ]
ys := nil;
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  xs := (tl xs)
done.
[ ys = (rev L) ]
```

On a

```
[ (append (rev xs) nil) = (rev L) ]
ys := nil
[ (append (rev xs) ys) = (rev L) ]
```

Reste donc à montrer

```
[ (append (rev xs) ys) = (rev L) ]
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  xs := (tl xs)
```

```
done.
[ ys = (rev L) ]
```

On a

```
xs = nil ∧ (append (rev xs) ys) = (rev L)
⇒ ys = (rev L)
```

Il faut donc montrer

```
[ (append (rev xs) ys) = (rev L) ]
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  xs := (tl xs)
done.
[ xs = nil ∧ (append (rev xs) ys) = (rev L) ]
```

C'est-à-dire (règle du `While`)

```
[ xs ≠ nil ∧ (append (rev xs) ys) = (rev L) ]
ys := (cons (hd xs) ys);
xs := (tl xs)
[ (append (rev xs) ys) = (rev L) ]
```

Or, on a

- `xs ≠ nil ⇒ xs = (cons (hd xs) (tl xs))` (vient de ce que toute liste est soit `nil` soit `cons`);
- `(append (rev (cons (hd xs) (tl xs))) ys) = (append (tl xs) (cons (hd xs) ys))` (facile à vérifier).

On a donc que

```
xs ≠ nil ∧ (append (rev xs) ys) = (rev L)
⇒ xs ≠ nil ∧ (append (tl xs) (cons (hd xs) ys)) = (rev L)
```

Par aff. précodition, il nous faut donc montrer

```
[ xs ≠ nil ∧ (append (tl xs) (cons (hd xs) ys)) = (rev L) ]
ys := (cons (hd xs) ys);
xs := (tl xs)
[ (append (rev xs) ys) = (rev L) ]
```

On a, bien entendu

```
[ xs ≠ nil ∧ (append (tl xs) (cons (hd xs) ys)) = (rev L) ]
ys := (cons (hd xs) ys)
[ xs ≠ nil ∧ (append (tl xs) ys) = (rev L) ]
```

Reste donc à voir que

```
[ xs ≠ nil ∧ (append (tl xs) ys) = (rev L) ]
xs := (tl xs)
[ (append (rev xs) ys) = (rev L) ]
```

Ce qui est tout vu en oubliant le `xs ≠ nil` de la précondition.

Raffinement

La spécification originale de notre programme est le couple de précondition et postcondition :

```
[ xs = L, ys = (rev L) ]
```

Par aff. précondition (cf OP1):

```
[ (rev xs) = (rev L), ys = (rev L) ]
```

puis (cf OP2)

```
[ (append (rev xs) nil) = (rev L), ys = (rev L) ]
```

Par séquence:

```
[ (append (rev xs) nil) = (rev L), (append (rev xs) ys) = (rev L) ] ;
[ (append (rev xs) ys) = (rev L), ys = (rev L) ]
```

Par affectation:

```
ys := nil;
[ (append (rev xs) ys) = (rev L), ys = (rev L) ]
```

Par renf. postcondition (cf OP3):

```
ys := nil;
[ (append (rev xs) ys) = (rev L), xs = nil ∧ (append (rev xs) ys) = (rev L) ]
```

Par boucle:

```
ys := nil;
While xs ≠ nil do
  [ xs ≠ nil ∧ (append (rev xs) ys) = (rev L), (append (rev xs) ys) = (rev L) ]
done.
```

Par aff. précond (cf OP4):

```
ys := nil;
While xs ≠ nil do
  [ xs ≠ nil ∧ (append (rev (tl xs)) (cons (hd xs) ys)) = (rev L),
    (append (rev xs) ys) = (rev L) ]
done.
```

Par séquence et affectation:

```
ys := nil;
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  [ xs ≠ nil ∧ (append (rev (tl xs)) ys) = (rev L),
    (append (rev xs) ys) = (rev L) ]
done.
```

Par séquence (et aff. précond.)

```
ys := nil;
While xs ≠ nil do
  ys := (cons (hd xs) ys);
  xs := (tl xs)
done.
```

Résumé des obligations de preuves

OP1: $\texttt{xs} = L \Rightarrow \texttt{(rev xs)} = \texttt{(rev } L\texttt{)}$
OP2: $\texttt{(rev xs)} = \texttt{(rev } L\texttt{)} \Rightarrow \texttt{(rev xs)} = \texttt{(append (rev xs) nil)}$
OP3: $\texttt{xs} = \texttt{nil} \wedge \texttt{(append (rev xs) ys)} = \texttt{(rev } L\texttt{)} \Rightarrow \texttt{ys} = \texttt{(rev } L\texttt{)}$
OP4: $\texttt{xs} \neq \texttt{nil} \wedge \texttt{(append (rev xs) ys)} = \texttt{(rev } L\texttt{)}$
$\quad \Rightarrow \texttt{(append (rev (tl xs)) (cons (hd xs) ys))} = \texttt{(rev } L\texttt{)}$

3 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 minimalistes de la théorie des ensembles. Néanmoins, nous pourrons nous contenter ici d'une présentation intuitive, une *théorie naïve*.

Langage ensembliste

Le langage ensembliste est à la fois riche et extrèment pauvre. Il est pauvre car on peut se contenter, pour développer la théorie des ensembles, du langage de la logique pure (connecteurs, quantificateurs) et du seul symbole $\in$. Il est riche car sur cette seule base, on peut introduire toutes les notions nécessaires aux mathématiques avec leur notation.

Relations entre ensembles

Appartenance 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* ». En théorie des ensembles, la notion d'*élément* n'a pas d'existence en soi : tout est ensemble et on est toujours élément de quelque chose.

Comme elle est d'un usage fréquent, on se donne la notation $\notin$ comme une *abréviation* servant à désigner la négation de la relation d'appartenance. On pose :

$$x \notin y \mathrel{\widehat{=}} \neg(x \in y)$$

où $\widehat{=}$ représente l'égalité *par définition*.

Dans le cadre de la théorie des ensembles utilise souvent les tournures : *pour tous x appartenant à y* ; encore de *il existe x appartenant à y tel que* ... Ces deux locutions donnent lieu aux deux abréviations suivantes :

$$\begin{array}{rcl} \forall x \in y.\varphi & \widehat{=} & \forall x.(x \in y \Rightarrow \varphi) \\ \exists x \in y.\varphi & \widehat{=} & \exists x.(x \in y \wedge \varphi) \end{array}$$

où *phi* est une formule quelconque.

Inclusion Sur la base de la relation d'appartenance, on *définit* la relation d'inclusion entre x et y :

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

On dit que *x est sous-ensemble de y*.

Égalité Attention, il ne faut pas confondre cette égalité par définition avec l'égalité comme relation entre ensemble. Cette dernière se note avec le symbole usuel $=$ dont on contraint l'usage par l'*axiome* :

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

Et d'autres termes, deux ensembles sont égaux si et seulement si ils possèdent exactement les mêmes éléments. Cet axiome est appelé *axiome d'extensionalité*.

Notez que l'on a le *théorème* :

$$x = y \Leftrightarrow (x \subseteq y \wedge y \subseteq x)$$

Constructeurs d'ensembles

On introduit maintenant comment obtenir de nouveaux ensembles par combinaison. Pour chacune de ces constructions, on introduira une notation ainsi que la contrainte axiomatique en régissant l'usage. Dans la plupart des cas, il s'agira d'une contrainte portant sur les conditions d'appartenance à l'ensemble construit.

Ensemble vide On note $\emptyset$ l'ensemble vide. On pose :

$$\forall x.x \notin \emptyset$$

Union On note $x \cup y$ l'union des éléments de x et y. On pose :

$$\forall z.(z \in x \cup y \Leftrightarrow (z \in x \vee z \in y))$$

Notez que l'on a (théorèmes)

$$(x \cup \emptyset) = (\emptyset \cup x) = x$$

Intersection On note $x \cap y$ l'instersection des éléments de x et de y. On pose :

$$\forall z.(z \in x \cap y \Leftrightarrow (z \in x \wedge z \in y))$$

Notez que l'on a (théorèmes)

$$(x \cap \emptyset) = (\emptyset \cap x) = \emptyset$$

Différence ensembliste On note $x \backslash y$ l'ensemble x privé des éléments de y. On pose :

$$\forall z.(z \in x \backslash y \Leftrightarrow z \in x \wedge z \notin y)$$

Montrez que l'on a

$$\begin{array}{l} x \backslash x = \emptyset \\ x \backslash (x \cap y) = x \backslash y \\ x \subseteq y \Rightarrow x \backslash y = \emptyset \end{array}$$

Ensemble des parties On note $I\!P(x)$ l'ensemble de tous les sous-ensembles de x. On pose :

$$\forall z.(z \in I\!Px \Leftrightarrow z \subseteq x)$$

Couples et produit cartésien

À partir de deux ensembles x et y, on forme le couple (ou *paire ordonnée*) de ces deux ensemble que l'on note (x, y). Un couple est caractérisé par l'égalité plutôt que par l'appartenance comme cela a été le cas pour les constructeurs précédents. Si x, y, x' et y' sont des ensembles, on pose donc :

$$(x, y) = (x', y') \Leftrightarrow x = x' \wedge y = y'$$

Le produit cartésien de deux ensembles x et y, que l'on note $x \times y$, est l'ensemble de tous les couples formés d'un élément de x suivi d'un élément de y. On pose :

$$\forall z.(z \in x \times y \Leftrightarrow \exists a \in x.\exists \underline{i}ny.z = (a, b))$$

Schéma de compréhension

Le schéma de compréhension exprime le concept originel d'ensemble : un ensemble est une entité réunissant des entités possédant une propriété commune. En logique, posséder une propriété commune, s'exprime à l'aide d'une formule caractérisant la propriété partagée par les éléments visés.

Pour des raisons théoriques profondes (le paradoxe de Russel), cette façon intuitive se donner des ensembles a dû être sérieusement contrôlée : on ne peut construire un ensemble d'éléments que comme *sous-ensemble* d'un ensemble déjà donné. Si φ exprime la propriété désirée, on note $\{y \in x \mid \varphi\}$ le sous ensemble des éléments de x qui satisfont φ. On pose :

$$\forall z.(z \in \{y \in x \mid \varphi\} \Leftrightarrow z \in x \land \varphi[z/y])$$

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 $y \in Y$ (i.e. $(x, y) \in X \times Y$). On peut alors définir *l'ensemble des relations binaires* entre deux ensembles X et Y :

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

On peut 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 :

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

Rappelons ce que sont les *domaine* (ensemble de départ) et *codomaine* (ensemble d'arrivée) d'une relation :

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

Fonctions Les fonctions sont une catégorie particulière de relations binaires : celles qui associe au plus un élément du codomaine au élément de leur domaine. Selon l'usage, nous allons employer l'opérateur logique d'existence et d'unicité $\exists!$ que l'on définit par :

$$\exists!x.\varphi \mathrel{\widehat{=}} \exists x.(\varphi \land \forall y.(\varphi[y/x] \Rightarrow y = x))$$

Comme pour les relations (puisque c'en sont), 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 (du résultat) de l'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.

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

- restriction du domaine :

$$Z \lhd R \mathrel{\widehat{=}} \{(x, y) \in X \times Y \mid x \in Z \land x \mathrel{\underline{R}} y\}$$

– exclusion du domaine :

$$Z \lhd\!\!\!- R \mathrel{\widehat{=}} (X \backslash Z) \lhd R$$

– restriction du codomaine :

$$R \rhd Z \mathrel{\widehat{=}} \{(x, y) \in X \times Y \mid y \in Z \wedge x \; \underline{R} \; y\}$$

– mise-à-jour (de la relation R_1 par la relation R_2) :

$$R_1 \oplus R_2 \mathrel{\widehat{=}} (\mathrm{dom}(R_2) \lhd\!\!\!- R_1) \cup R_2$$

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 un peu compliqué et abstrait. mais c'est possible).

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 :

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

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.

La fonction définie nulle part, c'est-à-dire, la fonction de domaine vide est aussi une suite car $1..0 = \emptyset$. C'est la suite vide. On définit la suite vide comme l'ensemble vide (i.e. la fonction définie nulle part) :

$$\langle\rangle \mathrel{\widehat{=}} \emptyset$$

Le fait que $\langle\rangle \in \mathrm{seq}\ X$ est donné par $\mathrm{dom}\langle\rangle = \emptyset = 1..0$.

La longueur d'une suite est le nombre de ses éléments, son *cardinal*. C'est un nombre entier car les suites sont, par définition, des ensembles finis. On note : $\#S$.

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. C'est un avatar de la méthode de développement VDM.

Définitions 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 :

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

Voici quelques exemples de spécifications de fonctions sur les suites.

Concaténation :

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

Sous suite :

$$
\begin{array}{|l}
sub : \text{seq } X \times I\!N \times I\!N \to \text{seq } X \\
\hline
\forall s \in \text{seq } X . \forall i, j \in I\!N . \\
\quad \text{dom}(sub(s,i,j)) = 1..j - i + 1 \\
\quad \forall k \in \text{dom}(sub(s,i,j)) . sub(s,i,j)(k) = s(k + i - 1)
\end{array}
$$

Suites vues comme des listes :

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

Suites vues comme des files d'attentes :

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

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 :

$$
\begin{array}{|l}
\multicolumn{1}{l}{Nom} \\
\hline
d\acute{e}clarations \\
\hline
formule \\
\hline
\end{array}
$$

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 :

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

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 :

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

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*) :

$$
\begin{array}{|l}
\multicolumn{1}{l}{TSQ} \\
\hline
ar : \mathrm{seq}\ TSElem \\
p : I\!N \\
\hline
p \leq \#ar \\
\hline
\end{array}
$$

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 $Remove0$

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

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 :

$Write0$

$$
\begin{array}{|l}
\Delta TSQ \\
item? : \mathbb{N} \\
from? : TSElem \\
\hline
item? \in 1..\#ar \land \\
ar' = ar \oplus \{item? \mapsto from?\} \land \\
p' = p \\
\hline
\end{array}
$$

$Read0$

$$
\begin{array}{|l}
\Delta TSQ \\
item? : \mathbb{N} \\
into! : TSElem \\
\hline
item? \in 1..\#ar \land \\
into! = ar(item?) \land \\
p' = item? \land \\
ar' = ar \\
\hline
\end{array}
$$

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 :

$ERROR$

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

L'opérateur θ 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 :

$NoneLeft$

$$
\begin{array}{|l}
ERROR \\
\hline
p = \#ar \land \\
report! = ItemErr \\
\hline
\end{array}
$$

$OutOfBounds$

$$
\begin{array}{|l}
ERROR \\
item? : \mathbb{N} \\
\hline
item? \notin 1..\#ar \land \\
report! = ItemErr \\
\hline
\end{array}
$$

$OutOfSpace$

$$
\begin{array}{|l}
ERROR \\
\hline
report! = NoSpace \\
\hline
\end{array}
$$

$Successful$

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

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 :

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

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.

Hoare-Floyd avec des spécifications ensemblistes

Préfixe d'une suite

Posons

$$Prefix(s_1, s) \mathrel{\widehat{=}} \forall i \in \mathrm{dom}(s_1).(i \in \mathrm{dom}(s) \wedge s_1(i) = s(i))$$

On veut établir la validité du triplet

$[\mathtt{s1} \in \mathrm{seq}\ X \wedge \mathtt{s} \in \mathrm{seq}\ X \wedge \mathtt{i} \in I\!N]$
```
i := 1;
While (i ≤ #s1) ∧ (i ≤ #s) ∧ s(i)=s1(i) do
  i := i+1
done
```
$[\mathtt{i} > \#\mathtt{s1} \Rightarrow Prefix(\mathtt{s1}, \mathtt{s})]$

La post condition permet de déduire si oui ou non `s1` est préfixe de `s`.

On utilise l'invariant suivant : $Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})$.

On a que :

$$\mathtt{s1} \in \mathrm{seq}\ X \wedge \mathtt{s} \in \mathrm{seq}\ X \wedge \mathtt{i} \in I\!N \Rightarrow Prefix(sub(\mathtt{s1}, 1, 0), \mathtt{s})$$

(puisque $sub(\mathtt{s1}, 1, 0) = \langle\rangle$ et $Prefix(\langle\rangle, s)$ pour toute suite s).

On a donc, par règle de l'affectation :

$[Prefix(sub(\mathtt{s1}, 1, 0)]$
```
i := 1
```
$[Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})]$

En utilisant la règle de la séquence, il faut montrer la validité de la boucle :

$[Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})]$
```
While (i ≤ #s1) ∧ (i ≤ #s) ∧ s(i)=s1(i) do
  i := i+1
done
```
$[\mathtt{i} > \#\mathtt{s1} \Rightarrow Prefix(\mathtt{s1}, \mathtt{s})]$

Montrons que :

$$\neg((\mathtt{i} \leq \#\mathtt{s1}) \wedge (\mathtt{i} \leq \#\mathtt{s}) \wedge (\mathtt{s(i)} = \mathtt{s1(i)})) \wedge Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s}) \Rightarrow \mathtt{i} > \#\mathtt{s1} \Rightarrow Prefix(\mathtt{s1}, \mathtt{s})$$

On suppose pour cela que

$H1 : \neg((\mathtt{i} \leq \#\mathtt{s1}) \wedge (\mathtt{i} \leq \#\mathtt{s}) \wedge (\mathtt{s(i)} = \mathtt{s1(i)}))$
$H2 : Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})$
$H3 : \mathtt{i} > \#\mathtt{s1}$

et on montre : $Prefix(\mathtt{s1}, \mathtt{s})$.
En effet, de $H3$, on peut déduire $sub(\mathtt{s1}, 1, \mathtt{i} - 1) = sub(\mathtt{s1}, 1, \#\mathtt{s1})$; et comme $sub(\mathtt{s1}, 1, \#\mathtt{s1}) = \mathtt{s1}$, on obtient, par $H2$ que $Prefix(\mathtt{s1}, \mathtt{s})$.

En utilisant le affaiblissement de la postcondition, il faut donc montrer la validité de

$[Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})]$
```
While (i ≤ #s1) ∧ (i ≤ #s) ∧ s(i)=s1(i) do
  i := i+1
done
```
$[\neg((\mathtt{i} \leq \#\mathtt{s1}) \wedge (\mathtt{i} \leq \#\mathtt{s}) \wedge (\mathtt{s(i)} = \mathtt{s1(i)})) \wedge Prefix(sub(\mathtt{s1}, 1, \mathtt{i} - 1), \mathtt{s})]$

C'est à dire, par la règle de la boucle

$[(\mathtt{i} \leq \#\mathtt{s1}) \wedge (\mathtt{i} \leq \#\mathtt{s}) \wedge (\mathtt{s(i)} = \mathtt{s1(i)}) \wedge \mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, \mathtt{i}-1), \mathtt{s})]$
```
i := i+1
```
$[\mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, \mathtt{i}-1), \mathtt{s})]$

Or on a

$$(\mathtt{i} \leq \#\mathtt{s1}) \wedge (\mathtt{i} \leq \#\mathtt{s}) \wedge (\mathtt{s(i)} = \mathtt{s1(i)}) \wedge \mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, \mathtt{i}-1), \mathtt{s}) \Rightarrow \mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, \mathtt{i}), \mathtt{s})$$

(c'est facile à vérifier).

Comme de plus $(\mathtt{i}+1)-1 = \mathtt{i}$, en utilisant le renforcement de la précondition, il faut montrer

$[\mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, (\mathtt{i}+1)-1), \mathtt{s})]$
```
i := i+1
```
$[\mathit{Prefix}(\mathit{sub}(\mathtt{s1}, 1, \mathtt{i}-1), \mathtt{s})]$

Ce qui est immédiat pas la règle de l'affectation.

Occurence dans une suite

On veut établir la validité de

$[\mathtt{s} \in \mathrm{seq}\ X \wedge \mathtt{i} \in \mathit{IN} \wedge \mathtt{e} \in X]$
```
i := 1;
While (i ≤ #s) ∧ s(i)≠e do
  i := i+1
done
```
$[\mathtt{i} \leq \#\mathtt{s} \Rightarrow \exists j \in \mathtt{dom}(\mathtt{s}).\mathtt{s}(j) = \mathtt{e}]$

Le schéma de preuve est trés proche de l'exemple précédent.

On considère l'invariant : $\forall j \in \mathrm{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})$

Par renforcement de la précondition et règle de l'affectation, on a

$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < 1 \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$
```
i := 1
```
$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$

Par règle de la séquence, il reste à montrer que

$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$
```
While (i ≤ #s) ∧ s(i)≠e do
  i := i+1
done
```
$[\mathtt{i} \leq \#\mathtt{s} \Rightarrow \exists j \in \mathtt{dom}(\mathtt{s}).\mathtt{s}(j) = \mathtt{e}]$

Montrons que

$$\neg((\mathtt{i} \leq \#\mathtt{s}) \wedge \mathtt{s(i)} \neq \mathtt{e}) \wedge (\forall j \in \mathrm{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})) \Rightarrow (\mathtt{i} \leq \#\mathtt{s}) \Rightarrow \exists j \in \mathrm{dom}(\mathtt{s}).\mathtt{s}(j) = \mathtt{e}$$

Pour ce, on suppose

$H1 : \neg((\mathtt{i} \leq \#\mathtt{s}) \wedge \mathtt{s(i)} \neq \mathtt{e})$
$H2 : \forall j \in \mathrm{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})$
$H3 : \mathtt{i} \leq \#\mathtt{s}$

et on montre : $\exists j \in \mathrm{dom}(\mathtt{s}).\mathtt{s}(j) = \mathtt{e}$.

L'hypotèse $H1$ est équivalente à $\neg(\mathtt{i} \leq \#\mathtt{s}) \vee \mathtt{s(i)} = \mathtt{e}$; combinée avec $H3$, on obtient que $\mathtt{s(i)} = \mathtt{e}$. Comme $H3$ nous donne aussi que $\mathtt{i} \in \mathrm{dom}(\mathtt{s})$, on a trouvé notre j.

Par règle d'affaiblissement de la postcondition, il nous donc donc montrer que

$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$
```
While (i ≤ #s) ∧ s(i)≠e do
  i := i+1
done
```
$[\neg((\mathtt{i} \leq \#\mathtt{s}) \wedge \mathtt{s(i)} \neq \mathtt{e}) \wedge \forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$

C'est-à-dire, par règle de la boucle

$[(\mathtt{i} \leq \#\mathtt{s}) \wedge \mathtt{s(i)} \neq \mathtt{e} \wedge \forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$
```
i := i+1
```
$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$

Comme on a que

$$(\mathtt{i} \leq \#\mathtt{s}) \wedge \mathtt{s(i)} \neq \mathtt{e} \wedge \forall j \in \mathrm{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e}) \Rightarrow \forall j \in \mathrm{dom}(\mathtt{s}).(j < \mathtt{i} + 1 \Rightarrow \mathtt{s}(j) \neq \mathtt{e})$$

(c'est facile à vérifier).

En utilisant le renforcement de la précondition, il faut montrer

$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} + 1 \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$
```
i := i+1
```
$[\forall j \in \mathtt{dom}(\mathtt{s}).(j < \mathtt{i} \Rightarrow \mathtt{s}(j) \neq \mathtt{e})]$

Ce qui est immédiat par la règle de l'affectation.