Programmes Corrects par Construction
Partie 2 : Exemples - Recherche binaire

Pierre-Edouard Portier

```
  var f(0..N-1): array of int {N>0}
  {(∀i,j : 0≤i<j<N : f.i≤f.j)}
; cst X: int
; "recherche"
  {present = (∃i : 0≤i<N : f.i = X)}
```

Pour déterminer si `X` est présent dans `f`, nous aurons certainement à trouver un entier `i` tel que :

```
f.i = X
```

Cette relation est trop contraignante car `X` peut ne pas être présent dans `f`. Dans ce cas, `i` indique au mieux où insérer `X` pour maintenir `f` trié :

```
f.i < X < f.(i+1)
```

`X` pouvant être ou ne pas être présent dans `f`, les deux relations ci-dessus sont unies :

```
f.i ≤ X < f.(i+1)
```

Cette dernière relation est encore trop contraignante car un `X` trop grand ou trop petit ne peut pas être encadré par des éléments de `f`.

Dans le cas d'un `X` trop grand, en adoptant la convention `f.N` $= \infty$, la relation est établie avec `i=N-1`.

Pour le cas d'un `X` trop petit, la relation est affaiblie avec la disjonction d'un prédicat supplémentaire :

```
R: f.i≤X<f.(i+1)  ∨ Q
Q: (∀i : 0≤i<N : f.i>X)
```

Puisque le tableau `f` est trié, une fois `R` établie, la postcondition du programme `"recherche"` s'établit facilement avec :

```
present:= (f.i=X)
```

Un invariant est proposé en affaiblissant R par le remplacement de `(i+1)` par une variable `j`. Ceci correspond à la stratégie de remplacement d'une constante (ici `1`) par une variable.

```
P: 0≤i<j≤N ∧ (f.i≤X<f.j ∨ Q)
```

D'où une première esquisse de programme :

```
  var i,j: int
; i,j:= 0,N {P}
; do j≠(i+1) ->
    "décrémenter (j-i) sous invariance de P"
  od {R}
; present:= f.i=X
```

Comment, sous la précondition `P ∧ j≠(i+1)`, réduire la différence `(j-i)` tout en maintenant l'invariant ?

Sous cette précondition, il existe un entier `h` tel que `i<h<j`.

La différence `(j-i)` est réduite aussi bien par `j:=h` que par `i:=h`.

Il faut choisir l'affectation qui laisse `P` invariant.

```
  P[i\h]
=
  0≤h<j≤N ∧ (f.h≤X<f.j ∨ Q)
⇐ {P, i<h<j}
  f.h≤X
```

De même pour `P[j\h]`. D'où :

```
  if f.h≤X -> i:=h
    |X<f.h -> j:=h
  fi
```

Comment choisir `h` ? Il faut établir `i<h<j`.

`h:=i+1` ou `h:=j-1` sont corrects.

Dans le pire des cas, ces choix conduisent à une recherche en `N` itérations.

Le choix `h:=(i+j) div 2` (avec `div` dénotant la division entière) garantit un nombre d'itérations proportionnel à `log.N`.

```
  var f(0..N-1): array of int {N>0}
  {(∀i,j : 0≤i<j<N : f.i≤f.j)}
; cst X: int
; var present: bool
; var i,j,h: int
; i,j:= 0,N {P}
; do j≠(i+1) ->
    h:= (i+j) div 2 {i<h<j}
  ; if f.h≤X -> i:=h
       |X<f.h -> j:=h
    fi {P}
  od
; present:= f.i=X
```

Quelques remarques :

— La connaissance de l'état trié du tableau n'entre en jeu qu'au moment de l'affectation à la variable `present`.

— La preuve de terminaison de la répétition est indépendante de `f` et de `X`.

— La correction de la solution ne dépend pas du choix d'un arrondi à l'entier supérieur ou inférieur pour `(i+j) div 2`. Ainsi, l'affectation suivante est correcte : `h:= i + (j-i) div 2`. De plus, elle réduit le risque de provoquer un débordement d'entier.

— `f` n'apparait que sous les formes `f.h` et `f.i`.
$0 \leq i < h < j \leq N \Rightarrow 0 < h < N \wedge 0 \leq i < N$
Donc `f.N` ne peut pas apparaitre pendant le calcul et la convention $f.N = \infty$ est bien fondée.