"Démontrer" qu'un programme a l'effet souhaité

- Ajouter des "assertions" valables à différents points du programme et telles que certaines sont évidentes et les autres sont des conséquences de celles déjà connues, etc.

- Et les conditions désirées sur le résultat du programme sont aussi des conséquences de certaines assertions

- Pour une suite d'instructions sans boucles c'est simple

- même avec instructions conditionnelles, ...

Exemple: échanger deux variables

- le code `t:=x; x:=y; y:=t;`
- Au début $x = x_0$ *et* $y = y_0$
- après la première affectation: $t = x_0$ *et* $y = y_0$
- et la deuxième: $t = x_0$ *et* $x = y_0$
- et à la fin: $y = x_0$ *et* $x = y_0$
- Règle de déduction: après l'affectation `x:=E`, assertion $A(x)$ est vraie si $A(E)$ était vraie avant (si `A` ne contient pas des `x`)

Exemple: échange conditionnelle

- le code `si x>y alors t:=x; x:=y; y:=t finsi`
- après le `alors` on a une nouvelle assertion garantie vraie: $x > y$
- d'où on peut déduire que avant le `finsi` on a $y > x$
- implicitement `sinon ;` ne rien faire
- après le `sinon` on a $x <= y$
- donc $x <= y$ est vrai à la fin de chaque branche

- donc, $x <= y$ est vrai à la fin de l'instruction conditionnelle.

Les boucles

- Ensemble d'assertions connues avant la boucle (la précondition)

- Ensemble d'assertions souhaitées après la boucle (la postcondition)

- Ajouter d'autres assertions vraies à chaque itération

- L'astuce est de trouver les bonnes nouvelles assertions

Démontrer qu'une condition est vraie après une boucle

Prendre l'exemple d'une boucle

```
tant que ExpressionBooleenne faire
   BlocInstruction
fintantque;
```

Trouver une assertion "invariante" telle que

- La précondition doit entraîner l'invariant au début de la première itération

- Et l'invariant Plus la `ExpressionBooleenne` entraînent que l'invariant est vrai à la fin de l'itération

- Et l'invariant Plus la négation de la `ExpressionBooleenne` entraînent la postcondition

Un exemple très simple

la somme des éléments d'un tableau T[1..n]:

```
i:=1;
s:=0;
tant que i<=n faire
   s:=s+T[i];
   i:=i+1
fintantque;
```

- L'invariant? s= somme des éléments de T de T[1] jusqu'à T[i-1]

- vraie chaque fois qu'on calcule l'expression booléenne (mais pas juste après la première affectation!)

- précondition i=1 et s=0

- pour déduire la postcondition souhaitée, ajouter à l'invariant $i \leq n + 1$

Terminaison des boucles

- La méthode donnée peut démontrer que la postcondition est vraie quand et si l'exécution d'une boucle termine

- Il faut aussi démontrer la terminaison

- définition d'une mesure du calcul qui reste à faire (p. ex. n+1-i)

- qui diminue à chaque itération

- et la boucle termine dès qu'elle est nulle (ou négative)

- une valeur entière !

Un (mauvais) algorithme de tri

- ```
  i:=1;
  tant que i<n faire
     si T[i]>T[i+1]
        alors ???;i:=1
        sinon i:=i+1
     finsi
  fintantque
  ```

- si on démontre que le `???` ne change ni l'ensemble des valeurs de `T` (mettons $val(T)$), ni `i`, on prend pour invariant $val(T) = val(T_0)$ *et* $T[1..i]$ *trié*; et démontre que la boucle trie `T` (si elle termine)

- et si `???` échange `T[i]` et `T[i+1]`, on peut démontrer aussi que la boucle termine
  ```

- mesure: $n \times nombre\ d'inversions + n - i$

Autres exemples

- recherche linéaire

- recherche dichotomique dans un tableau trié

- calcul des puissances

- pgcd (généralisé)

Des cas ou le bon invariant aide à concevoir l'algorithme?

- recherche d'une valeur dans un tableau de deux dimensions où chaque ligne et chaque colonne est triée

- recherche dichotomique dans chaque ligne prendrait temps $n \log n$;

- comparer la valeur chrerchée avec un élément au milieu du tableau n'aide pas

- comparer avec un élément dans un coin permet de supprimer une ligne ou une colonne

Un invariant peut servir à:
concevoir
expliquer
documenter
comprendre
tester
un algorithme

Trouver le vainqueur d'une élection

- Un tableau `V` (unidimensionnel) contient des votes; étant donné que quelqu'un a une majorité absolue, trouver qui dans une lecture du tableau!

- Invariant: Dans la section `V[1..x]`, personne n'a pas de majorité absolue; dans la section `V[x+1..i]`, candidat `c` a `m` votes de plus que tous les autres candidats ($m > 0$);

- facile à trouver le code qui maintient l'invariant

- ajouter une dernière phase de vérification si besoin

Les procédures et fonctions

- Démontrer pour chaque fonction () qu'elle produit le résultat souhaitée

- En utilisant ce même fait pour les autres fonctions déjà prouvées

- Pour les fonctions récursives, un raisonnement par récurrence
  qui nécessite souvent une description minutieuse du résultat dans tous les cas.