LINFO1122 :
Méthodes de Conception Programmes
Synthèse

Professeur : Charles Pecheur

Année 2023-2024

Lecture 1 : Introduction

Déboguage = programmer, tester, corriger, jusqu'à ne plus trouver d'erreurs

- ⇨ **Inefficace** en tant que méthodologie
- ⇨ Ne permet pas d'établir que le programme est **correct**
- ⇨ N'est pas fiable pour détecter que le programme n'est **pas correct**

Chaque test réussi augmente notre confiance que l'algorithme est correct mais on n'est jamais totalement certain qu'un algorithme est correct en faisant du débogage.

« Le test de programmes peut être utilisé pour montrer la présence de bugs, mais jamais pour montrer leur absence » - Dijkstra

Le Déboguage, test est incertain pour établir la correction et trouver des erreurs, mais surtout inutile comme base de construction de programmes.

3 utilités de la science de la programmation :

- ➔ Spécifier les programmes
- ➔ Vérifier les programme par rapport à leurs spécifications
- ➔ Construire les programmes sur base de leurs spécifications

Spécification = description des exigences et du comportement attendu d'un programme

Théorie du problème : on définit les structures, les opérations, caractéristiques et les propriétés utiles.

Solution : on détermine comment faire, en définissant une solution initiale et finale, pour définir par après une itération entre ces 2 situations.

Preuve : on vérifie si la solution est correcte ou non.

Correction totale = correction partielle + Terminaison

Terminaison : si les données satisfont les pré-conditions, alors le programme se termine.

Correction partielle : si les données satisfont les pré-conditions ET si le programme se termine, alors les résultats satisfont les post-conditions.

On prouve la terminaison grâce au **Variant**. Il diminue à chaque itération et est toujours positif ou nul. Le nombre d'itérations est donc fini.

On prouve la correction partielle grâce à **l'Invariant**.

- ⇨ Si les pré-conditions sont vraies initialement, alors l'invariant est vrai à l'entrée dans la boucle.
- ⇨ Si l'invariant est vrai avant une itération de la boucle, alors l'invariant est vrai après l'itération.
- ⇨ Si l'invariant est vrai à la sortie de la boucle, alors les post-conditions sont vraies finalement.

La preuve sur l'itération est une récurrence. Il y a une cas de base, un cas inductif et une conclusion.

Spécification	- Contrat entre le programmeur et le client - Dit **QUOI** le programme doit calculer
Programme	- Instructions exécutables - Dit **COMMENT** le programme doit calculer
Preuve	- Justifie que le programme satisfait la spécification - Dit **POURQUOI** le programme est ce qu'il est

Lecture 2 : Programmes Simples

Assertion de programme = proposition vraies à un point du programme

```
while l < r {
        [ 0 ≤ l < r ≤ N ]
    k := (l+r-1)/2 ;
        [ l ≤ k < r ]
    if pile[k] < X { l = k+1 ; }
    else { r = k ; }
}
```

Exemple d'assertion de programmes (en bleu)

Si $0 \le l < r \le N$ avant d'exécuter k := (l+r-1)/2,
Alors $l \le k < r$ après avoir exécuté k := (l+r-1)/2.

Triplet de Hoare [P] S [Q] :

- ⇨ Pré-condition P, programme S et post-condition Q
- ⇨ [P] S [Q] est une proposition logique (vraie ou fausse, valide ou invalide)
- ⇨ [P] S [Q] est valide si et seulement si :

Si P est vraie avant d'exécuter S

Alors l'exécution de S se termine

Et Q est vraie après l'exécution de S

Exemples de Triplet de Hoare :

[i = 0] i := i + 1 ; [i = 1]	valide
[i = 0] i := i + 1 ; [i = 0]	invalide
[i ≥ 0] i := i + 1 ; [i > 0]	valide
[i+j = 0] i := i+1 ; j := j-1 ; [i+j = 0]	valide
[true] i := 1 ; [i = 1]	valide
[true] i := i + 1 ; [i > 0]	invalide

```
t := x ;
x := y ;
y := t ;
```

Echange les valeurs de x et y.

Les **Variables Auxiliaires** sont des variables qui ne sont pas des variables du programmes et où leur valeur ne change pas au cours de l'exécution. Elles sont rigides. Ici x_0 et y_0 sont des variable auxiliaires, leur valeur ne change pas, mais c'est bien le cas de x et y cependant.

$[x = x_0 \quad y = y_0]$
```
t := x ;
x := y ;
y := t ;
```
$[x = y_0 \quad y = x_0]$

Pour toute valeur de x_0 et y_0,
Si $x = x_0 \wedge y = y_0$ avant
Alors l'exécution se termine
Et $x = y_0 \wedge y = x_0$ après

Assertion [P] en ce point, P est vrai

Affectation V := E ; où V est une variable et E une expression

Affectation simultanée $V_1, V_2,\ldots,V_n := E_1,E_2,\ldots,E_n$;

Axiome de l'Affectation :

[Q[V:=E]] V := E; [Q]

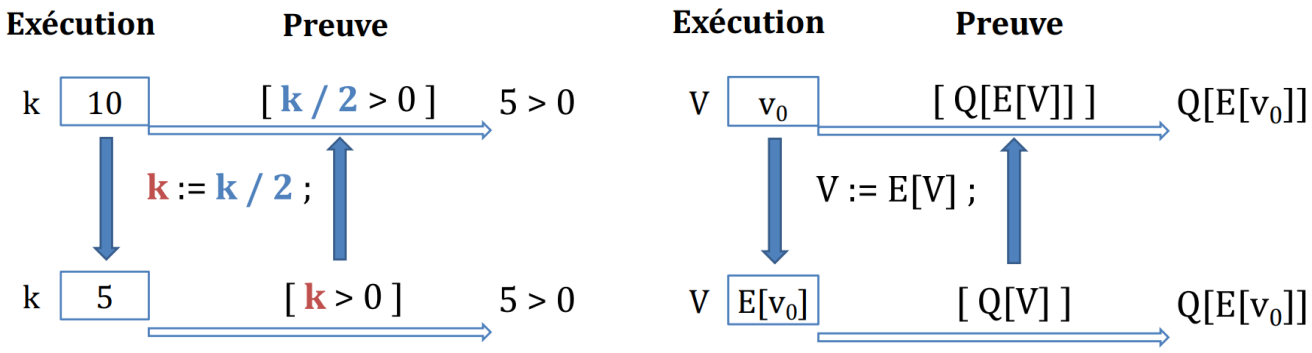
La notation [Q[V :=E] définit la condition [Q] où toutes les occurrences de V sont remplacées par E.

Axiome de l'affectation (forme alternative) :

$[Q[E]] \; V := E \; ; [Q[V]]$

$[Q[E_1, \ldots, E_n]] \; V_1, \ldots, V_n := E_1, \ldots, E_n \; ; [Q[V_1, \ldots, V_n]]$

Affectation : Exécution & Preuve



Prouver :
[y × i! = C] y, i := y × i, i – 1 ; [y × i! = C]

Axiome de l'affectation :
[(y×i) × (i–1)! = C]
y, i := y × i, i – 1 ;
[y × i! = C]

Conséquence :
y × i! = C ⇒ (y×i) × (i–1)! = C

Tableau :
[y × i! = C]
[(y×i) × (i–1)! = C]
y, i := y × i, i – 1 ;
[y × i! = C]

Séquence

Composition Séquentielle S1 S2 : est associative : {S1 S2} S3 = S1 {S2 S3} = S1 S2 S3

Règle de la Séquence : Si [P]S1[R] et [R]S2[Q], Alors [P] S1 S2 [Q].

La **preuve en arrière** : part de Q pour calculer R et puis P

La **preuve en avant** : part de P pour calculer R et puis Q

Pour construire une séquence, on invente R, S1 ou S2 et on calcule ensuite les 2 autres.

Conditionnelle

Conditionnelle if C{S1} else {S2} (avec if C {S1} ≡ if C {S1} else {})
Preuve sur conditionnelle en arrière :

[$a = a_0$]
[$(a < 0 \Rightarrow -a = |a_0|) \wedge (a \geq 0 \Rightarrow a = |a_0|)$]
if a < 0 **then**
  [$-a = |a_0|$]
  b := –a
  [$b = |a_0|$]
else
  [$a = |a_0|$]
  b := a
  [$b = |a_0|$]
fi
[$b = |a_0|$]

valeur absolue de a_0

Itération

Itération while C {S}

On se doit d'ajouter 2 nouvelles assertions :

1. L'invariant de la correction partielle
2. Le variant de la terminaison

```
while i > 0
    invariant y × i! = n! ∧ i ≥ 0 {
    y, i := y × i, i − 1 ;
}
```

- ⇨ La variable i décroit strictement à chaque itération
- ⇨ La variable i ≥ 0 (et i entier)
- ⇨ La variable i est un variant

Pour la preuve d'itération, on doit ajouter :

- Correction partielle : invariant
- Correction totale : invariant et variant

Règle de l'itération partielle

Si $[I \wedge C] S [I]$
Alors $[I]$ **while** $C \{ S \} [I \wedge \neg C]$

⇨ On utilise l'invariant I

Règle de l'itération totale

Si $I \Rightarrow V \geq 0$
et $[I \wedge C \wedge V = v_0] S [I \wedge V < v_0]$
Alors $[I]$ **while** $C \{ S \} [I \wedge \neg C]$

⇨ On utilise l'invariant I et le variant V

Lecture 3 : Induction

Un **Raisonnement Inductif** est le fait d'inférer une règle générale à partir d'observations particulières (opposition à un raisonnement déductif, qui lui dépend de preuves).

L'**Induction Mathématique**, quant à elle, part de propriétés de cas particuliers pour en déduire une propriété générale. Cette méthode est certaine car prouvée. Voici un exemple d'induction mathématique :

2 est pair et le double d'un nombre pair est pair, donc les puissances de 2 sont paires.

Principe d'induction simple
Soit P[n] une propriété dépendant de n.
- Si P[0] est vrai et
- Si P[n] est vrai alors P[n+1] est vrai

Alors P[n] est vrai pour tout n

Exemple : $P[n] \equiv \sum k \mid 0 \leq k < n :: 2^k = 2^n - 1$

Cas de base : P[0]

On prouve P[0] : $\sum k \mid 0 \leq k < 0 :: 2^k = 2^0 - 1$

Cas inductif: P[n] ⇒ P[n+1]

On pose P[n] : $\sum k \mid 0 \leq k < n :: 2^k = 2^n - 1$

On prouve P[n+1] :

$$\sum k \mid 0 \leq k < n+1 :: 2^k = \sum k \mid 0 \leq k < n :: 2^k + 2^n$$
$$= 2^n - 1 + 2^n = 2^{n+1} - 1$$

Attention à ne pas oublier le cas n = 0 ! Il est souvent très important :

- ⇨ Somme de 0 termes
- ⇨ Liste ou ensemble de taille 0
- ⇨ 0 itérations de la boucle

Principe d'Induction Complète
- Si P[k] est vrai pour tout k < n
- Alors P[n] est vrai

Alors P[n] est vrai pour tout n

Une relation $\prec$ est dite **Bien Fondée** sur un ensemble W si et seulement si :

- ⇨ Il n'existe pas de chaîne décroissante infinie dans l'ensemble W
- ⇨ Si tout sous-ensemble de W a au moins un élément minimal.

Par exemple, les nombres naturels (1,2,3,..) sont un exemple de relation bien fondée car chaque élément n'est pas plus petit que le suivant. Propriétés de la relation bien fondée :

Irréflexive : $x \not\prec x$
sinon $x \succ x \succ x \succ \dots$

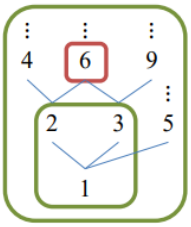
Asymétrique : $x \prec y \Rightarrow y \not\prec x$
sinon $x \succ y \succ x \succ y \succ \dots$

Pas nécessairement transitive :
on peut avoir $x \prec y$, $y \prec z$, $x \not\prec z$
donc pas nécessairement un **ordre**

Pas nécessairement totale :
on peut avoir $x \neq y$, $x \not\prec y$, $y \not\prec x$

Principe d'Induction bien-fondée

Pour une relation ($\prec$) bien-fondée (dans W)
Si
si P[y] est vrai pour tout $y \prec x$ (dans W)
alors P[x] est vrai
Alors P[x] est vrai pour tout x (dans W)



Induction complète = induction bien fondée sur l'ordre des entiers naturels

Une **Structure** est composée de structures, de formules, de phrases, de programmes, de termes. Toutes les valeurs sont à construction finie.

Relation de sous-terme

La **Relation de sous-terme** est un concept couramment utilisé en informatique, en linguistique et en logique. Elle définit une relation entre deux éléments, où l'un est considéré comme un sous-terme de l'autre. La relation de sous-terme est une relation bien-fondée.

La **Relation de sous-terme Stricte** est une relation de sous-terme complétée d'une restriction importante : un élément ne peut pas être considéré comme un sous-terme de lui-même.

$t < t'$
si et seulement si t est un **sous-terme strict** de t'
si et seulement si t **apparait dans** et est **différent de** t'
$t < t' \Leftrightarrow t' = u[t] \wedge t' \neq t$

Induction structurale = Induction bien-fondée sur les sous-termes

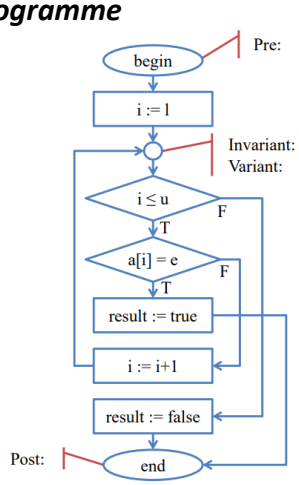
Si
si P[y] est vrai pour tout sous-terme y de x
alors P[x] est vrai
Alors P[x] est vrai pour tout terme x

- Induction Simple : P[0], P[n] ⇒ P[n+1]
- Induction Complète : (∀ k < n : P[k] ⇒ P[n]
- Induction Bien-Fondée : (∀ k ≺ n : P[k] ⇒ P[n]
- Induction Structurale : : (∀ k sous-terme de n : P[k]) ⇒ P[n]

Lecture 4 : Preuves de Programmes

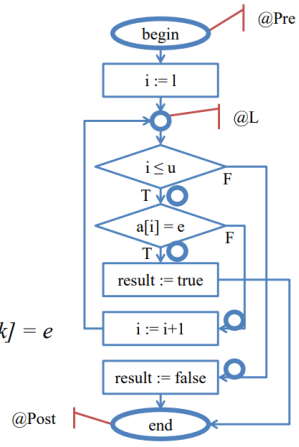
Graphe du Programme

```
[ 0 ≤ l ≤ u < |a| ]
var i := l;
while i <= u
    invariant l ≤ i ≤ u+1
        ∧ ¬∃k | l ≤ k < i :: a[k] = e
    variant u–i
{   if a[i] == e {
        result := true;
        return;
    }
    i := i+1;
}
result := false;
[ result ⇔ ∃k | l ≤ k ≤ u :: a[k] = e ]
```



Positions

```
@Pre [ 0 ≤ l ≤ u < |a| ]
var i := l;
while i <= u
    @L invariant l ≤ i ≤ u+1
        ∧ ¬∃k | l ≤ k < i :: a[k] = e
    variant u–i
{   if a[i] == e {
        result := true;
        return;
    }
    i := i+1;
}
result := false;
@Post [ result ⇔ ∃k | l ≤ k ≤ u :: a[k] = e
]
```

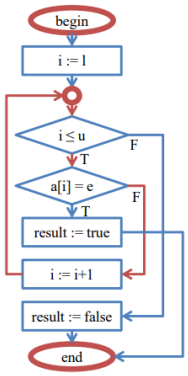


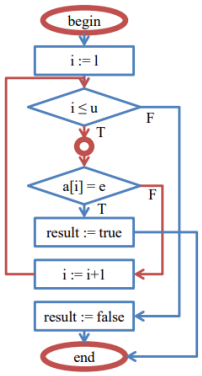
Une **Position** du programme est un point avant, après ou entre des instructions successive du programme. Cela se dénote par @label.

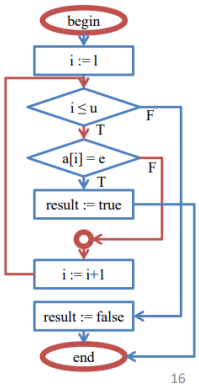
Un **Chemin** est une séquence exécutable d'instructions du programme, entre 2 positions.

Un **Chemin Simple** est un chemin qui ne passe pas 2 fois par la même instruction.

Un **Point de Coupe** est une position du programme tel que toutes les boucles contiennent au moins un point de coupe. Il peut y avoir différents point de coupe possible pour une même boucle.





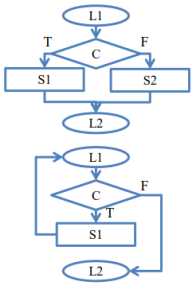


16

Les **conditions** dans les instructions du programme deviennent des **assomptions** dans les chemins simples :

@L1 ***if*** *C { S1 }* ***else*** *{ S2 } @L2*
→ *@L1* ***assume*** *C; S1 @L2*
@L1 ***assume*** *¬C; S2 @L2*

@L1 ***while*** *C { S } @L2*
→ *@L1* ***assume*** *C; S @L1*
@L1 ***assume*** *¬C; @L2*



assume C ;
≡ Le reste de l'exécution ne s'exécute que si *C* est vrai

Assume C ➔ Les conditions dans les instructions du programme deviennent des assomptions dans les chemins simples. On assume que C est vrai. On ignore les exécutions où C est faux.

Assert C ➔ en cette position, C est vrai. Assert permet de gérer certaines erreurs (dénominateur différent de 0,…).

Lorsqu'on utilise *assume*, on fait une hypothèse sur une condition qui est sensée être vraie à un certain endroit du programme. Lorsqu'on utilise *assert*, on spécifie une condition qui est censée être vraie à cet endroit du programme.

En résumé, "assume" est utilisé pour faire des hypothèses sur des conditions que vous espérez être vraies à un certain moment, tandis que "assert" est utilisé pour affirmer des conditions que vous attendez d'être vraies à un certain moment. Assert est plus strict qu'assume. Si un assert n'est pas respecté une erreur sera lancée alors que si un assume est violé, le programme pourra continuer de tourner.

3 types chemins simples :
- ➔ Affectation V := E;
- ➔ Assomption assume C
- ➔ Assertion assert C;

wp(S,Q) signifie la plus faible pré-condition de S pour Q (*Weakest Precondition*). Q est vrai après le chemin simple S ssi wp(S,Q) est vrai avant S ≡ **[P] S [Q] ssi P ⇒ wp(S,Q)**

Méthode des Assertions Inductive	***Méthode des ensembles bien-fondés***
⇨ Prouver la correction partielle de [P]S[Q] Il faut choisir un ensemble de points de coupe L_1, ..., L_n au début, à la fin du programme et dans les boucles. Ensuite, il faut associer une assertion (pré, post, invariants) P_i à chaque point de coupe L_i. Pour chaque chemin simple $@L_iS@L_j$, il faut prouver $[P_i]S[P_j]$ Si $[P_i]S_{ij}[P_j]$ est valide pour tout **chemin simple** $@L_iS_{ij}\ @L_j$, alors $[P_i]S_{ij}...S_{kl}[P_l]$ est valide pour tout **chemin** $@L_iS_{ij}...S_{kl}\ @L_l$.	⇨ Prouver la correction totale de [P]S[Q] Il faut appliquer la méthode des assertions inductives et choisir un sous-ensemble des points de coupe de L'_1,…L'_m tel que chaque boucle contient au moins un L'_i Il faut ensuite associer un variant V'_i à chaque point de coupe L'_i et pour chaque chemin simple $@L'_iS@L'_j$, prouver $[P_i \wedge V_i = v_0]S[V_j < v_0]$. Si $[P_i \wedge V_i = v_0]S[V_j < v_0]$ est valide pour tout **chemin simple** $@L'_iS_{ij}\ @L'_j$, alors $[P_i \wedge V_i = v_0]S_{ij}...S_{kl}[V_l < v_0]$ est valide pour tout **chemin** $@L'_iS_{ij}...S_{kl}\ @L'_l$.

Variant = expression dont la valeur est dans un domaine bien-fondé pour la relation considérée, c'est-à-dire sans chaîne infinie. Exemple de variant :

- ⇨ Entiers positifs, ordre arithmétique
- ⇨ Listes, ordre de tailles
- ⇨ Paires, ordre lexicographique

Lecture 5 : Procédures

L'Abstraction Procédurale consiste à abstraire un fragment de programme via un nom, une interface et une spécification. Ce fragment est utilisé pour son effet et indépendamment de son implémentation, comme un nouvel opérateur du langage

Abstraction = ignorer les détails inutiles

1. Abstraction par paramétrage : généralise sur le traitement des données et ignore les données particulières.
2. Abstraction par spécification : généralise sur l'effet du programme et ignore les implémentations particulières.

Bénéfices de l'abstraction ➔ Modularité (composé de la localité et modifiabilité) :

- ⇨ ***Localité*** : On peut lire ou écrire l'implémentation d'une abstraction sans devoir consulter l'implémentation des abstractions qu'elle utilise (ex : écrire *quicksort* sans voir le code de partition)
- ⇨ ***Modifiabilité*** : On peut modifier l'implémentation d'une abstraction sans modifier l'implémentation des abstractions qui l'utilisent. (ex : modifier swap sans modifier partition)

$$\begin{array}{l} \textit{procedure } F(V_1, \ldots, V_n) \\ \quad \textit{pre } P_F \\ \quad \textit{post } Q_F \\ \{\ S\ \} \end{array}$$

Nom F	(identifiant)
Paramètres $V_1, \ldots, V_n$	(variables)
Spécifications P_F, Q_F	(assertions)
Corps S	(instruction)

- **Typage** : les types sont des spécifications supportées dans le langage. Le Type-checking vérifier que les variables et résultats sont du bon type.
- **Paramètres et résultats multiples** : en général, une procédure peut avoir plusieurs paramètres et retourner plusieurs résultats. Dans la suite, on ne présentera des procédures qu'avec un paramètre et ne retournant qu'un résultat.
- **Instruction return** : Le résultat de la procédure ≡ une variable distinguée result. → return E ; ≡ result := E; Axiome du return : [Q[E/result]] return E [Q] ou wp(return E, Q) = Q[E/result]

Une **Variable Fraîche** est une variable dans un programme qui n'existe pas déjà dans le programme.

Une **Procédure Pure** est une procédure qui ne modifie pas les variables non-locales de la procédure. La précondition P_F porte uniquement sur les paramètres $V_1,\ldots,V_n$ et la post-condition Q_F porte uniquement sur les paramètres $V_1,\ldots,V_n$ non modifiées et le résultat *result*

procedure $F(V_1, \ldots, V_n)\ \{\ S\ \}$

- Paramètres $V_1, \ldots, V_n$ passés **par valeur**
- Paramètres $V_1, \ldots, V_n$ **non modifiés** par S
- Variables non-locales non modifiées par S

exp(a, b) est pur

swap(a, i, j) n'est pas pur : modifie *a*

Une **Procédure avec Effet** est une procédure non pure, elle peut modifier des variables non-locales, c'est-à-dire des paramètres V transmis par référence ou des parties de variables. Définition : ***procedure F(V) pre PF post QF modifies V {S}***. La valeur de V est différente à l'entrée et à la sortie de S. Exemple avec la fonction *swap* ci-dessous :

procedure *swap(a, i, j)*
  pre $0 \le i < |a| \wedge 0 \le j < |a| \wedge a = a_0$
  post $a[i] = a_0[j] \wedge a[j] = a_0[i]$
  modifies a[i], a[j]
{
  var *t := a[i] ;*
  a[i] := a[j] ;
  a[j] := t ;
}

Passage par valeur et par référence :

Passage par Valeur	***Passage par Référence***
⇨ Lorsqu'une valeur est passée à une fonction par valeur, la fonction reçoit une copie de la valeur d'origine. ⇨ Les modifications apportées à la valeur à l'intérieur de la fonction n'affectent pas la valeur d'origine en dehors de la fonction. ⇨ Ce mécanisme est souvent utilisé pour les types de données simples comme les entiers, les flottants et les caractères.	⇨ Lorsqu'une valeur est passée à une fonction par référence, la fonction reçoit une référence (un lien ou un pointeur) vers l'emplacement mémoire de la valeur d'origine. ⇨ Les modifications apportées à la valeur à l'intérieur de la fonction affectent directement la valeur d'origine en dehors de la fonction. ⇨ Ce mécanisme est souvent utilisé pour les structures de données complexes ou les objets.

Pour les procédures récursives, on suppose les appels corrects et on prouve le corps correct. Pour prouver la terminaison, on définit un variant sur les paramètres de la procédure qui doit décroître pour les appels récursifs de F dans S.

Lecture 6 : Structures de Données

Type de Données = Données + Opérations → $T \equiv (D_T, OP_T)$

Données = ensembles des valeurs possibles & Opérations = ensemble d'opérations

En général, les opérations de OP_T peuvent modifier leurs paramètres qui sont passés en référence. Elles sont donc des procédures non-pures, avec effet. Elle modifie le plus souvent un ou plusieurs paramètres de type T.

Pour un type $T = (D_T, OP_T)$, une opération $F(V_1 : T_1, ...V_n : T_n) : T' \in OP_T$:

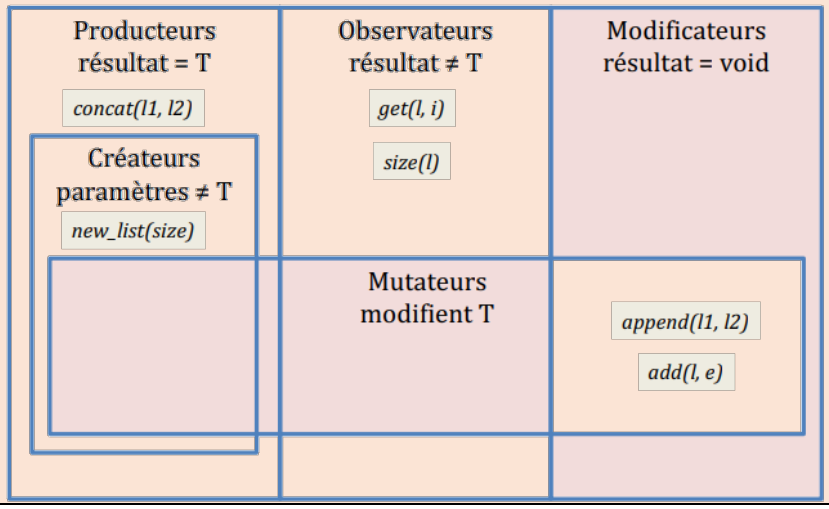
- F est **Producteur** de T ssi T' = T.
- F est **Créateur** de T ssi T' = T et $T_1, ..., T_n \neq T$
- F est **Observateur** de T ssi T' ≠ T, T' non-vide
- F est **Modificateur** de T ssi T' vide
- F est **Mutateur** de T ssi $T_i = T$ et F modifie V_i, càd il modifie un paramètre de type T. (souhaité : mutateur = modificateur, et ne modifie pas d'autres types que *T*)

Exemples de types d'opérations :

Sur un type *List* :

new_list() : List	créateur
concat(l1: List , l2: List) : List	producteur
add(l: List, e: Item) *modifies l*	mutateur
append(l1: List , l2: List) *modifies l1*	mutateur
size(l: List) : Int	observateur
get(l: List, i: Int) : Item	observateur

Types d'Opérations :



Un Type *T* est **Immutable** s'il n'a aucun mutateur. Aucune opération ne modifie de paramètre de type T (Ex : String Java). C'est une propriété du type et pas juste de son implémentation.

- ⇨ *Avantage* : on peut partager les valeurs car il n'y a plus de problème d'aliasing.
- ⇨ *Inconvénient* : il y a plus de création et de destruction d'objets.

1. Enregistrement

Un **Enregistrement** est une structure de données qui regroupe différentes variables sous un même nom. Il permet de stocker et organiser des informations liées dans une seule entité. Voici un exemple clair avec "Student" qui est un enregistrement qui contient deux champs : *name* et *birth* dans 3 langages de programmation différents :

- *Java : class Student { String name; Date birth; }*
- *C : struct Student { String name; Date birth; }*
- *Dafny : class Student {var name: String; var birth: Date;}*

type T = { F_1: T_1, …, F_n: T_n } et de type produit : $D_T = D_{T1} \times \cdots \times D_{Tn}$

3 types d'opérations sur des enregistrements :

producteur (constructeur)	$new\_F(V_1: T_1, \ldots, V_n: T_n) : T$
observateurs (accesseurs)	$get\_F_i(V: T) : T_i$
modificateurs	$set\_F_i(V: T, V_i: T_i)$ ***modifies*** V

2. Union

Une **Union** est une structure de données qui permet de stocker plusieurs types de données dans le même emplacement mémoire. Cela permet d'économiser de l'espace mémoire. Dans l'exemple ci-dessous, vous avez une union qui étend la même classe de base (Member). En Java, les classes Student et Teacher étendent la classe Member, tandis qu'en C, l'union Member peut contenir soit une variable de type Student, soit une variable de type Teacher :

- *Java, Daphny : class Student extends Member { … }*
  class Teacher extends Member { … }
- *C : union Member { Student student; Teacher teacher; }*

type T = F_1: T_1 | … | F_n: T_n et de type somme : $D_T = D_{T1} + \cdots + D_{Tn}$

2 types d'opérations sur les Unions :

producteurs (constructeurs)	$new\_F(V_i: T_i) : T$
observateurs (accesseurs)	$is\_F_i(V: T) : bool$
	$get\_F_i(V: T) : T_i$

3. Type Inductif

Un **Type Inductif** est algébrique, c'est une somme de produits, une union d'enregistrements. G1, ..., Gn sont les générateurs du type T. Exemples de types inductifs :

- *datatype List = nil | cons(head: Item , tail: List)*
- *datatype Tree = leaf(item: Item) | node(left: Tree, right: Tree)*

datatype T = $G_1(F_{11}: T_{11}, \ldots) \mid \ldots \mid G_n(F_{n1}: T_{n1}, \ldots)$

3 types d'opérations sur les Types Inductifs :

producteurs (constructeurs) $new\_G_i(V_{i1}: T_{i1}, \ldots) : T$
observateurs (accesseurs) $is\_G_i(V: T) : bool$
$get\_F_{ij}(V: T) : T_i$
modificateurs $set\_F_{ij}(V: T, V_{ij}: T_{ij})$ ***modifies*** V
erreur si $\neg is\_G_i(V)$!

Enregistrement ➔ 3 opérations

Union ➔ 2 opérations

Type Inductif ➔ 3 opérations

Le **Filtrage** en programmation est le fait de retourner différentes choses dans un programme en fonction du résultat obtenu. Une instruction de filtrage est appelée le **Pattern Matching**. ***match E { case M_1 => S_1 … case M_k => S_k }***

Exemples de filtrages ci-dessous :

```
procedure length(x: List) : int {
    match x {
        case nil => return 0 ;
        case cons(n, x1) => return length(x1)+1 ;
    }
}
```

```
procedure half(x: List) : List {
    match x {
        case nil => return nil ;
        case cons(n, nil) => return nil ;
        case cons(n1, cons(n2, x2)) => return cons(n2, half(x2)) ;
    }
}
```

M_1,…M_k sont des **Motifs** (*patterns*) constitués de générateurs et de variables. Le programme exécute le premier S_i tel que la valeur de E correspond avec le motif M_i. Le filtrage général peut se transformer en **Filtrage Primitif** :

```
match E {
    case G1(V1, ...) => S1
    ...
    case Gk(Vk, ...) => Sk
}
```

Le **Domaine** d'un type inductif sont toutes les valeurs qu'on peut construire à partir des générateurs de ce type. Voici l'algèbre des 3 premiers termes des générateurs de *Tree* :

$$D_{Tree}^{1} = \{ leaf(0), leaf(1), \ldots \}$$
$$D_{Tree}^{2} = D_{Tree}^{1} \cup \{ node(leaf(0), leaf(0)), \ldots \}$$
$$D_{Tree}^{3} = D_{Tree}^{2} \cup \{ node(node(leaf(0), leaf(1)), leaf(2)), \ldots \}$$

➔ Il faut au moins un créateur sinon $D_T = \emptyset$

Soit la **relation** $<_T$ définie sur D_T telle que $d <_T d'$ ssi d est un sous-terme de d'. Pour tout type inductif T, $<_T$ est bien-fondée sur D_T.
$d \prec_T d'$ ssi d est un sous-terme direct de d'. Pour tout type inductif T, $\prec_T$ est bien-fondée sur D_T.
Pour un type inductif T,
l'induction bien-fondé sur $<_T \equiv$ induction structurelle sur T.
Si si P[y] est vrai pour tout sous-terme y de x
alors P[x]est vrai
Alors P[x] est vrai pour tout x.

Procédure par induction structurelle :

Pour un type inductif T, pour une procédure récursive F(V: T, …) : T' { S } F est définie **par induction structurelle** sur T si pour tous les appels récursifs F(E, …) dans S, E est un sous-terme de V. *Si F est définie par induction structurelle alors F se termine toujours.* Exemple ici :

```
procedure length(x: List) : int
    pre true
    post result ≥ 0
    variant x
{
    match x {
        case nil => return 0 ;
        case cons(n, x1) =>
            return length(x1)+1 ;
    }
}
```

Définie par **induction structurelle** sur *List* :
terminaison garantie

Deux chemins simples :

```
[ x = x0 ]
assume x = nil ;
result := 0 ;
[ result ≥ 0 ]
————————————
[ x = x0 ]
assume x = cons(n, x1) ;
assert x1 < x0 ;
assume u ≥ 0 ;
result := u+1 ;
[ result ≥ 0 ]
```

Abstraction sur les Données :

Seuls certains types sont directement supportés par le langage de programmation. En général, on doit construire une représentation du type désiré sur base de types primitifs du langage. On représente un **type abstrait A** par un **type concret T**.

Le type abstrait A = structure quelconque, exprimé en mathématique et libre. (ex : ensemble fini d'entiers)

Le type concret T = type du langage de la programmation exprimé dans le langage de programmation. (ex : liste d'entiers)

On veut représenter un type abstrait A = (D_A, OP_A) sur base d'un type concret T = (D_T, OP_T). Il faut alors représenter les valeurs abstraites a ∈ D_A en valeurs concrètes d ∈ D_T. D'ailleurs, on peut avoir plusieurs d possibles pour le même a.

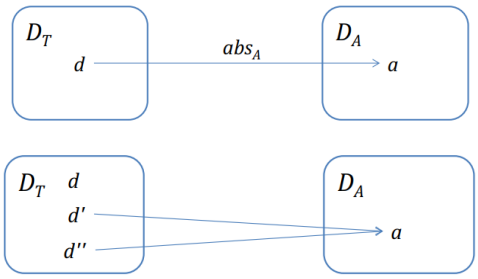
Et ensuite il faut implémenter les opérations abstraites op ∈ OP_A sous forme de procédures sur les valeurs concrètes qui utilisent les opérations concrètes de OP_T .

Fonction d'Abstraction : A = (D_A, OP_A) représenté par T = (D_T , OP_T).

- Fonction d'abstraction du type A : $\boldsymbol{abs_A : D_T \rightarrow D_A}$.
- $\boldsymbol{a = abs_A(d) \in D_A}$ est la valeur de A représenté par d ∈ D_T. C'est le coeur de l'abstraction, la première chose à définir.

abs_A est **PARTIELLE** : $abs_A(d)$ n'est pas défini si d n'est pas une représentation valide d'une valeur de A.

abs_A n'est **PAS INJECTIVE** : il peut y avoir plusieurs représentations de la même valeur a de A.



Exemple avec un ensemble d'entiers :

$D_A \equiv \wp_{fin}(\mathbb{Z})$ ensembles finis d'entiers

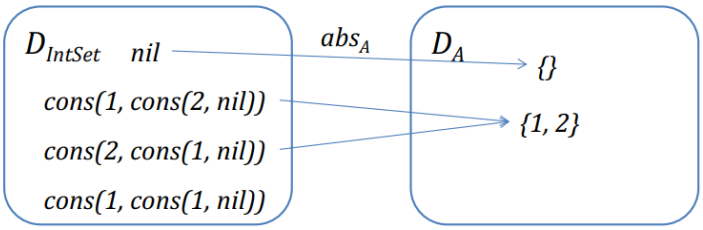
datatype *IntSet = nil | cons(head: Int, tail: IntSet)*

$abs_A : D_{IntSet} \rightarrow D_A$

$abs_A(nil) = \{\}$

$abs_A(cons(1, cons(2, nil))) = \{1,2\} = abs_A(cons(2, cons(1, nil)))$

$abs_A(cons(1, cons(1, nil)))$ n'est pas défini

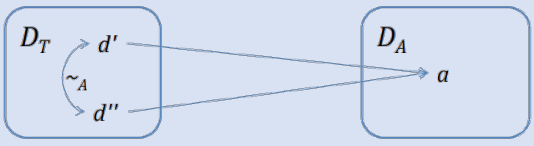


Relation d'équivalence du type A

$\sim_A : D_T \times D_T \rightarrow Bool$

$d \sim_A d' \equiv abs_A(d) = abs_A(d')$

$d \sim_A d' \equiv d$ et $d' \in D_T$ représentent la **même valeur** de A



Invariant de représentation du type A

(Rep-invariant de A)

$ok_A : D_T \rightarrow Bool$

$ok_A(d) \equiv abs_A(d)$ est défini

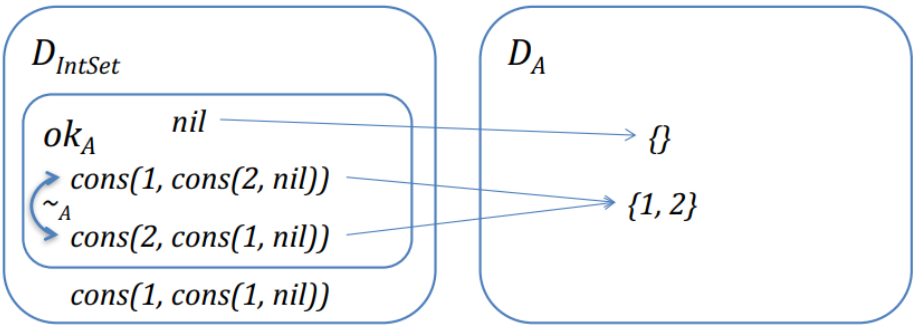
$ok_A(d) \equiv d$ représente une **valeur valide** de A

Exemple avec un ensemble d'entiers :

$D_A \equiv \wp_{fin}(\mathbb{Z})$ ensembles finis d'entiers

datatype *IntSet = nil | cons(head: Int, tail: IntSet)*

$ok_A(nil) = true$
$ok_A(cons(1, cons(2, nil))) = ok_A(cons(2, cons(1, nil))) = true$
$ok_A(cons(1, cons(1, nil))) = false$



~ détermine des classes d'équivalence dans D_T

Une Opération F est **Congruente** par rapport à l'équivalence ~ ssi des arguments équivalents donnent des résultats équivalents. ***Pour F(...) : d ~ d 0 ⇒ F(d, ...) ~ F(d 0 , ...)***

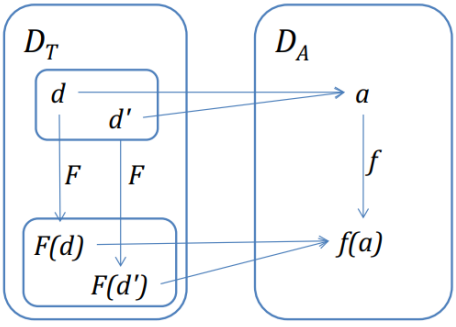
Toutes les opérations de l'abstraction doivent être congruentes par rapport à ~. C'est le cas si elles représentent correctement une opération abstraite, sinon c'est une violation de l'abstraction

Si on a une **opération abstraite** $f : D_A \to D_A$
et une **procédure** $F(V: T) : T$ qui représente f

$abs(F(d)) = f(abs(d))$

$d \sim d'$
$\Rightarrow abs(d) = abs(d')$
$\Rightarrow abs(F(d)) = f(abs(d))$
$= f(abs(d')) = abs(F(d'))$
$\Rightarrow F(d) \sim F(d')$

F est **congruente**



L'abstraction de données est à la base de la programmation orientée-objets :

- Un type abstrait représenté par une classe
- Une représentation équivalente aux variables d'instance de la classe
- Les opérations abstraites représentées par les méthodes de la classe
- Toutes ces méthodes ont un paramètre implicite, par référence, modifiable, this.

La représentation de A par T préserve le **rep-invariant** ssi

- ⇨ Tous les producteurs F(V : T, ...) : T{S} préservent ok : *[ok(V)]S[ok(result)]*
- ⇨ Tous les mutateurs F(V : T, ...) : T 0{S} préservent ok : *[ok(V)]S[ok(V)]*

Si tous les producteurs et mutateurs de O_{PT} préservent ok et
seules les opérations de O_{PT} sont utilisées pour produire des valeurs de T
Alors toutes les valeurs de T satisfont ok.

Pour assurer la modularité, il faut préserver et masquer la représentation :

- ➔ La représentation n'est pas modifiable en dehors de l'implémentation de l'abstraction **grâce à la localité**.
  On peut raisonner sur l'implémentation indépendamment du reste.
- ➔ La représentation n'est pas visible en dehors de l'implémentation de l'abstraction, **grâce à la modifiabilité**.
  On peut modifier l'implémentation indépendamment du reste.

La programmation orientée-objets permet la préservation de la représentation. **L'encapsulation** rassemble la structure qui représente l'abstraction et les opérations (méthodes) associés en une classe qui représente le type abstrait.
Les modificateurs de visibilité limitent les opérations accessibles et permettent de préserver et masquer la représentation.

Si la représentation est accessible en dehors de l'implémentation, on dit que ***l'implémentation expose la représentation***. C'est une erreur de conception !

- ⇨ causes possibles : faille de visibilité, référence exportée ou référence importée.

Effets Bienveillants :

- ⇨ Type abstrait mutable a forcément des opérations avec effet, des mutateurs.
- ⇨ Type abstrait immutable peut avoir des opérations avec effet qui change la représentation sans modifier l'abstraction, càd qui modifie d en préservant abs(d) = abs(d_0).

Un **Effet Bienveillant** est un effet qui n'est pas visible en dehors de l'abstraction. Exemple :

```
datatype Rat =
      rat(num: Real, den: Real)
ok(rat) = den(rat) > 0
abs(rat) = num(rat) / den(rat)

procedure new_rat(n: Real , d: Real)
      pre true
      post abs(result) = n/d
{     return rat(n, d);  }

procedure reduce(q: Rat)
      pre q=q0
      post abs(q)=abs(q0)
      modifies q
{     ...   }
```

```
procedure equal(q1: Rat, q2: Rat)
      pre q1 = q10 ∧ q2 = q20
      post result ⇔ abs(q1) = abs(q2)
            ∧ abs(q1)=abs(q10)
            ∧ abs(q2)=abs(q20)
      modifies q1, q2
{
      reduce(q1) ;
      reduce(q2) ;
      return (num(q1) = num(q2) ∧
            den(q1) = den(q2)) ;
}
```

a) Représentation de l'équivalence

On veut représenter l'égalité du type abstrait par une procédure eq(d, d')

Si on a *abs* sous forme logique, on peut spécifier et prouver
procedure *eq(d: T, d': T) : Bool*
pre *ok(d)* $\wedge$ *ok(d')*
post *result* $\Leftrightarrow$ *abs(d) = abs(d')*
{ ... }
et donc *eq* $\equiv \sim$ est une congruence

Sinon, *eq* décrit implicitement $\sim$ et donc *abs*
et il reste à prouver que *eq* est une congruence

b) Représentation Canonique

La représentation de A par T est canonique si toutes les valeurs absraites ont une représentation unique.

Dans ce cas, $d \sim d' \Leftrightarrow abs(d) = abs(d') \Leftrightarrow d = d'$
On peut définir *eq(d, d') { **return** (d = d'); }*

Lecture 7 : Preuves Automatiques

Chemin Simples :

On peut décomposer automatiquement tout programme en chemins simples constitués de :

- **Affectation *V := E ;***
- **Assomption *assume C ;***
- **Assertion *assert C ;***

On peut calculer automatiquement la + faible pré-condition *wp(S,Q)* sur un chemin simple S. La preuve automatique fonctionne donc bien pour des chemins simples, mais un prouveur d'assertions est nécessaire.

Boucles :

Aucun logiciel n'est encore capable de trouver l'invariant et le variant d'une boucle, il faut donc qu'ils soient fournis par le programmeur. Une fois les invariants et le variant trouvé, le logiciel peut décomposer ces boucles en chemins simples : la preuve automatique fonctionne.

Programme :

Comme pour les boucles, le programmeur doit fournir les spécifications (pré/post, effets, variants). Le prouveur automatique peut ensuite faire son travail par décomposition en chemins simples

Étapes d'une preuve automatique de programme :

1. Le programmeur spécifie les pré/post-conditions, les invariants, les variants et les effets
2. Le prouveur de programmes :

- considère chaque procédure séparément
- décompose en chemins simples [P] S [Q]
- calcule Q_0 = wp(S,Q) pour chaque chemin simple

3. Le prouveur d'assertions prouve P ⇒ Q_0

Le Programmeur peut ajouter des assertions et assomptions dans le programme et le Prouveur automatique peut ajouter des assertions concernant les erreurs d'exécutions. Le Prouveur de programme prouve :

➔ La correction partielle sur toutes les procédures par assertions inductives, chemins simples en utilisant les invariants de boucle
➔ La décroissance des variants spécifiés, sur les boucles et procédures

Si variants sont spécifiés sur toutes les boucles et procédures récursives ➔ correction totale.

Une **Théorie** est l'ensemble des propriétés satisfaites par un ensemble d'opérations et de fonctions donné. Le prouveur d'assertions fait des preuves par rapport à certaines théories. Voici des exemples de théories :

Egalité : =
Arithmétique : $\mathbb{N}$, +, ×, =, >
Arithmétique linéaire : $\mathbb{Z}$, +, –, =, >
Réels : $\mathbb{R}$, +, –, ×, =, ≥
Réels linéaire : $\mathbb{R}$, +, –, =, ≥
Structures inductives : *cons, head, tail, atom,* =
Tableaux : _[_], _[_]:=_, =

Prouveurs courants supportent l'arithmétique linéaire, par l'arithmétique générale

Prouveurs d'assertions est appelé un ***solveur SMT*** (Satisfaisabilité Modulo Théories)

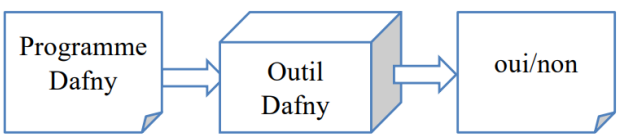
⇨ Il n'existe pas de procédure décidant de la validité pour tout P
⇨ La validité est semi-décidable, on peut énumérer les P valides
⇨ Un solveur SMT peut répondre, « je ne sais pas »

Il existe différents vérificateurs de programmes, dont Dafny, celui que nous allons utiliser.

Preuves de Programme en Dafny :

Langage Dafny = langage de programmation orienté-objets avec spécifications

Outil Dafny = logiciel de vérification de programmes Dafny



Lecture 8 : Algorithmes

Un **Algorithme** est une procédure pour accomplir une certaine tâche. Cela peut être soit :

- Un Problème (*trier une liste de nombres*)
- Une Instance (*trier la liste [23,49,57,34,8]*)
- Une Spécification (*Données : une liste de n nombres a1, …, an. Résultat : permutation a'1, …, a'n de a1, …, an telle que a'1 ≤ … ≤ a'n*)

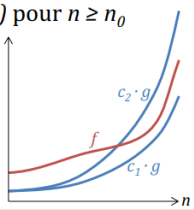
On veut des algorithmes corrects, efficaces mais facile à implémenter. On peut préférer un algorithme non-optimal mais plus simple. Il y a des principes généraux applicables à la conception d'algorithmes, et ceux-ci seront décrits dans les sections suivantes.

Rappel : analyse de complexité

Pour un problème de taille n
$T(n)$ = **temps** pour résoudre un problème de taille n dans le **pire cas**
$S(n)$ = **espace** pour résoudre un problème de taille n dans le **pire cas**
(variantes : temps/espace moyen, temps/espace amorti)

- $f(n) = O(g(n)) \equiv f(n) \le c_2 \cdot g(n)$ pour $n \ge n_0$
- $f(n) = \Omega(g(n)) \equiv f(n) \ge c_1 \cdot g(n)$ pour $n \ge n_0$
- $f(n) = \Theta(g(n)) \equiv c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ pour $n \ge n_0$

$O(c \cdot f(n)) = O(f(n))$
$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$



Classes de complexité

$O(1) < O(\lg n) < O(n) < O(n \cdot \lg n) < O(n^2) < O(2^n) < O(n!)$

n	$f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10		0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20		0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30		0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40		0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50		0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100		0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000		0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000		0.013 μs	10 μs	130 μs	100 ms		
100,000		0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 μs	1 sec	29.90 sec	31.7 years		

1. Décomposition en sous-problèmes

Diviser les problèmes difficiles à résoudre en plusieurs petits sous-problèmes faciles à résoudre.

```
procedure Prob(x) pre … post … {
    var x1 := Subprob1(x) ;
    var x2 := Subprob2(x, x1) ;
    return solution(x, x1, x2) ;
}
```

2. Diviser pour Régner

On décompose un problème en sous-problèmes plus petits qu'on résoud ensuite. Enfin, on combine les solutions des sous-problèmes en une solution du problème initial. Pour ce faire on utilise la récursion.

➔ Cette méthode est utile si le gain sur la réduction en problèmes plus petits est supérieur au travail supplémentaire pour la décomposition et combinaison.
➔ Cette méthode est très efficace lorsque les sous-problèmes sont indépendants, elle donnera en revanche des résultats beaucoup moins satisfaisants lorsqu'ils ne le sont pas (exemple : Fibonacci)

3. *Programmation Dynamique*

On décompose le problème en sous-problèmes, on résoud les sous-problèmes une seule fois et on mémorise leur solution, pour enfin utiliser les solutions pour la résolution du problème.

Compromis espace-temps : réduit le temps de calcul + augmente l'espace mémoire

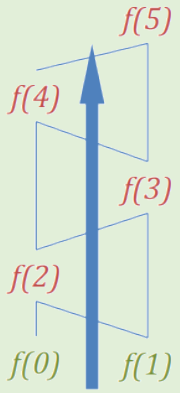
La programmation dynamique est très efficace lorsque les sous-problèmes comportent les mêmes parties communes (réaliser le même calcul par ex), résout chaque sous-problème une seule fois et mémorise la réponse dans un tableau, évitant ainsi le recalcul. Il faut cependant trouver un compromis entre l'espace mémoire utilisé et le temps.

Programmation dynamique est applicable lorsque :

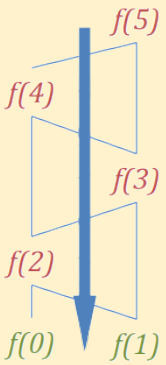
- **Sous-structures optimales** : la solution à un problème se décompose en plusieurs solutions à des sous-problèmes
- **Recouvrement des sous-problèmes** : la solution à un problème fait appel plusieurs fois aux mêmes sous-problèmes

2 Approches de programmation dynamique :

1. **BOTTOM-UP**
   - ⇨ Consiste à calculer et mémoriser les solutions des sous-problèmes liés au problème principal, à partir du plus simple jusqu'au problème principal (bottom-up) en utilisant les solutions des sous-problèmes déjà calculés.
   - ⇨ Solution peu naturelle, et risque de résoudre des sous-problèmes déjà résolus, mais réduit fortement le coût en espace car ne garde que les résultats nécessaires à la suite.



2. **TOP-DOWN**
   - ⇨ dans un algorithme récursif, consiste à mémoriser le résultat pour chaque sous-problème résolu (=mémoïsation) et retourner le résultat mémorisé pour les appels successifs au même problème.
   - ⇨ Cette solution résoud uniquement les sous-problèmes nécessaires, mais garde en mémoire tous les sous-problèmes résolus
   - ⇨ S'applique à un programme récursif, en particulier diviser pour régner.



La **Mémoïsation** est le fait de mémoriser les résultats d'une fonction (récursive ou non).

L'Aliasing arrive lorsqu'on a une référence partagée.

Lecture 9 : Programmation Orientée-Objets I

Programmation Orientée-Objets = paradigme de programmation le plus répandu et méthodologie de conception la plus répandue. Voici ces caractéristiques majeures :

- *Encapsulation* : Objets = données + opérations (variables + méthodes)
- *Héritage* : Construction incrémentale, sous-typage
- *Polymorphisme* : Réponses différentes pour la même requête

Quelques Définitions :

- ⇨ **Objet** = composant à l'exécution, identifiable (référence)
- ⇨ **Attributs** = données de l'objet. Variables, champs
- ⇨ **Méthodes** = opérations de l'objet, on invoque les méthodes d'un objet
- ⇨ **Constructeurs** = opérations qui créent un nouvel objet
- ⇨ **Classe** = module logiciel définissant des objets : attributs, méthodes spécifications, signatures, corps
- ⇨ **Interface** = ensemble d'attributs et d'opérations visibles spécifications, signatures
- ⇨ **Classe abstraite** = classe avec certaines méthodes non implémentées spécifications, signatures, certains corps
- ⇨ **Interface** = type abstrait de classe

L'Héritage consiste à définir une nouvelle classe en réutilisant et étendant une classe existante. classe-mère qui étend la classe-fille, et classe fille qui hérite de la classe-mère.

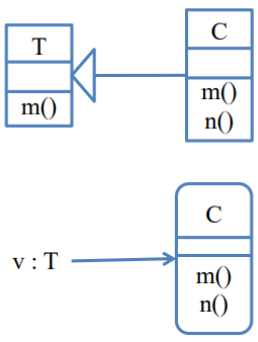
A est un sous-type de B :

- ➔ Si et seulement si tout les objets de type A sont de type B
- ➔ Si et seulement si les fonctionnalités de A incluent les fonctionnalités de B

Si A étend ou implémente B alors A est un sous-type de B.

Soit une variable *v* : T qui réfère à un objet *o* : C où C (avec méthode m() et n()) est un sous-type de T (avec méthode m()).

- ⇨ ***Typage Statique*** : Seules les méthodes de T peuvent être invoquées sur v.
- ⇨ ***Liaison Dynamique*** : quand on invoque une méthode v.m(), c'est C.m() qui est exécutée
- ⇨ ***Polymorphisme*** : un même appel à m() peut exécuter différentes implémentations de m().



Diagrammes de classe : Relations

Association : une relation entre A et B

Dépendence : A dépend de B, les changements de B affectent A

Navigation : 

A peut accéder à B, A a une référence à B

Aggrégation : 

A a-un B, B appartient à A (et d'autres)

Composition : 

A a-un B, B fait partie de A (et aucun autre)

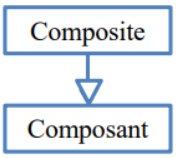
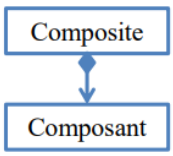
Généralisation : 

A est-un B, B généralise A, A est un sous-type de B

Multiplicité = Chaque A est associé avec entre m' et n' B, alors que chaque B est associé avec entre m et n A. Quelques cas spéciaux :

- n ou n..n : exactement n ;
- ∗ ou 0..∗ : n'importe combien ;
- 0..1 : zéro ou un ;
- 1..∗ : au moins un.

Héritage vs Composition :

Héritage	***Composition***
- Méthodes de Composant invocables directement - Composite sous-type de Composant - Composant visible sur Composite - Instance de Composant fixée par construction - Un seul Composant - Composant est une classe - **Uniquement si le sous-typage est utile** 	- Méthodes de Composant doivent être redirigées - Composite non lié à Composant - Composant pas visible sur Composite - Instance de Composant peut varier à l'exécution - Plusieurs composants de types variés - Composant peut être une interface - **Préféré en général** 

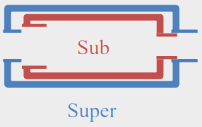
Principe de Substitutivité – Loi de Liskov : Un sous-type doit **respecter le contrat** de son super-type, de sorte qu'une instance du sous-type soit **substituable** à une instance du super-type.

Soit deux méthodes U_{super} **Super.m(T_{super}x)** et U_{sub} **Sub.m(T_{sub}x)**. Sub.m() est substituable pour Super.m() si :

- ⇨ Sub.m() accepte des paramètres de type T_{super}
- ⇨ retourne un résultat de type U_{super}

On a donc que T_{super} est un sous-type de T_{sub} et que U_{sub} est un sous-type de U_{super}.

- paramètres : T_{super} est un sous-type de T_{sub}
- résultat : U_{sub} est un sous-type de U_{super}

$T_{super} \leq T_{sub}$ 

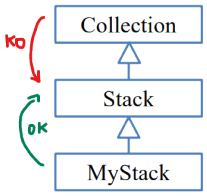
$U_{sub} \leq U_{super}$

Exemples de Substitutivité :

Stack :
Stack copy(Stack s)

MyStack :

MyStack copy(Stack s)	Correct
Collection copy(Stack s)	Incorrect
Stack copy(MyStack s)	Incorrect
Stack copy(Collection s)	Correct
MyStack copy(Collection s)	Correct

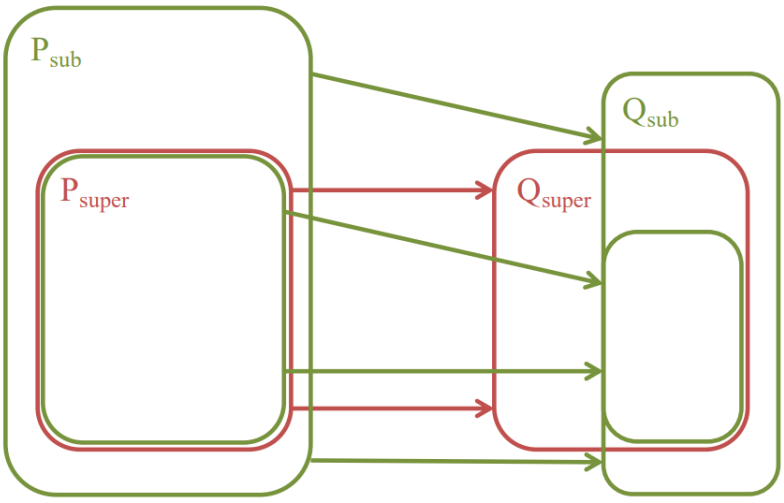


Super.m()	Sub.m()
pre P_{super}	pre P_{sub}
post Q_{super}	post Q_{sub}

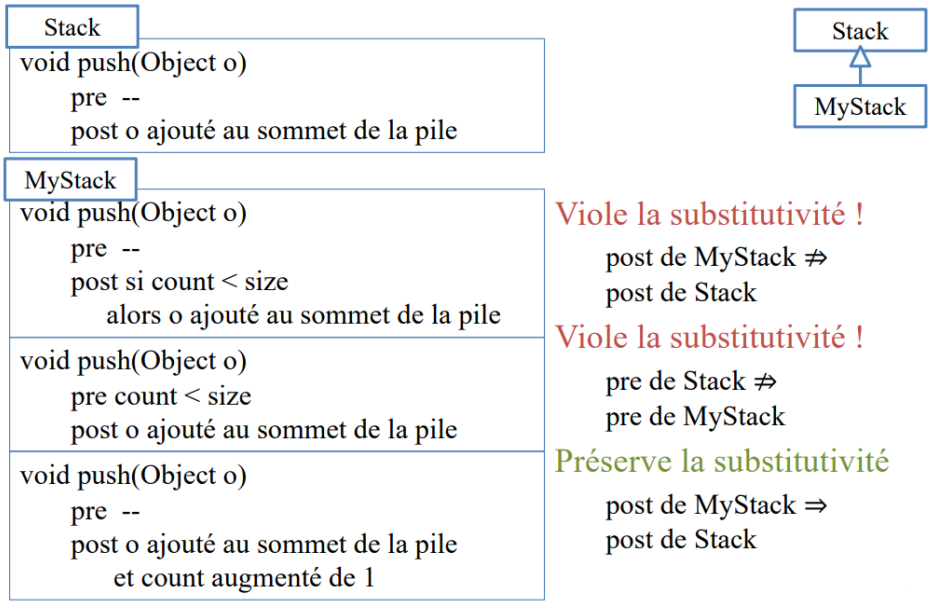
Les spécifications de Sub.m() doivent respecter les spécifications de Super.m().

- Pour la précondition, $P_{super} \Longrightarrow P_{sub}$
- Pour la postcondition, $P_{super} \Longrightarrow (Q_{sub} \Longrightarrow Q_{super})$.

⇨ Dans ce cas, $[P_{sub}]$ Sub.m() $[Q_{sub}] \Longrightarrow [P_{super}]$ Super.m() $[Q_{super}]$



Substitutivité : Exemples



Un Programme Orienté-Objet est **Modulaire** si des problèmes différents sont traités dans des modules différents. Séparation des problèmes : un problème = un module (+ facile à comprendre et écrire, + facile à modifier et + facile de trouver les fautes).

- ➔ Pour maximiser la cohésion on ajoute l'interdépendance au sein d'un module.
- ➔ Pour minimiser le couplage, on ajoute l'interdépendance entre les modules.

Principe de moindre connaissance : Un module ne doit avoir connaissancez que des modules qui lui sont directement relatifs. En orienté-objets : Une classe ne doit invoquer que les méthodes des classes qu'elle référence directement.

- Permet de réduire le couplage
- Demande des méthodes supplémentaires

Patrons de Conceptions (*Design Pattern*) :

Un **Patron de Conception** (design pattern) est un arrangement caractéristique de modules, permettant de résoudre un problème de conception en suivant certains principes de conception. C'est un schéma de solution, pas une librairie prête à l'emploi.

Un **Patron d'Architecture** est une solution générale pour l'organisation d'ensemble d'un logiciel. Il possède une plus large échelle qu'un patron de conception.

Voici les différents éléments d'un patron :

- ⇨ **Nom** : pour faciliter la référence
- ⇨ **Problème** : « quand peut-on l'appliquer et pour quels objectifs ?»
- ⇨ **Solution** : les éléments (classes, interfaces, variables, méthodes) et relations entre eux
- ⇨ **Conséquences** : bénéfices, possibilités, coûts, compromis.

On retrouve pour la conception orientée-objet 23 patrons (5 créationnels, 7 structurels et 11 comportementaux). ➔ Nous allons en voir certains parmi ces 23-là.

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

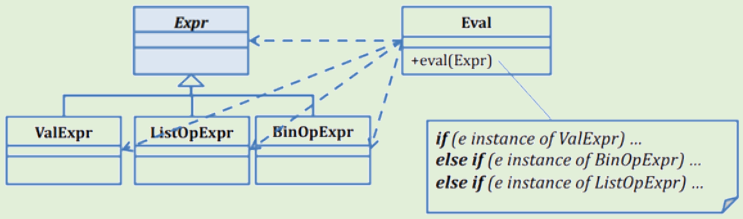
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

	Push	***Pull***
Accès à la source	Fournie en paramètre de update(), identifie la source. Un observateur peut observer plusieurs sources	Observateur doit y accéder par d'autres moyens, il faut connaitre la source par ailleurs. Un observateur pour une seule source
Données	Plus de données que nécessaire	Accède uniquement aux données nécessaires

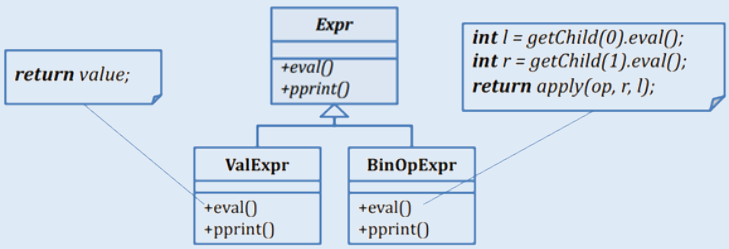
Parcours procédural



Une classe par calcul (dispatch sur le calcul)

- ⇨ Facile d'ajouter un nouveau calcul
- ⇨ Difficile d'ajouter un type d'expression

Dispatching sur les données



Une méthode de Expr par calcul

- ⇨ Facile d'ajouter un nouveau type d'expression
- ⇨ Difficile d'ajouter un nouveau calcul