Initiation au Génie Logiciel

Cours 7

–

Propriétés logiques et programmation par contrats

Plan

Plan

Invariants, pré et post conditions

- Ce sont des principes formels de raisonnement sur les programmes
- Définis pour la logique de Hoare (Hoare 1969)
- Utilisé pour la programmation par contrats (Meyer 1985)
- Trois grands types d'assertions/propriétés ($\approx$ conditions booléennes)

Définition 1 (Invariant)

Un invariant est une propriété qui est (constamment) vraie pendant l'exécution d'un programme.

Définition 2 (Précondition)

Une précondition est une propriété qui est vraie avant l'exécution d'un programme.

Définition 3 (Postcondition)

Une postcondition est une propriété qui est vraie après l'exécution d'un programme.

Invariants, pré et post conditions en Scala

Exemple 4 (Précondition en Scala – `require`)

```
var credit= 1000

def retrait(x:Int):Int={
  require(x <= credit)
  credit= credit - x
  credit
}
```

Exemple 5 (Invariant en Scala – `assert`)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  assert(credit >= 0)
  credit
}
```


Invariants, pré et post conditions en Scala (II)

Exemple 6 (Postcondition en Scala – `ensuring`)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  credit
} ensuring (credit >= 0)
```

Exemple 7 (Postcondition sur le résultat : fonction anonyme)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  credit
} ensuring (res => res >= 0)
```


Invariants, pré et post conditions en Scala (III)

Exemple 8 (Postcondition et valeurs antérieures : variables fantômes)

```
var credit= 1000


def retrait(x:Int):Int={

  credit= credit - x
  credit
} ensuring (res => (res== ???))

var credit= 1000
var creditPre=credit // Variable "fantôme" utilisée pour
                     // exprimer la propriété
def retrait(x:Int):Int={
  creditPre=credit
  credit= credit - x
  credit
} ensuring (res => (res== creditPre - x) && credit==res)
```


Invariants, pré, post-condition – Le quizz

Quizz 1 (Sur les programmes suivant quelle propriété va être violée ?)

```
def plus(x:Int,y:Int):Int={
  require(x+y>0)
  x
} ensuring (res =>(res==x+y))

plus(10,20)
```

V *precond.* R *postcond.*

```
def moins(x:Int,y:Int):Int={
  require(x>y)
  x - x
} ensuring (res => res>0)

moins(10,20)
```

V *precond.* R *postcond.*

```
def f(a:Int):Int={
  require(a match {
    case 18 | 19 => true
    case _ => false})
  18}
}
f(10)
```

V *precond.* R *aucune*

```
def max(a:Int,b:Int):Int={
  30
} ensuring (res => (res>=a
                    && res>=b))

max(10,20)
```

V *aucune* R *postcond.*

Utilisation des assertions en Scala sous Eclipse

Les assertions sont vérifiées pendant l'exécution du programme

- Elles permettent de détecter et de localiser plus facilement les erreurs
- Pour une fonction f donnée, elles déterminent si l'erreur provient :
  - d'un mauvais appel de f (exception sur `require`)
  - d'une erreur dans le corps de f (exception sur `assert`)
  - d'une erreur dans le résultat produit par f (exception sur `ensuring`)

Avant de livrer le logiciel, la vérification des assertions peut être désactivée

(par exemple, si elles diminuent significativement ses performances)

Remarque 1 (Désactivation des assertions Scala dans Eclipse)

Clic droit sur le projet → Properties → Scala Compiler → Advanced → Use Project Settings → Xdisable-assertions

Plan

Décrire les propriétés par des formules logiques ?

En GEN, formule logique ≈ condition booléenne

Exercice 1 (Quelle formule pourrait définir le résultat de `max` ?)

```
def max(a:Int,b:Int):Int={
  [...]
} ensuring (???)
```

Remarque 2 (Comment définir un résultat de fonction par une formule ?)

- *Ne pas se limiter à un cas particulier : généraliser la formule*
- *Si nécessaire, utiliser des fonctions auxiliaires pour la vérification*
- *Si nécessaire, définir des fonctions auxiliaires pour la vérification*

Exercice 2 (Quelle formule pourrait définir le résultat de `delete` ?)

```
def delete(x:Int,l:List[Int]):List[Int]={
  l.filter(_ != x)
} ensuring (???)
```


Décrire les propriétés par des formules logiques ? (II)

Quizz 2 (Les postconditions définissent elles correctement ce qui est attendu ?)

```
def divisionEntiere(n:Int,d:Int):(Int,Int)={
  require(d>0 && n>=0)
  [...]
} ensuring (res => n== (res._1 * d + res._2) &&
                   res._2<= d)
```

V Oui
R Non

```
def plus(x:Int,y:Int):Int={
  [...]
} ensuring (res => res==plus(x,y))
```

V Oui
R Non

```
def intersect(a:Set[Int],b:Set[Int]):Set[Int]={
  [...]
} ensuring (res => res.forall(a.contains(_)))
```

V Oui
R Non

Décrire les propriétés par des formules logiques ? (III)

Exercice 3 (Quelle formule pourrait définir `intersect` ?)

```
def intersect(a:Set[Int],b:Set[Int]):Set[Int]={
  [...]
} ensuring (???)
```

Exercice 4 (Quelle formule pourrait définir `triCroissant` ?)

```
def triCroissant(l:List[Int]):List[Int]={
  [...]
} ensuring (???)
```


Plan

Des formules pour factoriser les tests

On peut (parfois) automatiser l'écriture des tests unitaires par :

- écriture de la post-condition de la fonction (a.k.a. oracle)
- des tests automatiques exhaustifs/aléatoires (a.k.a. QuickCheck)

Exemple 10 (Sur `intersect(a:Set[Int],b:Set[Int]):Set[Int]`)

- au lieu de

```
assertEquals(Set(),intersec(Set(0),Set(1,2)))
assertEquals(Set(2),intersec(Set(1,2),Set(2,3)))
// 2 tests en 2 lignes
```

- écrire la post-condition de la fonction `intersec`
  définir un générateur aléatoire d'ensembles d'`Int` : `genSet:Set[Int]`

```
for (i<-1 to 100000) insersec(genSet,genSet)
// 100.000 tests en 1 ligne!
```

Démo sur le package testers.

Des formules pour factoriser les tests (II)

Exemple 11 (La post-condition d'`intersec`)

```
def intersec(a:Set[Int],b:Set[Int]):Set[Int]={
 [...]
} ensuring (inter =>{
     var res=true
     // tous les éléments de l'intersection sont dans a ET b
     for (e <- inter) res = res && a.contains(e) && b.contains(e)
     // les éléments dans a ET dans b sont dans l'intersection
     for (e <- b) if (a.contains(e)) res=res && inter.contains(e)
     res
  })
```

Exemple 12 (La post-condition d'`intersec`, version ordre supérieur)

```
def intersec(a:Set[Int],b:Set[Int]):Set[Int]={
 [...]
} ensuring (inter =>
             inter.forall (e => a.contains(e) && b.contains(e))
             && a.forall(e => !b.contains(e) || res.contains(e)))
```

Des formules pour factoriser les tests (III)

Exemple 13 (Testeur aléatoire)

```
import scala.util.Random
class TestInter {
  val rand= new Random

  // Générer un entier aléatoire entre i et j
  def genInt(i:Int,j:Int)= rand.nextInt(j-i+1)+i

  //Générer un ensemble, de taille i, d'entiers aléatoires
  def genSetL(i:Int):Set[Int]=
    if (i<=0) Set[Int]() else genSetL(i-1)+genInt(0,10)

  //Générer un ensemble, de taille aléatoire, d'entiers aléatoires
  def genSet= genSetL(genInt(0,5))

  @Test
  def test1{                          //100.000 tests, hop!
    for (i<-1 to 100000) intersec(genSet,genSet)
  }
```


Plan

Programmation défensive

En programmation défensive, autour d'une fonction `f` :

- l'utilisateur se prémunit d'une implantation erronée/malveillante de `f`
- l'implanteur se prémunit d'une utilisation erronée/malveillante de `f`

	Interface	
Utilisateur de `f`	`def f(x:T1):T2`	Implanteur de `f`
vérifie : • le résultat de `f` • que `f` n'a pas modifié `x` • que l'appel de `f` n'a pas modifié l'état global • ...		vérifie que les entrées (`x`) de `f` sont correctes

Se protéger en tant qu'implanteur

Exemple : `max(x:Array[Int]):Int`, maximum d'un tableau d'entiers

L'implanteur de `max` doit vérifier que :

- `x` n'est pas la référence `null`
- le tableau `x` comporte au moins un élément

```
def max(x:Array[Int]):Int={
    require(x!=null)    // rejette le pointeur null
    require(x.length>0) // rejette les tableaux vides
    var max= x(0)
    for (i <- x) if (i>max) max=i
    max
}
```

Se protéger en tant qu'utilisateur

Exemple : `max(x:Array[Int]):Int`, maximum d'un tableau d'entiers

Exercice 5 (Utilisez `max` pour calculer le maximum d'un tableau)

```
val t=Array(1,2,3)
...
var res= max(t)
...
```

Quel code faut-il placer autour de l'appel à `max` pour se protéger d'implantations erronées ? Des exemples d'implantations erronées :

1.
```
def max(x:Array[Int])=
  Int.MaxValue
```

2.
```
def max(x:Array[Int])= x(0)
```

3.
```
def max(x:Array[Int])= {
  var max= x(0)
  for (i<-x) if(i>max) max=i
  x(0)=10
  max
}
```


Plan

La programmation par contrats

La programmation par contrats définit les propriétés attendues sur les fonctions dès leur conception (avant leur implantation)

≪ de la programmation défensive prévisionnelle ≫

l'utilisateur et l'implanteur se protègent à l'aide d'un contrat

Le contrat définit l'utilisation/implantation des fonctions partagées

La programmation par contrats (exemple)

La programmation par contrat raffine la notion de trait/interface

- Un trait/interface donne le nom et le types des opérations
- Un contrat le complète par des propriétés attendues sur ces opérations

Exemple 14 (Reprise de l'exemple `IntQueue` du Cours 3)

Utilisateurs	Interface	Implanteurs

Une équipe réalise un logiciel *utilisant* des `IntQueue`	`trait IntQueue {`
	`def get:Int`
	`require( !empty)`
	`def put(x:Int)`
	`ensuring( !empty)`
	`def empty:Boolean}`
		Une équipe *implante* le trait `IntQueue` dans une classe `MyQueue`

Remarque 3 (Choix et syntaxe des contrats dans l'Exemple 14)

Dans l'exemple, les contrats sont volontairement simples, pour illustrer. Nous verrons la syntaxe Scala pour définir ces contrats dans la suite.

La programmation par contrats (II)

Le contrat définit les responsabilités de l'utilisateur (ou client) et de l'implanteur (ou fournisseur)

- L'utilisateur doit s'assurer qu'il appelle les opérations en respectant les préconditions
- L'implanteur doit s'assurer que les opérations qu'il développe satisfont les postconditions

Quizz 3 (L'utilisateur et l'implanteur ont-ils respecté le contrat ?)

```
def retarder(q:IntQueue):Unit=
  q.put(q.get)

def vider(q:IntQueue):Unit=
  while(!q.empty) q.get
```

V Oui | R Non

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0)
            b=b.drop(1)
            h}
  def put(x:Int)= {b=b:+x}
  def empty=b.isEmpty
 }
```

V Oui | R Non

Une façon d'implanter les contrats en Scala

Le contrat à définir	Le code Scala correspondant
`trait IntQueue {` `  def empty:Boolean`	`trait IntQueue{` `  def empty:Boolean`

`  def get:Int`
`    require( !empty)`
	`  def get:Int ={`
	`    require(!empty)`
	`    getIMP`
	`  }`
	
	`  protected def getIMP:Int`

`  def put(x:Int)`
`    ensuring( !empty)`
`}`
	`  def put(x:Int)={`
	`    putIMP(x:Int)`
	`  } ensuring (!empty)`
	
	`  protected def putIMP(x:Int):Unit`
	`}`

Une façon d'implanter les contrats en Scala (II)

Le code de la classe `MyIntQueue` implantant `IntQueue` devient :

Version sans contrats

```
class MyQueue extends IntQueue{
  private var b= List[Int]()

  def get= {
    val h=b(0)
    b=b.drop(1)
    h}

  def put(x:Int)= {
    b=b:+x
  }

  def empty=b.isEmpty
 }
```

Version avec contrats

```
class MyQueue extends IntQueue{
  private var b= List[Int]()

  protected def getIMP= {
    val h=b(0)
    b=b.drop(1)
    h}

  protected def putIMP(x:Int)= {
    b=b:+x
  }

  def empty=b.isEmpty
 }
```


Une façon d'implanter les contrats en Scala (III)

Exercice 6

Définir un contrat `ArrayCopy` qui propose une fonction `copy(t1:Array[Char],t2:Array[Char]):Unit` qui copie tous les caractères d'un tableau `t1` dans un tableau `t2`. Dans le contrat, on se protègera de tous les cas de figure suivants :

- *les tableaux peuvent être `null`*
- *`t2` est trop petit pour recevoir tous les éléments de `t1`*
- *la copie n'a pas été réalisée convenablement*
- *`t1` a été modifié par la copie*
- *...*

Remarque 4 (Invariants de classes)

En Scala il est également possible de définir des invariants de classes mais la construction est plus complexe.

Plan

Des formules logiques *prouvées* sur des programmes

- Au lieu de tester une formule sur un programme, on peut la prouver
- Démo de l'assistant de preuve Isabelle/HOL
- Une preuve remplace une infinité de tests !
- Principe émergent dans le dommaine du logiciel critique
  - CompCert : compilateur C certifié pour Airbus
  - METEOR : logiciel embarqué de la Ligne 14 du métro pour la RATP
- Vous en entendrez parler dans d'autres UE de
  - L3 : LOG, Prog2
  - M1 : ACF, MFDS, MVFA