Programmation Objet : Concepts Avancés
Cours 7 : Programmation générique

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

28 novembre 2008

Introduction au polymorphisme

Polymorphisme : définition

- « Poly- » : plusieurs, « morphisme » : formes.
- Un composant est polymorphe si on peut l'utiliser sous plusieurs formes ou si il utilise des composants de plusieurs formes.
- Cardelli et Wegner distinguent plusieurs formes de polymorphismes :
  - Universel
    - Paramétrique – le composant est paramétré par des types
    - Inclusion – le composant est compatible avec tous les sous-types d'un type
  - *Ad hoc*
    - Surcharge – plusieurs versions du composant en fonction du type d'entrée
    - Coercion – le composant peut se convertir dans un autre type
- Dans le cas du polymorphisme *ad hoc*, le composant travaille (ou semble travailler) sur des types di□érents avec un comportement éventuellement non uniforme.
- Dans le cas du polymorphisme *universel*, le composant travaille de manière uniforme sur les données quelque soit leur type.
- (nous ra□nerons cette définition tout au long de ce cours)

Programmation générique

- La programmation générique consiste à développer des composants paramétrés par des types.
- L'objectif principal est la réutilisabilité du développement.
- Le polymorphisme est le mécanisme central pour atteindre cet objectif.

Polymorphisme d'inclusion

Sous-typage

- On dit qu'un type T est un sous-type d'un type U si toute valeur de type T est utilisable dans un contexte attendant une valeur de type U.
- On notera cette relation :

$$T \prec: U$$

Covariance

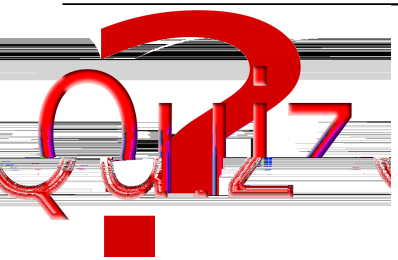
Propriété

Si $A \prec: A'$ et $B \prec: B'$ alors $A \times B \prec: A' \times B'$.

- Le type des paires formées de deux types A et B qui sont des sous-types de A' et B' est un sous-type des paires formées des deux types A' et B'.
- On dit que l'opérateur de type de formation des paires est covariant.

Le type des fonctions

- Soient A, B, A', B' tels que $A \prec: A'$ et $B \prec: B'$.



Est-ce que $A \to B \prec: A' \to B'$?

Contravariance

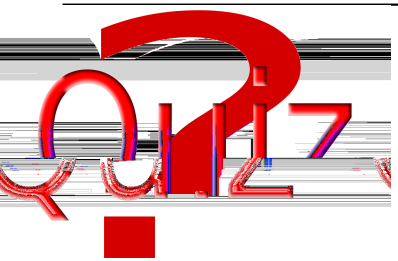
Propriété

Si $A \prec: A'$ et $B \prec: B'$ alors $A' \to B \prec: A \to B'$.

- On dit que l'opérateur de type des fonctions est contravariant sur son domaine.
- Il est covariant sur son codomaine.

Le type des références

- Soient A et B tels que $A \prec: B$.
- Le type des références sur A est muni de deux opérations :
  - write : ref $A \rightarrow A \rightarrow$ unit
  - read : ref $A \rightarrow A$



L'opérateur de type des références est-il covariant ou contravariant ?

Invariance

- Le type A devant être utilisé en position covariante et en position contravariante, l'opérateur de type ref est invariant.

Héritage et sous-typage

- On confond souvent héritage et sous-typage.
- Quand une classe C hérite d'une classe C' alors on a $C \prec: C'$.
- Par contre, l'implication inverse n'est pas nécessaire pour assurer la sûreté du typage.
- Le sous-typage structurel illustre cette remarque.

Contravariance des arguments des méthodes

- En première approximation, on peut dire qu'un objet est un n-uplet de fonctions (ses méthodes).
- Lorsqu'une classe C hérite d'une classe C', elle doit être un sous-type de C' donc, en particulier, chaque méthode de C doit avoir pour type un sous-type de la méthode correspondante de C'.
- Par exemple, si on a :

```
class A : {
   msg : T → U
}
```

```
class B extends A : {
   msg : T' → U'
}
```

alors $T' \prec: T$ et $U \prec: U'$

- Les arguments des méthodes doivent donc évoluer de façon contravariante à travers l'héritage.

Problème du type de *self*

- La signature de la méthode clone définie dans la classe Object est :

  public Object clone () ;

- En Java, le type de this est celui de la classe en cours de définition.
- Il n'est donc pas possible d'éviter au programmeur un changement explicite de type.
- Par exemple,

  A my_a = new A ;
  A copy_of_a = (A) my_a.clone () ;

Donner un type plus général à this

- Pour typer correctement la méthode clone, il faut que this ait le type de la future (sous-)classe avec laquelle il sera instancié.
- Les langages O'Caml et Scala o□rent ce mécanisme. [DEMONSTRATION]

En O'Caml

```
class my_class =
object (self)
   method clone = self
end

class my_class_bis =
object
   inherit my_class
end

(* Réponse du toplevel. *)
class my_class : object ('α) method clone : 'α end
class my_class_bis : object ('α) method clone : 'α end
```

En Scala

```
class MyClass {
  def clone_me () : this.type = this
}

class MyClassBis extends MyClass {}

object test {
  def main (args : Array[String]) = {
    var a = new MyClassBis;
    var b : MyClassBis = a.clone_me ();
  }
}
```

- Le type de b est « b.type » qui vaut « MyClassBis ».

Problème des méthodes binaires

Qu'arrive-t-il si on veut utiliser this.type
comme type de l'argument d'une méthode ?

Polymorphisme paramétrique

Factorisons !

```
class IntStack {
  case class EmptyStack extends Throwable;

  var elements : List[Int] = List ()

  def push (x: Int) =
    this.elements = (x :: this.elements)

  def pop () =
    elements match {
      case Nil => throw (new EmptyStack ());
      case x :: xs => this.elements = xs; x
    }
}
```

```
class StringStack {
  case class EmptyStack extends Throwable;

  var elements : List[String] = List ()

  def push (x: String) =
    this.elements = (x :: this.elements)

  def pop () =
    elements match {
      case Nil => throw (new EmptyStack ());
      case x :: xs => this.elements = xs; x
    }
}
```

Abstraire par rapport à un type

```
class Stack[T] {
  case class EmptyStack extends Throwable;
  var elements : List[T] = List ()
  def push (x: T) = this.elements = (x :: this.elements)
  def pop () =
    elements match {
      case Nil => throw (new EmptyStackException ());
      case x :: xs => this.elements = xs; x
    }
}
```

- Le type T est un paramètre.
- Stack est une classe paramétrée.
- Elle représente une famille *a priori* infinie de classes.

Instancier un paramètre de type

```
val int_s = new Stack[Int];
val string_s = new Stack[String];

int_s.push (0);
int_s.push (42);
println (int_s.pop ());
try {
   string_s.pop ();
} catch {
   case string_s.EmptyStack () =>
      println ("Do not pop from an empty stack, Luke.");
}
```

- On peut instancier explicitement le paramètre T.
- Remarquez la finesse du typage de Scala, string_s.EmptyStack est différent de int_s.EmptyStack !

Instanciation implicite grâce à l'inférence des types

- L'inférence de type de `Scala` permet de se passer de nombreuses instanciations.
- Soit une méthode paramétrée par un type T :

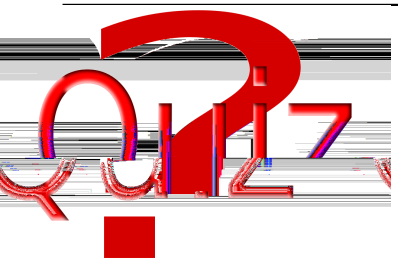
```
def push_twice[T] (s : Stack[T], x : T) = {
      s.push (x) ;
      s.push (x) ;
}
```

- Dans la plupart des cas, le type des arguments suffit à choisir la bonne instanciation de T.

```
/* T = Int est déduit automatiquement du type des arguments. */
push_twice (int_s, 42) ;
```

Liens entre sous-typage et abstraction de type

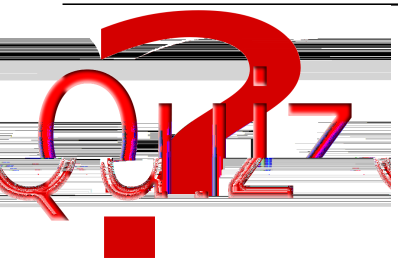
- Remarquons sans plus tarder que $A \prec: B$ **n'implique pas** $C[A] \prec: C[B]$.



Pourquoi ?

Et pourtant en Java...

```
public class A {}
public class B extends A {}
public class C extends A {}
public void do_it () {
   B[] tb = new B[42];
   A[] ta = v;
   ta [0] = new C ();
}
```



Que fait ce programme Java ?

Laxisme coûteux de Java

- Les tableaux de Java sont co-variants par rapport à leur paramètre de type !
- Le programme précédent est donc accepté par le typeur de Java.
- Pourtant, il stocke une instance de la classe C là où on attend une instance de la classe B alors que C n'est pas un sous-type de C !
- Pour résoudre ce problème de sûreté, Java introduit un test dynamique dans la méthode de mis-à-jour d'une cellule d'un tableau : le type de la valeur a insérer dans le tableau doit être un sous-type du type qui a servi à construire le tableau initialement.

$\Rightarrow$ Un gros problème de sûreté et de performance sur une opération standard !

$\Rightarrow$ Mieux vaut utiliser la classe paramétrée Array fournie par les versions récentes de Java.

Chez les autres . . .

- En O'Caml, C++ et Scala les paramètres de type sont invariants (par défaut).
- Eiffel suit la même philosophie que Java et permet la covariance des paramètres de type.

Et si on était sage ?

```
class HeadGetter[a] (list : List[a]) {
   class NoMoreElements extends Throwable ;

   def head () : a =
      list match {
         case Nil => throw (new NoMoreElements ())
         case x :: xs => x
      }
}

class A {}
class B extends A {}

object test {
   def main (args : Array[String]) = {
      val s : HeadGetter[A] = new HeadGetter[B] (List (new B ())) ;
   }
}
```

Y-a-t'il un problème de sûreté dans ce programme ?

Et si on était sage ?

class

Annotation de variances

- En O'Caml et en Scala, on peut annoter les paramètres de type en « promettant » d'utiliser un type en position contravariante ou covariante.

Cas d'utilisation contravariante sûre

```
abstract class Stack[a] {
  def push(x: a): Stack[a] = new NonEmptyStack[a](x, this)
  def isEmpty: boolean
  def top: a
  def pop: Stack[a]
}
class EmptyStack[a] extends Stack[a] {
  def isEmpty = true
  def top = throw new Error("EmptyStack.top")
  def pop = throw new Error("EmptyStack.pop")
}
class NonEmptyStack[a](elem: a, rest: Stack[a]) extends Stack[a] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

(source : *Programming in Scala* – Martin Odersky)

- Ici, le type a est utilisé en position contravariante et covariante.
- Il est donc impossible de mettre une annotation de covariance sur le type a.
- (la solution viendra un peu plus loin dans le cours.)

Type abstrait

- Nous avons vu qu'il est possible de définir un type à l'intérieur d'une classe.
- En Scala et en C++, la définition de ce type peut être inconnue.
- Un tel type est dit abstrait.
- En O'Caml, les types abstraits sont définis par le biais de modules.

Implémentation d'un type abstrait en C++

```
template ⟨class T⟩
struct element_traits;

template ⟨class SelfType⟩
class Buffer {
   typedef typename element_traits⟨SelfType⟩::T element_t;
   virtual element_t pop () = 0;
};
```

- L'utilisation du terme « traits » est propre au vocabulaire de C++ mais ne correspond pas aux traits vu dans le cours sur Scala.
- Nous utiliserons plutôt le terme de « type associé » par la suite.

Implémentation d'un type abstrait en Scala

```
abstract class Bu□er {
  type T
  val element: T
}
```

- Il su□t de ne pas donner la définition du type T pour le rendre abstrait.

Implémentation d'un type abstrait en O'Caml

```
module type BufferSig = sig
  type t
  class virtual buffer : object
    method virtual element : t
  end
end

module MakeBuffer (S : sig type t end) =
struct
  type t = S.t
  class virtual buffer = object
    method virtual element : t
  end
end

module Buffer : BufferSig = MakeBuffer (struct type t end)
```

Type partiellement abstrait

- En O'Caml et en Scala, on peut donner une information partielle sur un type.
- Un type partiellement abstrait permet de ne dévoiler au client que la partie du type nécessaire à son utilisation.
- Il s'agit encore de favoriser l'encapsulation.

Implémentation d'un type partiellement abstrait en Scala

```
abstract class SeqBu□er extends Bu□er {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

- Cette définition signifie qu'il existe un type U abstrait et que le type des éléments du tampon se comportent comme une séquence (on peut par exemple la parcourir.
- On en dit pas plus !

Implémentation d'un type partiellement abstrait en O'CamI

```
module type SeqBu□erType = sig
  type u
  type t = u seq
  class virtual seq_bu□er : object
    method virtual element : t
    method length : int
  end
end
```

```
module SeqBu□er : SeqBu□erType
struct
  type u
  include MakeBu□er(struct
                      type t = u seq
                    end)
  class virtual seq_bu□er =
  object (self)
    inherit bu□er
    method length =
      self#element#length
  end
end
```

Spécification totale au moment de l'instanciation

- En Scala, on peut retardé la définition complète d'un type abstrait jusqu'au moment de l'instanciation de la classe.

```
abstract class SeqBu□er extends Bu□er {
  type U
  type T <: Seq[U]
  def length = element.length
}

object test {
  def main (args : Array[String]) = {
    val buf =
      new SeqBu□er
      {
        type U = Int;
        type T = List[U];
        val element = List(37, 42);
      };
  }
}
```

Abstraction de type et type abstrait

```
abstract class Bu□er[+T] {
  val element: T
}
abstract class SeqBu□er[U, +T <: Seq[U]] extends Bu□er[T] {
  def length = element.length
}
object AbstractTypeTest2 extends Application {
  def newIntSeqBuf(e1: Int, e2: Int): SeqBu□er[Int, Seq[Int]] =
    new SeqBu□er[Int, List[Int]] {
      val element = List(e1, e2)
    }
  val buf = newIntSeqBuf(7, 8)
  println("length = " + buf.length)
  println("content = " + buf.element)
}
```

(source : `http://www.scala-lang.org/node/105`)

- On peut très souvent (mais pas toujours) transformer une classe paramétrée en une classe équivalente munie de types abstraits.
- Nous allons voir la notation T <: Seq[U] un peu plus loin dans ce cours.

Différentes implémentations du polymorphisme paramétrique

- Si les di□érents implémentations des classes paramétrées semblent similaires dans les langages objets modernes (C#, Java, Scala, C++, O'Caml, ...), cette ressemblance est superficielle si on y regarde d'un peu plus près.

Instanciation = Duplication en C++

- L'implémentation des classes templates de C++ s'apparente à un copier-coller du code.
- L'instanciation duplique le code de la classe template et remplace les paramètres par leurs valeurs.
- Aucune vérification n'est faite sur la classe template !
- La vérification est faite une fois la classe instanciée.

$\Rightarrow$ Ceci pose d'énorme problème pour fournir des garanties à l'utilisateur d'une bibliothèque générique.

$\Rightarrow$ Le développeur d'une telle bibliothèque doit instancier manuellement les classes et les méthodes pour vérifier leur bon fonctionnement.

- La duplication du code permet certaines optimisations (mise en ligne du code de méthodes du paramètre une fois qu'il est connu) mais la taille des exécutables peut devenir problématique.
- Enfin, le temps de compilation est augmenté puisqu'il faut ré-analyser la classe template à chaque instanciation !

Instanciation = Effacement des types en Java

- Pour éviter les duplications de code, Java réutilise toujours le même code pour toutes les instanciations d'une classe paramétrée.
- Pour ce faire, le compilateur e□ace les paramètres de type et remplace leurs occurences par le type Object.
- Java autorise le programmeur a faire référence à cette classe de la façon suivante (pour des raisons de compatibilité arrière) :

```
public static class A<T> {
   public boolean compare (T lhs, T rhs) {
      return (lhs.equals (rhs)) ;
   }
}
public static void main (String[] args) {
   A cmp = new A () ;
   cmp.compare (42, "foobar") ;
}
```

- Un avertissement est produit car bien souvent le programmeur a fait une erreur en ne précisant pas le type sur lequel il voulait travailler :

```
GenericsErasure.java: 11: warning: [unchecked] unchecked call to compare(T,T)
as a member of the raw type GenericsErasure.A
      cmp.compare (42, "foobar") ;
```

Instanciation = Effacement ou duplication en C#

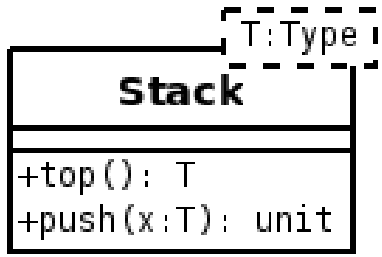
- En C#, le compilateur JIT (Just In Time) produit un code spécialisé lorsqu'une classe paramétrée est instanciée avec un type de valeur (int, bool, ...).
- Dans le cas des objets référencés, il se comporte comme Java(utilisation de la classe object).

En Scala et en O'Caml, à peu près comme Java

- Scala e□ectue un e□acement des types et réutilise le même code pour toute classe générique. Par contre, il n'autorise pas l'utilisation de cette classe.
- O'Caml utilise aussi le même code pour toutes les instanciations de la classe paramétrée : il s'agit de code polymorphe standard dans ce langage (qui s'appuie sur l'homogénéité de la représentation des données dans ce langage).

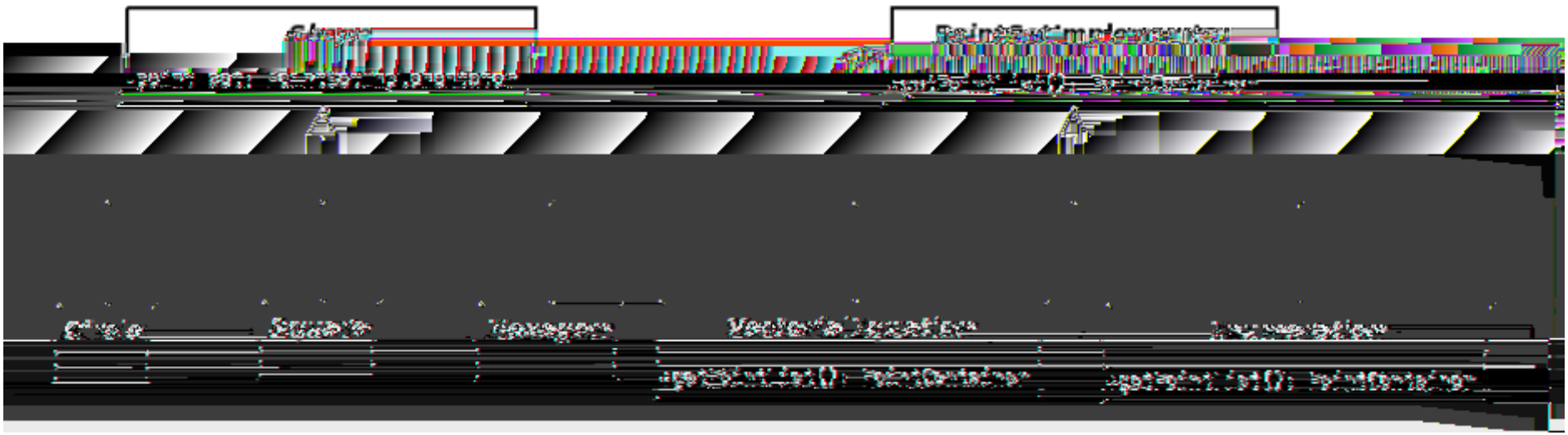
Des patrons de conception génériques

- UML fournit une notation pour les classes paramétrées.



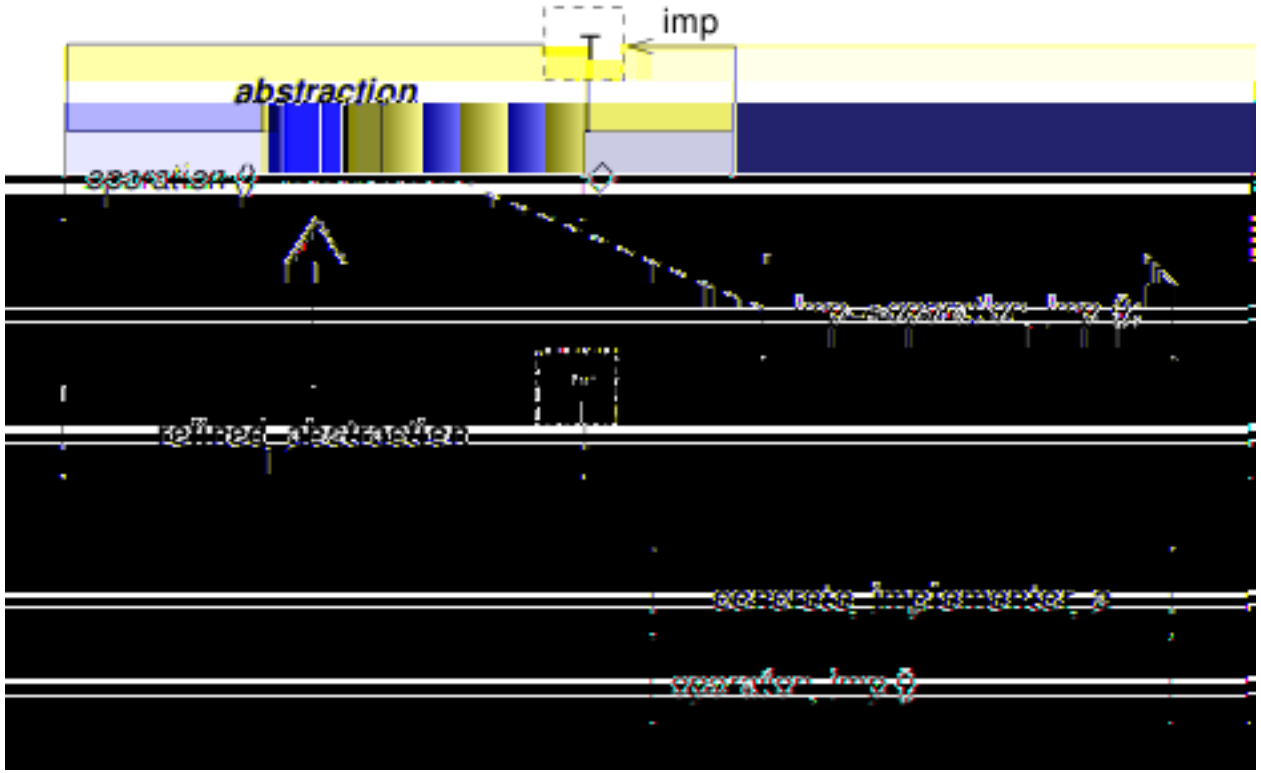
- Les patrons de conception que nous avons vus en cours ont été généralisés dans le cadre de la programmation générique.

Patron de conception « Pont » (souvenez-vous)



- Objectifs : découpler l'implémentation sous-jacent d'un objet abstrait de la hiérarchie « canonique » de l'objet.

Pont générique



- Conséquence : les implémentations ne sont pas nécessairement des classes filles d'une classe abstraite « implementor ».

Solution pour implémenter le « sujet-observateur » générique

```
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer

  abstract class Subject { self: S =>
    private var observers : List[O] = List ()
    def subscribe (obs : O) =
      observers = obs :: observers
    def publish =
      for (val obs ← observers) obs.notify (this)
  }

  trait Observer {
    def notify (sub: S) : unit
  }
}
```

- La classe SubjectObserver force la synchronisation des types des observateurs et des sujets.

Polymorphisme paramétrique borné

Comparons C++ et Scala

```
template ⟨class T⟩
struct num {
   num (T _x) : x (_x) {}
   T x;
   bool is_zero () { return (this→x.is_zero ()); }
   T max (T y) { return ((this→x ⟨ y)? y : this→x); }
};
```

Valide en C++

```
class num[T] (x: T) {
   def is_zero () = x.is_zero ()
   def max (y : T) = if (x > y) x else y
}
```

Refusé en Scala :

```
max.scala:2: error: value is_zero is not a member of T
   def is_zero () = x.is_zero ()

max.scala:3: error: value > is not a member of T
   def max (y : T) = if (x > y) x else y
```

- C++ ne rejetera la classe qu'au moment où on l'instanciera avec une classe ne fournissant pas l'opérateur "<" ou la méthode "is_zero".
- Les hypothèses sur le type T ne sont pas explicitées en C++.

Polymorphisme borné

- En Scala, O'Caml, Java et C#, il est possible d'imposer des contraintes sur le paramètre de type.
- On peut par exemple exprimé que le paramètre doit être un sous-type d'une certaine classe ou implémenter une interface.

```
trait ZeroTestable {
  def is_zero () : Boolean
}

class num[T <: ZeroTestable] (x: T) {
  def is_zero () = x.is_zero ()
  def max (y : T) = if (x > y) x else y
}
```

- Il ne reste alors plus qu'une erreur ...

Polymorphisme borné

- Peut-on utiliser le même procédé pour parler de l'opérateur “ > ” ?

```
trait ZeroTestable {
  def is_zero () : Boolean
}

trait MyOrd {
  def > (other :???) : Boolean
}

class num[T <: ZeroTestable with MyOrd] (x: T) {
  def is_zero () = x.is_zero ()
  def max (y : T) = if (x > y) x else y
}
```

Quel type donner à l'argument “other” ?

Implémentation d'un protocole objet

- L'utilisation conjointe des traits et des paramètres de type permet d'exprimer des hypothèses récursives sur les paramètres de type !

```
trait MyOrd[t <: MyOrd[t]] {
  def > (other : t) : Boolean
}

class num[T <: ZeroTestable with MyOrd[T] ] (x: T) {
  def is_zero () = x.is_zero ()
  def max (y : T) = if (x > y) x else y
}

class a (val x : Int) extends ZeroTestable with MyOrd[a] {
  def > (other: a) = other.x == x
  def is_zero () = x == 0
}

object test {
  def main (args: Array[String]) {
    val x = new num[a] (new a (0));
    val z = x.max (new a (42));
  }
}
```

Polymorphisme paramétrique encadré

Retour sur le problème de la contravariance

```
abstract class Stack[a] {
  def push(x: a): Stack[a] = new NonEmptyStack[a](x, this)
  def isEmpty: boolean
  def top: a
  def pop: Stack[a]
}
class EmptyStack[a] extends Stack[a] {
  // ...
}
class NonEmptyStack[a](elem: a, rest: Stack[a]) extends Stack[a] {
  // ...
}
class A {}
class B extends A {}
object test {
  def main (args: Array[String]) {
    val l = new NonEmptyStack[B] (new B (), new EmptyStack[B])
    l.push (new A ());
  }
}
```

Comment se programme pourrait être bien typé ?

Affaiblissement du type de Stack

- En Scala, on peut donner une borne inférieure à un paramètre de type :

```
def push[b >: a](x: b): Stack[b] = new NonEmptyStack[b](x, this)
```

- Cela signifie qu'on a le droit de stocker des éléments de type a dans une pile de type Stack[b] à condition de produire une pile de type Stack[a], c'est-à-dire en a□aiblissant le type de la pile.

Polymorphisme intentionnel

Polymorphisme « adaptatif »

- Avec le polymorphisme paramétrique, un composant générique se comporte uniformément sur un ensemble de types (soit correspondant à l'ensemble de tous les types ou à un sous-ensemble borné de ces types).
- Avec le polymorphisme *ad hoc* ou le polymorphisme intentionnel, un composant générique s'applique aussi à un ensemble de types (plus ou moins restreint) mais peut avoir un comportement dépendant du type sur lequel on l'instancie.
- Il s'agit donc de donner au programmeur un moyen d'inspecter un paramètre de type et de donner plusieurs définitions en fonction de son observation.

Premier exemple : le polymorphisme *ad hoc* de C++

```
void my_function (int x) { /* ... */ }

void my_function (bool b) { /* ... */ }

void client () {
   my_function (42) ;
   my_function (true) ;
}
```

- Du point de vue du client, on a l'impression qu'il existe une unique fonction.
- Par observation du type de l'argument, le compilateur choisit un code spécifique.
- Une expressivité limitée : on peut seulement discriminer sur des hiérarchies de classe ou des types de base.

Second exemple : les fonctions templates en C++

```
template ⟨class T⟩
void my_function (T x) { std::cout ≪ "General type" ≪ std::endl; }

template ⟨class T⟩
void my_function (T* x) { std::cout ≪ "Pointer type" ≪ std::endl; }

template ⟨class T⟩
void my_function (const std::list⟨T⟩& x) { std::cout ≪ "List type" ≪ std::endl; }

template ⟨⟩
void my_function (double x) { std::cout ≪ "Convertible to double type" ≪ std::endl; }

int main (int argc, char** argv) {
   my_function ("On peut rire de tout, mais pas avec n'importe qui.");
   my_function (true);
   my_function (std::list⟨int⟩());
   my_function (42.);
   my_function (0);
}
```

Qu'affiche ce programme ?

Des règles de résolution pas toujours évidentes

- Il est nécessaire d'inspecter la norme du C++ pour savoir exactement quelle fonction est appelée ...
- Cette complexité est la conséquence de l'absence d'un type unique pour toute expression du langage doublé de la présence de conversion implicite.

Types associés

- Cette spécialisation des composants par rapport à des types est souvent utilisée pour coder une relation entre deux types.
- Exemples :
  - Le type d'un itérateur sur une valeur de type std::vector⟨int⟩ est int*.
  - Le type d'un itérateur sur une liste de type List⟨int⟩ est ListNode⟨int⟩.
  - L'implémentation d'un ensemble d'entiers de type std::set⟨int⟩ est std::vector⟨bool⟩.
  - Le type des événements d'un gestionnaire de fenêtres est WindowEvents
  - ...
- Implémenter une relation entre deux types permet d'écrire des algorithmes génériques :

```
for (typename std::list<int>::iterator x = l.begin(); x != l.end(); ++x) {
  ...
}
```

Fonctions des types dans les types

- Ce procédé de spécialisation peut être utilisé pour écrire des fonctions évaluées au moment de la compilation.
- En e□et, les types peuvent être vus comme des données manipulées par le compilateur.
- Observer des types et instancier des types permet de coder de véritables fonctions.
- Le système de *template*s de C++ est turing-complet !
- Certaines bibliothèques utilisent ce procédé pour e□ectuer certains pré-calculs (pour optimisations) à l'aide du compilateur.
- Il s'agit de méta-programmation.

Exemple de méta-programmation en C++

```
template ⟨unsigned n⟩
struct factorial {
   enum { result = n * factorial⟨n-1⟩::result };
};

template ⟨⟩
struct factorial⟨0⟩ {
   enum { result = 1 };
};

int main (int argc, char** argv) {
   int t [factorial⟨5⟩::result];
}
```

Méta-programmation en C++

- Le langage C++ n'a pas été conçu pour une telle utilisation des templates.

$\Rightarrow$ Ces mécanismes sont très longs à compiler (car ils instancient énormément de types) et sont très di□ciles à maintenir (les messages d'erreur sont horribles).

- Par contre, certains langages commencent à intégrer cette programmation multi-niveau dans leur conception (cf. MetaOCaml par exemple).

Fonctions des types vers les valeurs

- On décompose le mécanisme du polymorphisme *ad hoc* en deux étapes :
  1. Observer un type ;
  2. Choisir une expression à évaluer en fonction de ce type.
- Le polymorphisme *ad hoc* de C++ choisit une fonction à évaluer.
- En Haskell et en Scala, il est possible de choisir une valeur à utiliser en tant qu'argument d'une fonction en fonction du types des autres arguments.
- Un tel argument est dit implicite.
- En Scala, seuls des objets spéciaux notés implicites peuvent être ainsi automatiquement inférés.

Exemple

```
abstract class SemiGroup[A] {
    def add(x: A, y: A): A
}
abstract class Monoid[A] extends SemiGroup[A] {
   def unit: A
}

object ImplicitTest extends Application {

    implicit object StringMonoid extends Monoid[String] {
       def add(x: String, y: String): String = x concat y
       def unit: String = ""
    }

    implicit object IntMonoid extends Monoid[int] {
       def add(x: Int, y: Int): Int = x + y
       def unit: Int = 0
    }

    def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
       if (xs.isEmpty) m.unit
       else m.add(xs.head, sum(xs.tail))

    println(sum(List(1, 2, 3)))
    println(sum(List("a", "b", "c")))
}
```

(source : `http://www.scala-lang.org/node/114`)