

8.1 型パラメータの境界

クラスをジェネリックにする方法を知った今では、前に書いたクラスの中には一般化した方が自然なものがあります。たとえばクラス `IntSet` は、任意の要素型を持てるように一般化できます。やってみましょう。ジェネリックな集合の抽象クラスは簡単に書けます。

```
abstract class Set[A] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

しかしもし二分木として実装し続けたいなら、問題に直面します。メソッド `contains` と `incl` はどちらも、要素をメソッド `<` と `>` を使って比較します。`IntSet` については、これは OK です。なぜなら `Int` はそれら 2 つのメソッドを持っているからです。しかし任意の型パラメータ `a` に対しては、それは保証できません。したがって先の実装、たとえば `contains` ではコンパイルエラーが生じるでしょう。

```
def contains(x: Int): Boolean =
  if (x < elem) left contains x
  ^ < not a member of type A.
```

問題を解決する一つの方法は、型 `A` と置き換え可能な正当な型を、正しい型のメソッド `<` と `>` を持つ型だけに制限することです。Scala 標準ライブラリには、型 `A` の値と (`<` と `>` によって) 比較可能な値を表現する、トレイト `Ordered[A]` があります。このトレイトは次のように定義されています。

```
/** データを完全に順序づけるためのクラス */
trait Ordered[A] {
  /** this をオペランド that と比較した結果。
   * returns 'x' where
   *   x < 0   iff this < that
   *   x == 0  iff this == that
   *   x > 0   iff this > that
   */
  def compare(that: A): Int
  def < (that: A): Boolean = (this compare that) < 0
  def > (that: A): Boolean = (this compare that) > 0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

型が `Ordered` のサブタイプであることを要求することで、互換性を強制できます。これは `Set` の型パラメータに上界境界(upper bound)を与えることでなされます。

```
trait Set[A <: Ordered[A]] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

パラメータ宣言 `A <: Ordered[A]` は型パラメータとしての `A` を、`A` は `Ordered[A]` のサブタイプでなくてはならない、として導入します。すなわち、その値は同じ型の値と比較可能でなくてはなりません。

この制限によって、ジェネリックな集合の抽象化を、前の `IntSet` の場合と同じように実装できます。

```
class EmptySet[A <: Ordered[A]] extends Set[A] {
  def contains(x: A): Boolean = false
  def incl(x: A): Set[A] = new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}
class NonEmptySet[A <: Ordered[A]]
  (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  def contains(x: A): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
}
```

```
else true
def incl(x: A): Set[A] =
  if (x < elem) new NonEmptySet(elem, left incl x, right)
  else if (x > elem) new NonEmptySet(elem, left, right incl x)
  else this
}
```

オブジェクト生成 `new NonEmptySet(...)` において、型引数を書いていないことに注意して下さい。多相的メソッドと同様に、コンストラクタ呼び出しで型引数がかかれていない時、それは値引数と期待される結果型(戻り値型)から推論されます。

以下はジェネリックな集合の抽象化を使う例です。まづ、`Ordered` のサブクラスを次のように作りましょう。

```
case class Num(value: Double) extends Ordered[Num] {
  def compare(that: Num): Int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

すると

```
val s = new EmptySet[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(1.5))
```

これは OK です。なぜなら型 `Num` はトレイト `Ordered[Num]` を実装しているからです。しかし次の例はエラーになります。

```
val s = new EmptySet[java.io.File]
      ^ java.io.File does not conform to type
      parameter bound Ordered[java.io.File].
```

型パラメータ境界の問題は、前もって考えておくべきことです。もし `Num` を `Ordered` のサブクラスとして宣言していなければ、`Num` を集合の要素として使用できません。同様に、Java から継承した `Int`, `Double`, `String` といった型は `Ordered` のサブクラスではないので、それらの型の値は集合の要素として使用できません。

それらの型を要素として許すもっと柔軟なデザインがあります。**可視境界** (view bound) を今までに見てきた単純な型境界の代わりに使うことです。先の例でこれによって起こる変更は、単に型パラメータが次のようになることです。

```
trait Set[A <% Ordered[A]] ...
class EmptySet[A <% Ordered[A]] ...
class NonEmptySet[A <% Ordered[A]] ...
```

可視境界 `<%` は通常の上境界 `<` より弱いです。可視境界をもつ型パラメータ節 `[A <% T]` は単に、境界づけられる型 `A` が境界となる型 `T` へ、暗黙の型変換を使って **変換可能** なことだけを指定します。

Scala ライブラリは、プリミティブ型や `String` を含む数多くの型について、暗黙の型変換を事前に定義しています。したがって、再デザインされた集合の抽象化は、これらの型についてもインスタンス化できます。暗黙の変換と可視境界に関する更なる説明は第 15 章にあります。

[前ページ](#) [8章 目次](#) [次ページ](#)

- bound, type bound, view bound の良い訳語が判らない!... -- tmiya (2008-05-19 00:34:40)
- 束縛、っていつっちゃうとやっばまずいですかね。「縛り」の方がまし? -- a (2009-07-25 03:51:29)

名前:	<input type="text"/>
コメント:	<input type="text"/>

投稿