The Type Astronaut's Guide to Shapeless

Dave Gurnell
foreword by Miles Sabin

underscore

The Type Astronaut's Guide to Shapeless

January 2017

Published by Underscore Consulting LLP, Brighton, UK.

Traduit de l'anglais pas Etienne Couritas

Relecture Audrey Carpentras

Contents

Avant-propos

Retour en début 2011, alors que je commençais à réaliser des expériences de programmation générique qui deviendront plus tard shapeless. Jamais je n'aurais pensé que cinq ans plus tard, shapeless deviendrait une librairie tant utilisée. Je suis profondément reconnaissant envers les personnes qui on cru en moi et qui ont ajouté shapeless à leurs dépendances. Le vote de confiance que cela représente apporte un véritable coup de boost à tous les projets opensource. Je suis aussi reconnaissant envers les nombreuses personnes qui ont contribué à shapeless au fil des années, quatre-vingt-une au moment où j'écris ces lignes. Sans leur aide, shapeless serait une librairie bien moins intéressante.

Malgré ces points positifs, shapeless a souffert d'une carence qui caractérise les projets open source : le manque de documentation compréhensible, juste et accessible. La reponsabilité me revient en partie : bien que conscient de ce problème, je n'ai jamais trouvé le temps d'y remédier. shapeless a été sauvé dans une certaine mesure par la performance héroïque de Travis Brown sur stack Overflow ainsi que par de nombreuses personnes qui ont organisé des talks et des workshops sur shapeless (je voudrais remercier tout particulierement Sam Halliday pour son workshop « Shapeless for Mortals »).

Mais Dave Gurnell est parvenu à changer tout ça : nous avons maintenant ce merveilleux livre qui traite de manière aprofondie la partie la plus importante de shapeless : la déduction de *type class* par la programmation générique. Il a regroupé des fragments de folklore et de documentation, il m'a poussé à la réflexion, et il a transformé cet enchevêtrement impénétrable en quelque-chose de clair, concis et tres concret. Avec un peu de chance, il sera capable de

confirmer mon affirmation selon laquelle le cœur de shapeless est une librairie très simple qui incarne un ensemble de concepts tout aussi simples.

Merci Dave, tu nous rends à tous un grand service.

Miles Sabin
Créateur de shapeless

Chapter 1

Introduction

Ce livre est un guide d'utilisation de shapeless[1], une bibliotèque de *programmation générique* en Scala. Shapeless est une vaste bibliotèque ; plutôt que tout couvrir, nous allons nous concentrer sur quelques cas d'utilisation intéréssants, puis les utiliser pour recréer un tableau des outils et patterns existants.

Avant de commencer, il convient d'expliquer la programmation générique et la raison pour laquelle shapeless est si intéressant pour les développeurs Scala.

1.1 La programmation générique, qu'est-ce que c'est ?

Les types sont utiles car ils sont spécifiques : ils nous aident a comprendre comment les différents morceaux de code s'imbriquent les uns aux autres, ils nous aident à éviter les bugs et nous conduisent vers une solution lorsque l'on code.

Pourtant, parfois, les types sont *trop* spécifiques. Il y a des situations où l'on veut pouvoir profiter des similarités entres les types pour eviter les répétitions. Prenons les définitions suivantes comme exemple :

[1] https://github.com/milessabin/shapeless

```
case class Employee(name: String, number: Int, manager: Boolean)

case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

Ces deux *case classes* représentent des données différentes mais elles on des points communs évidents : elles possèdent toutes deux des champs du même type. Supposons que nous voulons implémenter une opération générique, par exemple les sérialiser en un fichier CSV. En dépit des similarités, nous devons écrire une méthode de sérialisation pour chaque type :

```
def employeeCsv(e: Employee): List[String] =
  List(e.name, e.number.toString, e.manager.toString)

def iceCreamCsv(c: IceCream): List[String] =
  List(c.name, c.numCherries.toString, c.inCone.toString)
```

La programmation générique consiste à venir à bout de ce genre de différences. Shapeless met a disposition un moyen pratique de convertir des types spécifiques en une représentation générique, qui sont ensuite manipulables avec un code commun.

Par exemple, nous pouvons utiliser le code suivant pour convertir les `Employee` et les `IceCream` en valeurs du même type. Ne vous inquiétez pas si vous ne comprenez pas encore cet exemple, ce sujet sera traité plus tard.

```
import shapeless._
```

```
val genericEmployee = Generic[Employee].to(Employee("Dave", 123, false
    ))
// genericEmployee: shapeless.::[String,shapeless.::[Int,shapeless.::[
    Boolean,shapeless.HNil]]] = Dave :: 123 :: false :: HNil

val genericIceCream = Generic[IceCream].to(IceCream("Sundae", 1, false
    ))
// genericIceCream: shapeless.::[String,shapeless.::[Int,shapeless.::[
    Boolean,shapeless.HNil]]] = Sundae :: 1 :: false :: HNil
```

Les deux valeurs sont maintenant du même type. Elle sont toutes les deux des liste hétérogène (des `HLists`) contenant une `String`, un `Int`, et un `Boolean`.

Nous jetterons bientôt un œil au `HLists` et à l'importance de leur rôle. Pour l'instant, le plus intéressant est que nous pouvons sérialiser chaque valeur avec une seule fonction.

```
def genericCsv(gen: String :: Int :: Boolean :: HNil): List[String] =
  List(gen(0), gen(1).toString, gen(2).toString)
```

```
genericCsv(genericEmployee)
// res2: List[String] = List(Dave, 123, false)

genericCsv(genericIceCream)
// res3: List[String] = List(Sundae, 1, false)
```

Cet exemple est basique mais il pointe du doigt la nature de la programmation générique. Nous reformulons les problèmes pour pouvoir les résoudre avec des constructions génériques et ainsi écrire de petit bloc de code qui fonctionnent avec une large variété de types. Programmer avec shapeless nous permet d'éliminer une grande quantité de `boilerplate`, de rendre les applications Scala plus faciles à lire, écrire et entretenir.

Votre curiosité a été piquée ? Alors c'est parti !

1.2 À propos du livre

Le livre est divisé en deux parties.

Dans la Partie I, nous introduisons la *déduction de* type class* (type class derivation), *qui permet de créer des instances de* type class* pour tous types de données algébriques avec quelques règles génériques pour tout matériel. La Partie I comprend quatre chapitres :

- Dans le chapitre 2, nous introduisons les *generic representations*, mais aussi la *type class* `Generic` de shapeless, qui permet de produire un encodage générique de n'importe quelle case class ou famille scellée.

- Dans le chapitre 3 on utilise `Generic` pour déduire une instance d'une de vos propres type class. On crée un exemple de *type class* pour encoder des données Scala en Comma Separated Values (CSV), mais la technique utilisée peut être adaptée à de nombreuses situations. Nous présentons également le type `Lazy` de shapeless, qui permet de manipuler les types de données récusives comme les listes ou les abres.

- Dans le chapitre 4, nous présentons les théories et les patterns de programmation nécessaires à la généralisation des techniques du chapitre précédent. On se penchera sur les types dépendants, les fonctions à type dépendant et la programmation au type level. Cela nous permet d'utiliser des applications plus avancées de shapeless.

- Dans le chapitre 5, nous présentons `LabelledGeneric`, une variante de `Generic` qui divulgue les champs et les noms des types dans sa repésentation générique. Nous présentons également une théorie supplémentaire : les types littéraux, les types singleton, les types fantôme et les type tagging. Nous illustrons `LabelledGeneric` en créant un encodeur *JSON* qui préserve les champs et les noms des types dans sa sortie.

Dans la Partie II, nous présentons les "ops type classes" fournies dans le package `shapeless.ops`. Les ops type classes constituent une large bibliotèque d'outils pour manipuler les représentations génériques. Plutôt que d'expliquer en détails chaque op, nous offrons une base théorique en trois chapitres :

- Dans le chapitre 6, nous proposons une présentation générale des ops type classes et fournissons un exemple qui relie plusieurs ops ensemble pour former un puissant outil de "migration de case class".

- Dans le chapitre 7, nous présentons *les fonctions polymorphique*, également connues sous le nom de `Polys`, et montrons comment les utiliser dans les ops type classes pour « mapper », « flat mappés » et « folder » les représentations génériques.

- Enfin, dans le chapitre 8 nous présentons le type `Nat` que shapeless utilise pour représenter les nombres naturels au type level. Nous

présentons plusieurs ops type classes associés, et utilisons `Nat` pour développer notre propre version de `Arbitrary` de Scalacheck.

1.3 Code source et exemples

Ce livre est opensource. Vous pouvez trouver les sources Markdown sur Github[2]. La communauté met constament à jour ce livre. Veillez donc à vérifier sur le repo Github que vous disposez de la dernière version.

Il y existe aussi des implémentations complètes des exemples principaux dans un repo compagnon[3]. Consultez le README pour plus de détails sur l'installation.

Les exemples utilisent shapeless 2.3.2 et Typelevel Scala 2.11.8+ ou Lightbend Scala 2.11.9+ / 2.12.1+.

1.4 Remerciements

Merci à Miles Sabin, Richard Dallaway, Noel Welsh, Travis Brown, et nos camarades de Github[4] pour leur contribution inestimable.

Un remerciement tout particulier à Sam Halliday pour sont excellent workshop. Shapeless for Mortals[5], qui nous a fourni l'inspiration initiale ainsi que la structure.

Enfin, merci à Rob Norris et à ses contributeurs pour le génial Tut[6], qui assure la compilation de nos exemples.

[2] https://github.com/underscoreio/shapeless-guide
[3] https://github.com/underscoreio/shapeless-guide-code
[4] https://github.com/underscoreio/shapeless-guide/graphs/contributors
[5] https://github.com/fommil/shapeless-for-mortals
[6] https://github.com/tpolecat/tut

Part I

Type class derivation

Chapter 2

Type de données algébriques et représentation générique

La programmation générique a pour objectif principal de résoudre des problèmes pour une grande variété de types en utilisant un minimum de code générique. Shapeless fournit deux ensembles d'outils dans ce but :

1. Un ensemble de types de donnée générique qui peuvent être inspectés, itérés et manipulés au `type-level`
2. Le mapping automatique entre *algebraic data types (*ADT*s)* (encodé en Scala par les *case classes* et les *sealed traits*) et leurs représentations génériques.

Dans ce chapitre, nous commençons par un récapitulatif sur la théorie des types alébriques et la raison pour laquelle ils peuvent être familiers pour le développeur Scala. Puis, nous verrons les représentations génériques utilisées par shapeless et nous traiterons de la façon dont ils sont reliés aux *ADT*s concrets. Enfin, nous présenterons une type class appelée `Generic` qui fournit un mapping automatique bidirectionnel entre un *ADT* et sa représentation générique. Enfin, nous utiliserons `Generic` dans quelques exemples pour convertir des valeurs d'un type vers un autre.

2.1 Rappel: types de données algébriques

Algebraic data types (ADT*s*)[1] est un nom raffiné qui cache un concept très simple. C'est un moyen idiomatique de représenter une donnée en utilisant les « et » et « ou » logiques. Par exemple :

- une forme est un rectangle **ou** un cercle
- un rectangle a une largeur **et** une hauteur
- un cercle a un rayon

Dans la terminologie des *ADT*s, les types « et » comme le rectangle et le cercle sont appelés *products (produits)*, les types « ou » comme le type forme sont appelés *coproducts (coproduits)*. En Scala, nous représentons typiquement les produits par des case class et les coproduits par des sealed traits :

```
sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends
    Shape
final case class Circle(radius: Double) extends Shape

val rect: Shape = Rectangle(3.0, 4.0)
val circ: Shape = Circle(1.0)
```

Ce qui est beau avec les *ADT*s, c'est qu'ils sont complètement type safe. Le compilateur a une connaisence complète de l'algèbre[2] que nous définissons, il peut donc nous aider à écrire des méthodes correctement typées avec nos types.

```
def area(shape: Shape): Double =
  shape match {
    case Rectangle(w, h) => w * h
    case Circle(r)       => math.Pi * r * r
```

[1] Ne doivent pas être confondus avec les « abstract data types », qui sont un outil différent en informatique et qui ont peu d'influence sur notre sujet.

[2] Définition du mot « algèbre » : les symboles que nous définissons, comme les rectangles et les cercles ainsi que les règles de manipulation de ces symboles, celles-ci formulées sous forme de méthodes.

```
  }
```

```
area(rect)
// res1: Double = 12.0

area(circ)
// res2: Double = 3.141592653589793
```

2.1.1 Les écritures alternatives

Les sealed traits et les *case classes* sont sans aucun doute les encodages les plus pratique des *ADT*s en Scala. Mais ce ne sont pas les *seules* possibilités d'encodage. Par exemple, la bibliotèque standard de Scala fournit des produits génériques appelés `Tuples` ainsi qu'un coproduit générique : `Either`. Nous aurions pu choisir d'écrire notre Shape comme suit :

```
type Rectangle2 = (Double, Double)
type Circle2    = Double
type Shape2     = Either[Rectangle2, Circle2]

val rect2: Shape2 = Left((3.0, 4.0))
val circ2: Shape2 = Right(1.0)
```

Bien que cette écriture soit moins lisible que la précédente avec des case class, elle dispose de certaines des propriétés requises. Nous pouvons toujours écrire du code type safe avec Shape2 :

```
def area2(shape: Shape2): Double =
  shape match {
    case Left((w, h)) => w * h
    case Right(r)     => math.Pi * r * r
  }
```

```
area2(rect2)
// res4: Double = 12.0

area2(circ2)
// res5: Double = 3.141592653589793
```

Il est important de noter que `Shape2` est une écriture plus *générique* que `Shape`[3]. Tout code qui fonctionne avec une paire de `Doubles` fonctionnera avec `Rectangle2` et vice versa. En tant que développeur Scala nous avons tendance à préférer les types sémantiques comme `Rectangle` et `Circle` aux types génériques tels que `Rectangle2` et `Circle2`, précisément à cause de leur nature spécialisée. Toutefois, dans certain cas, la généralité est préférable. Par exemple, si l'on sérialise des données sur un disque, nous ne nous soucions pas des différences entre une paire de `Doubles` et un `Rectangle2`. On écrit seulement les nombres et rien d'autre.

Shapeless nous donne le meilleur des deux mondes ; nous pouvons utiliser les types sémantiques par défaut et passer aux représentations génériques lorsque le besoin d'interopérabilité se fait sentir (nous y reviendrons).

En revanche, au lieu d'utiliser `Tuples` et `Either`, shapeless utilise son propre type de données pour représenter les produits et coproduits génériques. Nous présenterons ces types dans la section suivante.

2.2 Écriture générique pour les produits

Dans la section précédente, nous avons présenté les *tuples* comme une représentation générique des produits. Malheureusement, les *tuples* présentent deux inconvénients qui les rendent inappropriés aux besoins de shapeless.

1. Chaque taille de tuple est dotée d'un type différent, ces types sont indépendants les uns des autres, ce qui rend difficile l'écriture de code qui fait abstraction de la taille des produits.

[3] nous utilisons « générique » de façon informelle, au lieu du sens conventionnel : « un type paramétré ».

2. Il n'existe pas de *tuples* de taille zero, qui sont pourtant importants pour représenter les produits sans champ. On pourrait utiliser `Unit`, mais on souhaite que chaque représentation générique ait un supertype commun et tangible. Le plus petit ancêtre commun à `Unit` et `Tuple2` est `Any`, une combinaison des deux est donc impensable.

C'est pourquoi shapeless utilise un encodage générique différent pour les produits : *heterogeneous lists* ou `HLists`[4].

La `HList` est soit une liste vide `HNil`, soit une paire `::[H, T]` où `H` est un type arbitraire et `T` une autre `HList`. Parce que chaque `::` a son propre `H` et `T`, le type de chaque élément est encodé séparément dans le type de la liste global:

```
import shapeless.{HList, ::, HNil}

val product: String :: Int :: Boolean :: HNil =
  "Sunday" :: 1 :: false :: HNil
```

Le type et la valeur de la `HList` ci-dessus se reflètent. Les deux représentent les 3 membres : une `String`, un `Int`, et un `Boolean`. On peut y retrouver la `tête` et la queue, et le type des éléments y est préservé :

```
val first = product.head
// first: String = Sunday

val second = product.tail.head
// second: Int = 1

val rest = product.tail.tail
// rest: shapeless.::[Boolean,shapeless.HNil] = false :: HNil
```

Le compilateur connaît la taille exacte de chaque `HList`, cela devient donc une erreur de compilation de demander la `tête` ou la queue d'une liste vide :

[4] `Product` est probablement un meilleur nom pour `HList`, mais la bibliothèque standard dispose malheureusement déjà d'un `scala.Product`.

```
product.tail.tail.tail.head
// <console>:15: error: could not find implicit value for parameter c:
      shapeless.ops.hlist.IsHCons[shapeless.HNil]
//        product.tail.tail.tail.head
//                                 ^
```

En plus de pouvoir inspecter et itérer sur les `HLists`, on peut les manipuler et les transformer. Par exemple, on peut ajouter un élément avec la méthode `::`. Encore une fois, notez comme le type du résultat reflète le nombre des éléments de la liste.

```
val newProduct: Long :: String :: Int :: Boolean :: HNil =
  42L :: product
```

Shapeless fournit aussi des outils pour effectuer des opérations plus complexes comme un mapping, un filtrage ou une concaténation de listes. On abordera cela plus en détails dans la Partie II.

Les propriétés que l'on obtient avec `HLists` ne sont pas magiques. On aurait pu obtenir ces fonctionalités en utilisant `(A, B)` et `Unit` comme alternative à `::` et `HNil`. Néanmoins, il existe un avantage à garder nos types génériques séparés de nos types sémantiques dans les applications. `HList` fournit cette séparation.

2.2.1 Changer de représentation en utilisant *Generic*

Shapeless fournit une type class appelée `Generic` qui permet de convertir des *ADT* concrets en représentation générique et vice versa. Il y a quelques macros en coulisses qui permettent d'invoquer des instances de `Generic` sans boilerplate :

```
import shapeless.Generic

case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

```
val iceCreamGen = Generic[IceCream]
// iceCreamGen: shapeless.Generic[IceCream]{type Repr = shapeless.::[
     String,shapeless.::[Int,shapeless.::[Boolean,shapeless.HNil]]]} =
      anon$macro$4$1@6e2fd2dc
```

Notez que les instance de `Generic` ont un membre de type `Repr` qui contient le type de sa représentation générique. Dans notre cas `iceCreamGen.Repr` est `String :: Int :: Boolean :: HNil`. Les instances de `Generic` disposent de deux méthodes : Une pour convertir vers le type `Repr` appelé `to` et la méthode inverse `from` :

```
val iceCream = IceCream("Sundae", 1, false)
// iceCream: IceCream = IceCream(Sundae,1,false)

val repr = iceCreamGen.to(iceCream)
// repr: iceCreamGen.Repr = Sundae :: 1 :: false :: HNil

val iceCream2 = iceCreamGen.from(repr)
// iceCream2: IceCream = IceCream(Sundae,1,false)
```

On peut convertir de l'un vers l'autre deux *ADT*s qui ont le même `Repr` en utilisant leurs `Generics` :

```
case class Employee(name: String, number: Int, manager: Boolean)
```

```
// Create an employee from an ice cream:
val employee = Generic[Employee].from(Generic[IceCream].to(iceCream))
// employee: Employee = Employee(Sundae,1,false)
```

> *Les autres types de produits*
>
> Il vaut la peine de souligner que les *tuples Scala* sont des *case classes*, donc `Generic` fonctionne très bien avec :

```
val tupleGen = Generic[(String, Int, Boolean)]

tupleGen.to(("Hello", 123, true))
// res4: tupleGen.Repr = Hello :: 123 :: true :: HNil

tupleGen.from("Hello" :: 123 :: true :: HNil)
// res5: (String, Int, Boolean) = (Hello,123,true)
```

Cela marche aussi avec les *case classes* de plus de 22 champs:

```
case class BigData(
  a:Int,b:Int,c:Int,d:Int,e:Int,f:Int,g:Int,h:Int,i:Int,j:Int,
  k:Int,l:Int,m:Int,n:Int,o:Int,p:Int,q:Int,r:Int,s:Int,t:Int,
  u:Int,v:Int,w:Int)

Generic[BigData].from(Generic[BigData].to(BigData(
  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23)))
// res6: BigData = BigData
    (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23)
```

2.3 Coproduit générique

Maintenant que nous savons comment Shapeless encode les types produit, qu'en est-il des coproduits ? Précédemment, nous avons abordé le `Either`, mais il souffre des mêmes inconvénients que les *tuples*. Encore une fois, shapeless fournit son propre encodage, similaire au `HList` :

```
import shapeless.{Coproduct, :+:, CNil, Inl, Inr}

case class Red()
case class Amber()
case class Green()

type Light = Red :+: Amber :+: Green :+: CNil
```

En général, les coproduits prennent la forme de `A :+: B :+: C :+: CNil` signifiant « A ou B ou C, » où `:+:` peut être librement interprété comme `Ei-`

ther. Globalement, le type d'un coproduit encode tous les types possibles d'une disjonction, mais ces instances contiennent uniquement la valeur de l'une des possibilités. :+: dispose de deux sous types, Inl et Inr, qui correspondent vagument à Left et Right. On crée des instances de coproduit en imbriquant des constructeurs de Inl et de Inr :

```
val red: Light = Inl(Red())
// red: Light = Inl(Red())

val green: Light = Inr(Inr(Inl(Green())))
// green: Light = Inr(Inr(Inl(Green())))
```

Chaque type de coproduit se termine par CNil, qui est un type inhabité (sans valeur), similaire à Nothing. On ne peut donc pas instancier CNil ou construire un Coproduct uniquement à partir d'instance de Inr. Il y a toujours exactement un Inl dans chaque valeur de coproduit.

Encore une fois, il convient de signaler que Coproducts n'a rien de spécial. La fonctionnalité du dessus peut être obtenue en utilisant Either et Nothing à la place de :+: et CNil. Utiliser Nothing induit des difficultés techniques, mais on aurait pu utiliser n'importe quel autre type inhabité ou type singleton à la place de CNil.

2.3.1 Échanger les encodages à l'aide de *Generic*

À première vue, les Coproduct sont difficiles à parser. Cependant, nous voyons comment ils s'intègrent dans le contexte plus large des écritures génériques. En plus de comprendre les *case classes* et les *case objects*, Generic de shapeless comprend également les *sealed traits* et les classes abstraites :

```
import shapeless.Generic

sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends
    Shape
final case class Circle(radius: Double) extends Shape
```

```
val gen = Generic[Shape]
// gen: shapeless.Generic[Shape]{type Repr = shapeless.:+:[Rectangle,
     shapeless.:+:[Circle,shapeless.CNil]]} = anon$macro$1$1@1d254277
```

Le `Repr` du `Generic` de `Shape` est un coproduit des sous-types de `Shape`: `Rectangle :+: Circle :+: CNil`. On peut utiliser les méthodes `to` et `from` de l'instance de `Generic` pour convertir `Shape` en `gen.Repr` et vice versa :

```
gen.to(Rectangle(3.0, 4.0))
// res3: gen.Repr = Inl(Rectangle(3.0,4.0))

gen.to(Circle(1.0))
// res4: gen.Repr = Inr(Inl(Circle(1.0)))
```

2.4 Résumé

Dans ce chapitre, nous avons abordé les représentations génériques que Shapeless fournit pour les types de données algébriques de Scala. `HLists` pour les types produits et `Coproducts` pour les types coproduits. Nous avons aussi présenté la *type class* `Generic` qui fournit mapping bidirectionnel entre un *ADT* et sa représentation générique. Nous n'avons pas encore abordé ce qui rend les écritures génériques si attractives. Le cas d'utilisation que nous avons couvert (conversion entre *ADT*s) est amusant mais pas particulièrement utile.

La vraie puissance de `HLists` et de `Coproducts` provient de leur structure récursive. On peut écrire du code qui traverse leur représentation et calcule des valeurs à partir des éléments qui les constituent. Dans le prochain chapitre nous verrons notre premier véritable cas d'utilisation : la déduction automatique d'instance de *type class*es.

Chapter 3

Déduire automatiquement les instances de *type class*

Dans le dernier chapitre, nous avons vu comment le type `Generic` nous permet de convertir toute instance d'un *ADT* en un encodage générique composé de `HLists` et `Coproducts`. Dans ce chapitre, nous verrons notre premier véritable cas d'utilisation : la déduction automatique d'instances de *type class*.

3.1 Bref rappel : les type classes

Avant d'entrer dans les détails de la déduction d'instance, faisons un bref rappel des aspects importants des *types classes*.

Les type classes sont un pattern de programmation emprunté à *Haskell* (le mot « classe » n'a rien a voir avec les classes de la programmation orientée object). Nous les écrivons en Scala avec des traits et des implicites. Une *type class* est un trait paramétré représentant une fonctionalité générale que l'on voudrait appliquer à une grande variété de types :

```
// Turn a value of type A into a row of cells in a CSV file:
trait CsvEncoder[A] {
```

```
  def encode(value: A): List[String]
}
```

On implémente notre type class avec une *instance* pour chaque type visé. Si nous voulons avoir l'instance dans le scope automatiquement, on peut les placer dans l'objet compagnon de la *type class*. Sinon, on peut les placer dans une objet séparé qui fera office de bibliothèque et que l'utilisateur pourra importer manuellement :

```
// Custom data type:
case class Employee(name: String, number: Int, manager: Boolean)

// CsvEncoder instance for the custom data type:
implicit val employeeEncoder: CsvEncoder[Employee] =
  new CsvEncoder[Employee] {
    def encode(e: Employee): List[String] =
      List(
        e.name,
        e.number.toString,
        if(e.manager) "yes" else "no"
      )
  }
```

On marque chaque instance avec le mot clé `implicit`, et on définit une ou plusieurs méthodes qui acceptent le paramètre implicite du type de notre type class, elles feront office de point d'entrée :

```
def writeCsv[A](values: List[A])(implicit enc: CsvEncoder[A]): String
    =
  values.map(value => enc.encode(value).mkString(",")).mkString("\n")
```

Testons `writeCsv` avec quelques données de test :

```
val employees: List[Employee] = List(
  Employee("Bill", 1, true),
  Employee("Peter", 2, false),
  Employee("Milton", 3, false)
)
```

Quand on appelle `writeCsv`, le compilateur calcule la valeur du paramètre de type et cherche un implicite de `CsvEncoder` du type correspondant :

```
writeCsv(employees)
// res4: String =
// Bill,1,yes
// Peter,2,no
// Milton,3,no
```

On peut utiliser `writeCsv` avec nimporte quel type de donnée, du moment que l'on a dans le scope l'implicite de `CsvEncoder` correspondant :

```
case class IceCream(name: String, numCherries: Int, inCone: Boolean)

implicit val iceCreamEncoder: CsvEncoder[IceCream] =
  new CsvEncoder[IceCream] {
    def encode(i: IceCream): List[String] =
      List(
        i.name,
        i.numCherries.toString,
        if(i.inCone) "yes" else "no"
      )
  }

val iceCreams: List[IceCream] = List(
  IceCream("Sundae", 1, false),
  IceCream("Cornetto", 0, true),
  IceCream("Banana Split", 0, false)
)
```

```
writeCsv(iceCreams)
// res7: String =
// Sundae,1,no
// Cornetto,0,yes
// Banana Split,0,no
```

3.1.1 Résoudre les instances

Les *types classes* sont très flexibles mais elles nous imposent de définir une instance pour chaque type qui nous intéresse. Heureusement, le compilateur de Scala a plus d'un tour dans son sac, si on lui donne certaines regles, il est capable de résoudre les instances pour nous.
Par exemple, on peut ecrire une règle qui créé un `CsvEncoder` pour `(A, B)` pour un `CsvEncoder` pour `A` et un pour `B` donnés :

```
implicit def pairEncoder[A, B](
  implicit
  aEncoder: CsvEncoder[A],
  bEncoder: CsvEncoder[B]
): CsvEncoder[(A, B)] =
  new CsvEncoder[(A, B)] {
    def encode(pair: (A, B)): List[String] = {
      val (a, b) = pair
      aEncoder.encode(a) ++ bEncoder.encode(b)
    }
  }
```

Quand tous les paramètres d'un `implicit def` sont eux meme marquer `implicit`, alors le compilateur peut l'utiliser comme une règle de résolution pour crée des instance a partire d'autres instances. Par exemple, si l'on appel `writeCsv` et qu'on lui passe une `List[(Employee, IceCream)]`, le compilateur est capable de combiner `pairEncoder`, `employeeEncoder` et `iceCreamEncoder` pour produire le `CsvEncoder[(Employee, IceCream)]` requis:

```
writeCsv(employees zip iceCreams)
// res8: String =
// Bill,1,yes,Sundae,1,no
// Peter,2,no,Cornetto,0,yes
// Milton,3,no,Banana Split,0,no
```

À partir d'une liste de règles écrites à partir d' `implicit vals` et d'`implicit defs`, le compilateur est capable de *rechercher* les combinaisons pour retourner l'instance requise.

Cette fonctionalité, connue sous le nom de *résolution d'`implicits`*, est ce qui rend le pattern des *types classes* si puissant en Scala.

Même avec cette puissance, le compilateur ne peut démanteler nos *case classes* et *sealed traits*. On est tenu de définir à la main les instances de nos *ADT*s. Les représentations génériques de shapeless changent la donne, car ils nous permettent de déduire automatiquement les instances de nimporte quel *ADT*.

3.1.2 Les définitions de *type class* idiomatique

Le style généralement accépter pour la définition d'une type classe idiomatique Il est généralement accépte d'inclure un objet compagon contenant certaines methode standar lors de la définition d'une type classe idiomatique :

```
object CsvEncoder {
  // "Summoner" method
  def apply[A](implicit enc: CsvEncoder[A]): CsvEncoder[A] =
    enc

  // "Constructor" method
  def instance[A](func: A => List[String]): CsvEncoder[A] =
    new CsvEncoder[A] {
      def encode(value: A): List[String] =
        func(value)
    }

  // Globally visible type class instances
}
```

La methode `apply` connue sous le nom de *summoner* ou *materializer*, nous permet d'invoquer une instance de *type class* selon un type donnée:

```
CsvEncoder[IceCream]
// res9: CsvEncoder[IceCream] = $anon$1@7f751313
```

Dans les cas les plus simple le *summoner* fait la meme chose que la méhode `implicitly` définie dans `scala.Predef`:

```
implicitly[CsvEncoder[IceCream]]
// res10: CsvEncoder[IceCream] = $anon$1@7f751313
```

Cependant, comme nous le verrons dans la section 4.2, lors que l'on travaille avec shapeless il arrive que la methode `implicitly` n'infère pas les types correctement. On peut toujours définir une methode *summoner* pour avoir le bon comportement, donc cela vaut le coup d'en écrire une pour chaque *type class* que l'on créé. On peut aussi utiliser une methode de shapeless appelée "`the`" (nous y reviendrons):

```
import shapeless._
```

```
the[CsvEncoder[IceCream]]
// res11: CsvEncoder[IceCream] = $anon$1@7f751313
```

La methode `instance` parfois appelé `pure`, fournis une syntaxe simple pour crée de nouvelle instance de *type classe*, réduisant au passage le boilerplate pour les classes anoymes:

```
implicit val booleanEncoder: CsvEncoder[Boolean] =
  new CsvEncoder[Boolean] {
    def encode(b: Boolean): List[String] =
      if(b) List("yes") else List("no")
  }
```

Est réduit en:

```
implicit val booleanEncoder: CsvEncoder[Boolean] =
  instance(b => if(b) List("yes") else List("no"))
```

Malheureusement, les limitations imposées par le livre nous empèchent d'écrire un grand singleton contenant beaucoup de méthodes et d'instances. Nous préférons donc décrire les définitons en dehors de leurs *object* compagnon. Ceci est à garder a l'esprit lorsque que vous lisez ce livre mais rappelez-vous que code complet se trouve dans le dépôt mentionné dans la section 1.2

3.2 Déduire des instances pour les produits

Dans cette section nous utilisons shapeless pour déduire des instances de *types classes* pour des types de produits. (*ie. case classes*).

Utilisons les deux idées suivantes :

1. Si l'on a une instance de type *class* pour la tête et la queue d'une `HList`, on peut en déduire l'instance de toute la `HList`.
2. Si nous avons une case class `A`, un `Generic[A]` et une instance de *type class* pour le `Repr` de ce générique, nous pouvons alors les combiner pour obtenir une instance de `A`.

Prenons `CsvEncoder` et `IceCream` comme exemples :

- `IceCream` a un `Repr` générique de type `String :: Int :: Boolean :: HNil`.
- Le `Repr` est fait d'une `String`, d'un `Int`, d'un `Boolean` et d'une `HNil`. Si nous avons un `CsvEncoder` pour ces types alors nous disposons d'un encodeur pour le tout.
- Si nous pouvons déduire un `CsvEncoder` pour le `Repr`, nous pouvons en créer un pour `IceCream`.

3.2.1 Les instances de *HLists*

Commençons par définir les constructeurs d'instance de `CsvEncoders` pour `String`, `Int` et `Boolean`:

```
def createEncoder[A](func: A => List[String]): CsvEncoder[A] =
  new CsvEncoder[A] {
    def encode(value: A): List[String] = func(value)
  }

implicit val stringEncoder: CsvEncoder[String] =
  createEncoder(str => List(str))
```

```
implicit val intEncoder: CsvEncoder[Int] =
  createEncoder(num => List(num.toString))

implicit val booleanEncoder: CsvEncoder[Boolean] =
  createEncoder(bool => List(if(bool) "yes" else "no"))
```

Nous pouvons combiner ces blocs de construction pour créer un encodeur pour notre `HList`. Nous utiliserons deux règles : une pour `HNil` et une pour `::` comme illustré ci-dessous :

```
import shapeless.{HList, ::, HNil}

implicit val hnilEncoder: CsvEncoder[HNil] =
  createEncoder(hnil => Nil)

implicit def hlistEncoder[H, T <: HList](
  implicit
  hEncoder: CsvEncoder[H],
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :: T] =
  createEncoder {
    case h :: t =>
      hEncoder.encode(h) ++ tEncoder.encode(t)
  }
```

Prises toutes ensemble, ces cinq instances nous permettent d'invoquer un `CsvEncoders` pour toute `HList` impliquant `Strings`, `Ints` et `Booleans`:

```
val reprEncoder: CsvEncoder[String :: Int :: Boolean :: HNil] =
  implicitly
```

```
reprEncoder.encode("abc" :: 123 :: true :: HNil)
// res9: List[String] = List(abc, 123, yes)
```

3.2.2 Des instance pour nos produits

On peut combiner nos règles de déduction de `HLists` avec nos instances de `Generic` pour produire un `CsvEncoder` pour `IceCream`:

```
import shapeless.Generic

implicit val iceCreamEncoder: CsvEncoder[IceCream] = {
  val gen = Generic[IceCream]
  val enc = CsvEncoder[gen.Repr]
  createEncoder(iceCream => enc.encode(gen.to(iceCream)))
}
```

et l'utiliser comme suit :

```
writeCsv(iceCreams)
// res11: String =
// Sundae,1,no
// Cornetto,0,yes
// Banana Split,0,no
```

Mais cette solution est spécifique à `IceCream`. Idéalement on voudrait définir une seule règle pour toutes les *case classes* qui ont un `Generic` et le `CsvEncoder` correspondant. Montrons pas à pas comment faire cette déduction. Voici la première étape :

```
implicit def genericEncoder[A](
  implicit
  gen: Generic[A],
  enc: CsvEncoder[???]
): CsvEncoder[A] = createEncoder(a => enc.encode(gen.to(a)))
```

Nous devons choisir le type à placer à la place de `???`. Mais le problème est que l'on ne peut pas utiliser le type `Repr` associé avec `gen`, la solution suivante n'est pas possible :

```
implicit def genericEncoder[A](
  implicit
  gen: Generic[A],
  enc: CsvEncoder[gen.Repr]
): CsvEncoder[A] =
  createEncoder(a => enc.encode(gen.to(a)))
// <console>:27: error: illegal dependent method type: parameter may
     only be referenced in a subsequent parameter section
```

```
//          gen: Generic[A],
//          ^
```

Nous avons ici un problème de scope : On ne peut faire référence à un membre de type d'un paramètre à partir d'un autre paramètre du même bloc. L'astuce pour contourner ce problème est d'ajouter un nouveau paramètre de type à notre méthode, et y faire référence dans chacune des valeurs associées :

```
implicit def genericEncoder[A, R](
  implicit
  gen: Generic[A] { type Repr = R },
  enc: CsvEncoder[R]
): CsvEncoder[A] =
  createEncoder(a => enc.encode(gen.to(a)))
```

Nous traiterons ce style d'écriture dans le chapitre suivant. Maintenant, cette définiton compile et fonctionne comme attendu avec n'importe quelle case class. Intuitivement, la définition nous dit :

> *Avec un A donné et une `HList` de type R, avec un implicite `Generic` qui relie A a R et un `CsvEncoder` pour R, alors on crée un `CsvEncoder` pour A.*

Nous avons maintenant un système complet qui peut gérer n'importe quelle case class.

L'appel suivant

```
writeCsv(iceCreams)
```

est résolu par le compilateur comme suit :

```
writeCsv(iceCreams)(
  genericEncoder(
    Generic[IceCream],
    hlistEncoder(stringEncoder,
```

```
    hlistEncoder(intEncoder,
      hlistEncoder(booleanEncoder, hnilEncoder)))))
```

et il peut inférer les valeurs correctes pour un grand nombre de types différents. Tout comme moi, je suis sur que vous appréciez ne pas avoir à écrire ce code à la main !

l'alias de type Aux

Les types refinements comme `Generic[A] { type Repr = L }` sont verbeux et difficiles à lire, c'est la raison pour laquelle shapeless fournit un alias de type, `Generic.Aux`, pour reformuler le membre de type en un paramètre de type :

```
package shapeless

object Generic {
  type Aux[A, R] = Generic[A] { type Repr = R }
}
```

Avec l'alias de type la définition devient beaucoup plus lisible :

```
implicit def genericEncoder[A, R](
  implicit
  gen: Generic.Aux[A, R],
  env: CsvEncoder[R]
): CsvEncoder[A] =
  createEncoder(a => env.encode(gen.to(a)))
```

Notez bien que le type Aux ne change pas la sémantique, cela rend juste les chose plus faciles à lire. Le pattern Aux est souvent utilisé dans le code de shapeless.

3.2.3 Alors quels en sont les inconvénients ?

Si tout ce que l'on vient de voir semble magique, permettez-moi de vous ramener à la réalité. Si quelque-chose tourne mal, le compilateur ne vous sera pas d'une grande aide.

Il existe deux raisons pour laquelle le code précédent pourrait ne pas compiler. La première est si le compilateur ne peut pas trouver l'instance de `Generic`. Par exemple, si nous essayons d'appeler `writeCsv` avec une simple classe :

```
class Foo(bar: String, baz: Int)
```

```
writeCsv(List(new Foo("abc", 123)))
// <console>:31: error: could not find implicit value for parameter
     encoder: CsvEncoder[Foo]
//        writeCsv(List(new Foo("abc", 123)))
//                 ^
```

Dans ce cas le message est relativement simple à comprendre. Si shapeless ne peut calculer un `Generic` cela veut dire que le type en question n'est pas un *ADT* (il y a quelque-part dans l'algèbre un type qui n'est pas une case class ou un trait scellé).

L'autre source potentielle d'erreur survient lorsque le compilateur ne peut calculer un `CsvEncoder` pour notre `HList`. Cela arrive normalement car l'on n'a pas d'encodeur pour un des champs de notre *ADT*. Par exemple nous n'avons pas encore défini de `CsvEncoder` pour `java.util.Date`, donc le code suivant ne fonctionne pas :

```
import java.util.Date

case class Booking(room: String, date: Date)
```

```
writeCsv(List(Booking("Lecture hall", new Date())))
// <console>:33: error: could not find implicit value for parameter
     encoder: CsvEncoder[Booking]
//        writeCsv(List(Booking("Lecture hall", new Date())))
//                 ^
```

Le messsage d'erreur ne nous aide pas vraiment. Tout ce que le compilateur sait, c'est qu'il a essayé un grand nombre de combinaisons d'implicites et qu'aucune ne fonctionnait. Il n'a aucune idée de quelle combinaison était la

plus proche de celle attendue, donc il ne peut nous dire où se trouve la source du problème.

Il n'y a pas de quoi se réjouir ici. Nous devons trouver nous-même la source des erreurs par un processus d'élimination. Nous aborderons les techniques de debuggage dans la section 3.5. Pour l'instant la seule fonctionalité qui compense c'est que la résolution d'implicite plantera toujours à la compilation. Il y a une petite chance que cela finisse par produire du code qui plante durant l'exécution.

3.3 Déduire les instances pour les coproduits

Dans la section précédente nous avons crée un ensemble de règle pour déduire automatiquement un `CsvEncoder` pour n'importe quel type de produits. Dans cette section allons appliqué les memes patterns aux coproduits. Retournons a notre exemple, l'*ADT* shape :

```
sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends
    Shape
final case class Circle(radius: Double) extends Shape
```

La repésentation générique de `Shape` est `Rectangle :+: Circle :+: CNil`. Dans la section 3.2.2 nous avons définie des encodeurs de produit pour `Rectangle` et `Circle`. Maintenant, pour écrire des `CsvEncoders` générique pour `:+:` et `CNil`, nous alons utilisé les mêmes principes que pour `HLists`:

```
import shapeless.{Coproduct, :+:, CNil, Inl, Inr}

implicit val cnilEncoder: CsvEncoder[CNil] =
  createEncoder(cnil => throw new Exception("Inconceivable!"))

implicit def coproductEncoder[H, T <: Coproduct](
  implicit
  hEncoder: CsvEncoder[H],
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :+: T] = createEncoder {
```

```
  case Inl(h) => hEncoder.encode(h)
  case Inr(t) => tEncoder.encode(t)
}
```

Il importe de noter deux choses :

1. Comme `Coproducts` est une *disjonction* de types, l'encoder de `:+:` doit *choisir* si il a à encoder la valeur de droit ou de gauche. On fait un pattern matching sur les deux sous types de `:+:`, qui sont `Inl` pour la gauche et `Inr` pour la droite.

2. Etonnament, l'encoder de `CNil` lève une exception! Mais pas de panique. Rappelez vous que l'on ne peut crée de valeur pour le type `CNil`, donc the `throw` et en fait du code mort.

Si l'on utilise ces définitions avec celle de nos produit de la section 3.2, on sera capable de sérializer une liste de shapes. Essayons:

```
val shapes: List[Shape] = List(
  Rectangle(3.0, 4.0),
  Circle(1.0)
)
```

```
writeCsv(shapes)
// <console>:33: error: could not find implicit value for parameter
     encoder: CsvEncoder[Shape]
//        writeCsv(shapes)
//                 ^
```

Oh non, ca n'a pas fonctionné ! Le message d'erreur ne nous aide pas comme prévu. Nous avons cette erreur car nous n'avons pas d'instance de `CsvEncoder` pour `Double` :

```
implicit val doubleEncoder: CsvEncoder[Double] =
  createEncoder(d => List(d.toString))
```

Avec cette nouvelle définition, tout fonctionne comme prévus:

```
writeCsv(shapes)
// res7: String =
// 3.0,4.0
// 1.0
```

SI-7046 et vous Il y a dans Scala un bug du compilateur appelé SI-7046[a] qui peut amener la résolution de générique pour comproduct a ne pas fonctionner. Le bug provoque dans certaines partie de l'API de macro, dont shapeless dépend, deviens sensible a l'ordre des définitions dans le code source. C'est un problème qui peut souvant etre contourner en réordonant le code et en renoment les fichiers, mais ces solution on tendence a ne durée qu'un temps et sont peut fiable.

Si vous utilisez *Lightbend Scala 2.11.8* ou une version plus ancienne et que vous ètes toucher par ce problème, pensez a mettre a jour vers la version *Lightbend Scala 2.11.9* ou *Typelevel Scala 2.11.8*. SI-7046 est corrigée dans chacune de de ces versions.

[a] https://issues.scala-lang.org/browse/SI-7046

3.3.1 Aligner les colonnes dans la sortie CSV

Notre encodeur de CSV n'est idéal dans ca forme courante. Il permet au champs de `Rectangle` et `Circle` de se retrouver dans la meme colonne. Pour remédier a a ce problème il faut changer la définition de `CsvEncoder` pour ajouter la largeur des type de données et ainsi espacer les colonnes en conséquence. Le repo d'exemple contenant l'implémentation complète d'un `CsvEncoder` traitant ce problème est linké dans la section 1.2

3.4 La déduction d'instance pour les type récursif.

Essayons quelque chose de plus ambitieu : un arbre binaire :

```
sealed trait Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](value: A) extends Tree[A]
```

En théorie nous devrions deja avoir toutes les définiton pour instancier un CSV writer pour notre arbre. Pourtant, appler `writeCsv` provoque une erreur : However, calls to `writeCsv` fail to compile:

```
CsvEncoder[Tree[Int]]
// <console>:25: error: could not find implicit value for parameter
     enc: CsvEncoder[Tree[Int]]
//        CsvEncoder[Tree[Int]]
//                  ^
```

Le problème est que notre type est qu'il est récusif. Le compilateur rentre dans une boucle infie en essayant d'appliquer nos implicits puis il abandonne.

3.4.1 Les divergences d'implicits

La résolution d'`implicit` est un processus de recherche. Le compilateur utilise des heuristiques pour s'orienter vers la bonne solution. Le compilateur effectue ses recherches branche par branche, si l'une d'elle ne donne pas de résultat favorable, le compilateur considère que cette branche n'aboutira pas et continue ses recherches sur une autre branche.

Une des heuristique est conçue spécifiquement pour éviter les boucles infinies. Si le compilateur rencontre le type cible deux fois dans une branche de recherche; il abandonne cette branche et passe a une autre. On peut l'observer si on regarde l'expression `CsvEncoder[Tree[Int]]`. Le processus de résolution d'implicits suit les étapes suivantes :

```
CsvEncoder[Tree[Int]]                          // 1
CsvEncoder[Branch[Int] :+: Leaf[Int] :+: CNil] // 2
CsvEncoder[Branch[Int]]                        // 3
CsvEncoder[Tree[Int] :: Tree[Int] :: HNil]     // 4
CsvEncoder[Tree[Int]]                          // 5 uh oh
```

On rencontre `Tree[A]` deux fois lignes 1 et 5, donc le compilateur abandonne la recherche dans cette branche. La conséquence est que le compilateur echoue à trouver le bon *implicit*.

En fait, la situation est pire. Si le compilateur voit le même constructeur de types deux fois et que la complexité du paramètre de type *augmente* il suppose que la branche n'aboutira pas.* C'est un problème avec shapeless car les types comme `::[H, T]` et `:+:[H, T]` peuvent apparaître plusieurs fois lorsque le compilateur développe les représentations génériques. Ce qui pousse le compilateur a abandonner prématurément même si il aurait pu trouver la solution si il avait persisté à chercher dans cette voie.

Compte-tenu des types suivants :

```
case class Bar(baz: Int, qux: String)
case class Foo(bar: Bar)
```

Le déroulement de la recherche pour `Foo` ressemble à ça :

```
CsvEncoder[Foo]                   // 1
CsvEncoder[Bar :: HNil]           // 2
CsvEncoder[Bar]                   // 3
CsvEncoder[Int :: String :: HNil] // 4 uh oh
```

Le compilateur essait de résoudre `CsvEncoder[::[H, T]]` deux fois dans cette branche, une fois a la ligne 2 et une fois a la ligne 4. Le paramètre de type T est plus complexe à la ligne 4 qu'à la ligne 2, donc le compilateur pense (de façon erronée dans ce cas) que la recherche dans cette branche n'aboutira pas. Il change de branche encore et encore, il ne trouvera pas de quoi générer la bonne instance.

3.4.2 *Lazy*

La divergence d'`implicit` peut marquer un arrêt pour la bibliotèque shapeless. Heureusement, shapeless fournit un type appeler `Lazy` pour contourner ce problème. `Lazy` fait deux choses :

1. Il supprime la divergence d'`implicit` à la compilation en se protégeant des heuristiques trop défensives.
2. Il reporte l'évaluation des paramètres implicites au runtime, ce qui permet la déduction des implicits récursifs.

On utilise `Lazy` en le paramétrant avec nos paramètres implicites. En règle générale, il est toujours bon d'englober dans `Lazy` le paramètre de "tête" de toute règle de `HList` ou de `Coproduct` ainsi que n'importe quel paramètre `Repr` d'un `Generic`:

```
implicit def hlistEncoder[H, T <: HList](
  implicit
  hEncoder: Lazy[CsvEncoder[H]], // wrap in Lazy
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :: T] = createEncoder {
  case h :: t =>
    hEncoder.value.encode(h) ++ tEncoder.encode(t)
}
```

```
implicit def coproductEncoder[H, T <: Coproduct](
  implicit
  hEncoder: Lazy[CsvEncoder[H]], // wrap in Lazy
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :+: T] = createEncoder {
  case Inl(h) => hEncoder.value.encode(h)
  case Inr(t) => tEncoder.encode(t)
}
```

```
implicit def genericEncoder[A, R](
  implicit
  gen: Generic.Aux[A, R],
  rEncoder: Lazy[CsvEncoder[R]] // wrap in Lazy
): CsvEncoder[A] = createEncoder { value =>
  rEncoder.value.encode(gen.to(value))
}
```

Ceci permet au compilateur de ne pas jeter l'éponge prématurément et donc permet aux types complex/récursif comme `Tree` de fonctionner :

```
CsvEncoder[Tree[Int]]
// res2: CsvEncoder[Tree[Int]] = $anon$1@5d5d4072
```

3.5 Débugger les résolutions d'implicites

Les erreurs dans la résolution d'implicites peuvent être déconcertantes et frustrantes. Voici quelques techniques à utiliser quand les choses vont mal.

3.5.1 Débugger en utilisant *implicitly*

Que peut-on faire quand les compilateurs ne trouvent simplement pas la valeur implicite ? L'erreur peut être causée par la résolution de n'importe quel implicit. Par exemple :

```
case class Foo(bar: Int, baz: Float)
```

```
CsvEncoder[Foo]
// <console>:31: error: could not find implicit value for parameter
     enc: CsvEncoder[Foo]
//        CsvEncoder[Foo]
//                  ^
```

Nous avons une erreur car nous navons pas défini `CsvEncoder` pour `Float`. Pourtant, cela n'est peut-être pas évident à voir dans le code de l'application. On peut chercher l'erreur en imaginant comment l'implicite est censé se développer, insérer des appels à `CsvEncoder.apply` ou à `implicitly` au-dessus de l'erreur pour voir si cela compile. Commençons avec la représentation générique de `Foo`:

```
CsvEncoder[Int :: Float :: HNil]
// <console>:29: error: could not find implicit value for parameter
     enc: CsvEncoder[shapeless.::[Int,shapeless.::[Float,shapeless.
     HNil]]]
//        CsvEncoder[Int :: Float :: HNil]
```

```
//                 ^
```

L'erreur ici nous en dit un peu plus, on doit continuer à chercher plus profondément dans la chaînes d'appel des implicites. L'étape suivante est de tester les composants de `HList` :

```
CsvEncoder[Int]
```

```
CsvEncoder[Float]
// <console>:29: error: could not find implicit value for parameter
     enc: CsvEncoder[Float]
//        CsvEncoder[Float]
//                  ^
```

`Int` fonctionne mais `Float` lève une erreur. `CsvEncoder[Float]` est une feuille dans le développement de notre abre, on sait donc que l'on doit commencer par implémenter l'instance manquante. Si ajouter l'instance ne corrige pas le problème, on répète le processus pour trouver le prochain point problématique.

3.5.2 Debugger en utilisant *reify*

La méthode `reify` de `scala.reflect` prend une expression Scala en paramètre et retourne un objet AST représentant l'expression sous forme d'abre avec toutes les annotations de types.

```
import scala.reflect.runtime.universe._
```

```
println(reify(CsvEncoder[Int]))
// Expr[CsvEncoder[Int]]($read.$iw.$iw.$iw.$iw.CsvEncoder.apply[Int](
     $read.$iw.$iw.$iw.$iw.intEncoder))
```

Les types inferés pendant la résolution d'implicite peuvent nous donner un indice sur le problème. Après la résolution d'implicite, tous les types existentiels

restants comme A ou T indiquent que quelque-chose n'a pas bien fonctionné. Les types "top" et "bottom" comme `Any` and `Nothing` sont aussi la preuve d'une erreur.

3.6 En résumé

Dans ce chapitre nous avons expliqué comment utiliser `Generic`, `HLists`, et `Coproducts` pour déduire automatiquement l'instance d'une *type class*. Nous avons couvert aussi le type `Lazy` qui est un moyen de manipuler les types complexes/recursifs. Compte tenu de tout ceci, on peut écrire le squelette commun suivant qui permet de déduire des instances de *type class*.

Premièrement, définissons la *type class* :

```
trait MyTC[A]
```

Définissons les instances pour les types primitifs :

```
implicit def intInstance: MyTC[Int] = ???
implicit def stringInstance: MyTC[String] = ???
implicit def booleanInstance: MyTC[Boolean] = ???
```

Définissons les instances pour `HList`:

```
import shapeless._

implicit def hnilInstance: MyTC[HNil] = ???

implicit def hlistInstance[H, T <: HList](
  implicit
  hInstance: Lazy[MyTC[H]], // wrap in Lazy
  tInstance: MyTC[T]
): MyTC[H :: T] = ???
```

Si nécessaire, définissons les instances de `Coproduct`:

```
implicit def cnilInstance: MyTC[CNil] = ???

implicit def coproductInstance[H, T <: Coproduct](
  implicit
  hInstance: Lazy[MyTC[H]], // wrap in Lazy
  tInstance: MyTC[T]
): MyTC[H :+: T] = ???
```

Enfin, définissons les instances pour `Generic`:

```
implicit def genericInstance[A, R](
  implicit
  generic: Generic.Aux[A, R],
  rInstance: Lazy[MyTC[R]] // wrap in Lazy
): MyTC[A] = ???
```

Dans le chapitre suivant nous verrons la théorie et des patterns de programmation utiles pour écrire ce genre de code. Dans le Chapitre 5, nous reviendrons sur la déduction de *type class* en utilisant une variante de `Generic` qui permet d'inspecter les champs et les noms des types dans nos *ADT*s.

Chapter 4

Travailler avec les types et les implicites

Dans le chapitre précédent, nous avons vu l'un des cas d'utilisation les plus fascinants de shapeless : la déduction automatique d'instance de case class. Nous allons découvrir qu'il existe plusieurs autres exemples au moins aussi saisissants. Cependant, avant d'en arriver là, nous devons prendre un peu de temps pour traiter la théorie que nous avons évité jusqu'ici et établir un ensemble de patterns pour écrire et débugger du code hautement implicite et hautement typé.

4.1 Types dépendants

Dans le dernier chapitre, nous avons passé beaucoup de temps à utiliser `Generic`, la *type class* qui sert à lier un type *ADT* à sa représentation générique. Pourtant, nous n'avons pas encore abordé la théorie sous-jacente à `Generic` et à tout shapeless : *les types dépendants*.

Pour illustrer tout ca, intéressons-nous de plus près à `Generic`. Voici une version simplifié de sa définition :

```
trait Generic[A] {
  type Repr
  def to(value: A): Repr
  def from(value: Repr): A
}
```

Les instances de `Generic` font référence à deux autres types : un paramètre de type `A` et un membre de type `Repr`. Imaginons que l'on implémente une méthode `getRepr` de la façon suivante. Quel type allons-nous obtenir en retour ?

```
import shapeless.Generic

def getRepr[A](value: A)(implicit gen: Generic[A]) =
  gen.to(value)
```

La réponse est : tout dépend de l'instance de `gen` que nous avons. En développant l'appel de `getRepr`, le compilateur va chercher un `Generic[A]` et le type sera le `Repr` défini dans l'instance :

```
case class Vec(x: Int, y: Int)
case class Rect(origin: Vec, size: Vec)
```

```
getRepr(Vec(1, 2))
// res1: shapeless.::[Int,shapeless.::[Int,shapeless.HNil]] = 1 :: 2
     :: HNil

getRepr(Rect(Vec(0, 0), Vec(5, 5)))
// res2: shapeless.::[Vec,shapeless.::[Vec,shapeless.HNil]] = Vec(0,0)
      :: Vec(5,5) :: HNil
```

Ce que nous voyons ici est appelé *typage dependent*: le type du résultat de `getRepr` dépend de la valeur de son paramètre de type via son membre de type. Imaginons que nous avons spécifié `Repr` comme paramètre de type de `Generic` à la place du membre de type:

```
trait Generic2[A, Repr]

def getRepr2[A, R](value: A)(implicit generic: Generic2[A, R]): R =
  ???
```

Nous aurions dû passer la valeur désirée de `Repr` dans le paramètre de type de `getRepr`, ce qui finit par rendre `getRepr` inutile. On peut en déduire que les paramètres de type sont utiles en tant « qu'inputs » et les membre de type sont utiles en tant « qu'outputs. »

4.2 Fonctions à type dépendant

Shapeless utilise les types dépendants partout : dans `Generic`, dans `Witness` (que nous traiterons dans le chapitre suivant), et dans de nombreux type classes « ops » que nous traiterons dans la Partie II de ce guide. Par exemple, shapeless fournit une case class appelée `Last` qui retourne le dernier élément d'une `HList`. Voici une version simplifiée de son implémentation :

```
package shapeless.ops.hlist

trait Last[L <: HList] {
  type Out
  def apply(in: L): Out
}
```

On peut invoquer des instances de `Last` pour inspecter les `HLists` dans notre code. Notez que dans les deux exemples ci-dessous les types de `Out` sont dépendants des types des `HList` :

```
import shapeless.{HList, ::, HNil}

import shapeless.ops.hlist.Last
```

```
val last1 = Last[String :: Int :: HNil]
// last1: shapeless.ops.hlist.Last[shapeless.::[String,shapeless.::[
     Int,shapeless.HNil]]]{type Out = Int} = shapeless.ops.
     hlist$Last$$anon$34@e4a7ace

val last2 = Last[Int :: String :: HNil]
// last2: shapeless.ops.hlist.Last[shapeless.::[Int,shapeless.::[
     String,shapeless.HNil]]]{type Out = String} = shapeless.ops.
     hlist$Last$$anon$34@7b7fa21d
```

Une fois les instances de `Last` invoquées, et on peut les utiliser au value level via leurs méthodes `apply` :

```
last1("foo" :: 123 :: HNil)
// res1: last1.Out = 123

last2(321 :: "bar" :: HNil)
// res2: last2.Out = bar
```

Nous avons deux formes de protection contre les erreurs. L'implicite définie pour `Last` nous garantit que l'on peut construire une instance de `Last` uniquement si la `HList` en entrée est dotée d'au moins un élément :

```
Last[HNil]
// <console>:15: error: Implicit not found: shapeless.Ops.Last[
     shapeless.HNil]. shapeless.HNil is empty, so there is no last
     element.
//        Last[HNil]
//             ^
```

De plus, le paramètre de type de l'instance de `Last` vérifie qu'on lui passe bien une instance de `HList` du bon type :

```
last1(321 :: "bar" :: HNil)
// <console>:16: error: type mismatch;
//  found   : shapeless.::[Int,shapeless.::[String,shapeless.HNil]]
//  required: shapeless.::[String,shapeless.::[Int,shapeless.HNil]]
//        last1(321 :: "bar" :: HNil)
//                     ^
```

Pour faire un autre exemple, implémentons notre propre type classe, appelé `Second`, qui retourne le second élément d'une `HList` :

```
trait Second[L <: HList] {
  type Out
  def apply(value: L): Out
}

object Second {
  type Aux[L <: HList, O] = Second[L] { type Out = O }

  def apply[L <: HList](implicit inst: Second[L]): Aux[L, inst.Out] =
    inst
}
```

Ce code utilise l'agencement idiomatique décrit dans la section 3.1.2. On définit un type `Aux` dans l'objet compagnon aux cotés de la méthode standard `apply`, qui permet d'invoquer des instances.

> *Méthode d'invocation versus "implicitly" versus "the"*
>
> Remarquez que le type de retour d'`apply` est `Aux[L, O]` et non pas `Second[L]`. C'est important. Utiliser `Aux` assure que la methode `apply` n'écrase pas les membres de type de l'instance invoquée. Si l'on définit `Second[L]` comme type de retour, le membre de type `Out` sera perdu dans le type retourné et la *type class* ne fonctionnera plus correctement.
>
> La méthode `implicitly` de `scala.Predef` est dotée de cette propriété. Comparons le type d'une instance de `Last` invoqué par `implicitly` :
>
> ```
> implicitly[Last[String :: Int :: HNil]]
> // res6: shapeless.ops.hlist.Last[shapeless.::[String,shapeless
>     .::[Int,shapeless.HNil]]] = shapeless.ops.
>     hlist$Last$$anon$34@4e55db2f
> ```
>
> au type retourné par `Last.apply` :

```
Last[String :: Int :: HNil]
// res7: shapeless.ops.hlist.Last[shapeless.::[String,shapeless
     .::[Int,shapeless.HNil]]]{type Out = Int} = shapeless.ops.
     hlist$Last$$anon$34@6399f8bb
```

Le type invoqué par `implicitly` n'a pas de membre de type `Out`. C'est pourquoi on doit éviter d'utiliser `implicitly` lorsque l'on travaille avec des fonctions à type dépendant. On peut utiliser une méthode d'invocation custom ou directement la méthode `the` de shapeless :

```
import shapeless._

the[Last[String :: Int :: HNil]]
// res8: shapeless.ops.hlist.Last[shapeless.::[String,shapeless
     .::[Int,shapeless.HNil]]]{type Out = Int} = shapeless.ops.
     hlist$Last$$anon$34@10db087b
```

Nous n'avons besoin que d'une seule instance définie pour une `HLists` d'au moins deux éléments :

```
implicit def hlistSecond[A, B, Rest <: HList]: Aux[A :: B :: Rest, B]
    =
  new Second[A :: B :: Rest] {
    type Out = B
    def apply(value: A :: B :: Rest): B =
      value.tail.head
  }
```

On peut invoquer les instances en utilisant `Second.apply` :

```
val second1 = Second[String :: Boolean :: Int :: HNil]
// second1: Second[shapeless.::[String,shapeless.::[Boolean,shapeless
     .::[Int,shapeless.HNil]]]]{type Out = Boolean} = $anon$1@c949ddc

val second2 = Second[String :: Int :: Boolean :: HNil]
// second2: Second[shapeless.::[String,shapeless.::[Int,shapeless.::[
     Boolean,shapeless.HNil]]]]{type Out = Int} = $anon$1@7e456faa
```

L'invocation est sujette aux mêmes contraintes que `Last`. Si l'on essaie

d'invoquer une instance pour une `HList` incompatible, alors la résolution échoue et l'on obtient une erreur de compilation :

```
Second[String :: HNil]
// <console>:27: error: could not find implicit value for parameter
     inst: Second[shapeless.::[String,shapeless.HNil]]
//         Second[String :: HNil]
//                ^
```

Les instances invoquées avec la méthode `apply` fonctionne avec les types de `HList` appropriés au niveau des valeurs (value level)

```
second1("foo" :: true :: 123 :: HNil)
// res10: second1.Out = true

second2("bar" :: 321 :: false :: HNil)
// res11: second2.Out = 321
```

```
second1("baz" :: HNil)
// <console>:28: error: type mismatch;
//  found   : shapeless.::[String,shapeless.HNil]
//  required: shapeless.::[String,shapeless.::[Boolean,shapeless.::[
     Int,shapeless.HNil]]]
//         second1("baz" :: HNil)
//                         ^
```

4.3 Enchaîner les fonctions dépendantes

Les fonctions à type dépendant offrent un moyen de calculer un type à partir d'un autre. On peut enchaîner les fonctions à type dépendant pour effectuer un calcul nécessitant plusieurs étapes. Par exemple, on pourrait utiliser `Generic` pour calculer le `Repr` d'une case classe et utiliser `Last` pour calculer le type du dernier élément. Essayons de coder ceci :

```
def lastField[A](input: A)(
  implicit
  gen: Generic[A],
  last: Last[gen.Repr]
): last.Out = last.apply(gen.to(input))
// <console>:29: error: illegal dependent method type: parameter may
     only be referenced in a subsequent parameter section
//          gen: Generic[A],
//          ^
```

Malheureusement, notre code ne compile pas. Il s'agit du même problème que nous avions rencontré dans la section 3.2.2 avec notre définition de `genericEncoder`. Nous avions contourné le problème en remontant la variable de type non-parametré dans la liste des paramètres de types :

```
def lastField[A, Repr <: HList](input: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  last: Last[Repr]
): last.Out = last.apply(gen.to(input))
```

```
lastField(Rect(Vec(1, 2), Vec(3, 4)))
// res13: Vec = Vec(3,4)
```

En règle général, on écrit toujours du code dans ce style. En paramétrant toutes les variables de type, on permet au compilateur de les unifier avec les types appropriés. Cela vaut également pour des contraintes plus subtiles. Par exemple, imaginons qu'on veuille invoquer un `Generic` pour une case class qui porte exactement un champ. On pourrait être tenté d'écrire le code suivant :

```
def getWrappedValue[A, H](input: A)(
  implicit
  gen: Generic.Aux[A, H :: HNil]
): H = gen.to(input).head
```

Le résultat est plus pernicieux. La définition de la méthode compilera mais le compilateur ne trouvera jamais d'implicits au moment de l'appel :

```
case class Wrapper(value: Int)
```

```
getWrappedValue(Wrapper(42))
// <console>:31: error: could not find implicit value for parameter
     gen: shapeless.Generic.Aux[Wrapper,shapeless.::[H,shapeless.HNil
     ]]
//        getWrappedValue(Wrapper(42))
//                        ^
```

Le message d'erreur fait allusion au problème. L'apparition du type `H` est en fait notre indice. C'est le nom d'un des paramètres de type de la méthode, il ne devrait pas apparaître dans le type que le compilateur tente d'unifier. Le problème tient au fait que le paramètre `gen` est sur-contraint : *le compilateur ne peut pas trouver `Repr`* et* garantir sa taille en même temps. Le type `Nothing` peut aussi fournir un indice, il apparaît quand le compilateur ne parvient pas à unifier les paramètres de types covariants. La solution au problème ci-dessus est de diviser la résolution d'implicite en plusieurs étapes :

1. trouver un `Generic` avec un `Repr` approprié pour `A` ;
2. s'assurer que `Repr` a un élément en tête de type `H`.

Voici une version revisitée de la méthode utilisant `=:=` pour contraindre `Repr`:

```
def getWrappedValue[A, Repr <: HList, Head, Tail <: HList](input: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  ev: (Head :: Tail) =:= Repr
): Head = gen.to(input).head
// <console>:31: error: could not find implicit value for parameter c:
     shapeless.ops.hlist.IsHCons[gen.Repr]
//        ): Head = gen.to(input).head
//                                ^
```

Ça ne compile pas car la methode head dans le corps de la methode getWrappedValue nécessite un paramètre implicite de type `IsHCons`. Ce sont des messages d'erreur beaucoup plus simples à corriger, il nous suffit

d'apprendre à utiliser un outil de la toolbox de shapeless. `IsHCons` est une type classe de shapeless qui coupe une `HList` en une `Head` et une `Tail`. On peut utiliser `IsHCons` en lieu est place de `=:=` :

```
import shapeless.ops.hlist.IsHCons

def getWrappedValue[A, Repr <: HList, Head](in: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  isHCons: IsHCons.Aux[Repr, Head, HNil]
): Head = gen.to(in).head
```

Ça corrige le bug. Maintenant la definition de la méthode et l'appel de la méthode compile correctement :

```
getWrappedValue(Wrapper(42))
// res16: Int = 42
```

Le point important n'est pas que l'on a résolu le problème en utilisant `IsHCons`. Shapeless fournit de nombreux outils comme celui-ci (voir Chapitre 6 à 8), et on peut les compléter si nécessaire avec nos propres type classes. Le point important est la compréhension du processus que l'on utilise pour parvenir à ecrire un code qui compile et la capacité de trouver les solutions Cette section se conclue par un guide pas à pas qui résume ce que nous avons découvert jusqu'ici.

4.4 Résumé

Lorsque l'on code avec shapeless, on cherche souvent à trouver un type qui dépend de valeurs présentes dans le code. Cette relation est appelée *Types dépendants*.

Les problèmes impliquant les types dépendants peuvent être aisément exprimés via la recherche d'implicites, ce qui permet au compilateur de déterminer les types intermédaires et les types ciblés d'après un point de départ à l'emplacement d'appel.

Plusieurs étapes sont souvent nécessaires pour calculer le resultat (par exemple : utiliser `Generic` pour avoir un `Repr`, puis utiliser une autre type class pour avoir un autre type). Dans ce cas, on peut suivre quelques règles pour nous assuer que notre code compile et fonctionne comme prévu :

1. Il faut placer tout les types intermédiaires dans la liste des paramètres de types. De nombreux paramètres de type ne seront pas utilisés dans le résultat mais le compilateur en a besoin pour savoir quel type il doit unifier.

2. Le compilateur résoud les implicites de gauche à droite et effectue du backtracking s'il ne peut pas trouver une combinaison qui fonctionne. Nous devons donc écrire les implicites dans l'ordre où nous en avons besoin, à l'aide d'une ou plusieurs variables de types pour les connecter aux implicites précédents.

3. Le compilateur ne peut résoudre qu'une contrainte à la fois, nous ne devons donc pas sur-contraindre un implicite.

4. Il faut énoncer le type de retour explicitement, spécifier tout paramètre de types et tout membre de types qui pourrait être utilisé ailleurs. Les membres de types sont souvent importants, il faut donc utiliser les types `Aux` pour les préserver le cas échéant. Si on ne les énonce pas dans le type de retour ils ne seront pas disponibles pour les recherches d'implicites suivantes.

5. Le pattern du `type alias Aux` reste utile pour maintenir un code propre. Il faut faire attention aux alias `Aux` quand on utilise les outils de la toolbox shapeless et implémenter les alias `Aux` dans nos propres fonctions à types dépendants.

Quand nous trouvons une chaîne d'opération à type dépendant nous pouvons les regrouper dans une seule type class. Ceci est souvent appelé pattern « lemma » (un terme emprunté aux preuves mathématiques). Nous verrons un exemple de ce pattern dans la section 6.2.

Chapter 5

Accéder aux noms durant la déduction d'`implicit`

Souvent, les instances de *types classes* que l'on définit ont besoin d'accéder à plus de choses que les types uniquement. Dans ce chapitre nous découvririons une variante de `Generic` appelée `LabelledGeneric` qui donne accès aux noms des champs et aux noms des types.

Pour commencer, abordons un peu la théorie. `LabelledGeneric` emploie des techniques astucieuses pour exposer les informations sur les noms au niveau des types. Pour comprendre ces techniques, nous devons traiter les *types littéraux*, *types singleton*, *types fantômes* et le *type tagging*.

5.1 Types littéreaux.

Une valeur de Scala peut avoir plusieurs types. Par exemple, la string `"hello"` a au moins trois types : `String`, `AnyRef` et `Any`[1] :

[1] `String` est également doté de pluisieurs autres types tels que `Serializable` et `Comparable`, mais ignorons-les pour l'instant.

```
"hello" : String
// res0: String = hello

"hello" : AnyRef
// res1: AnyRef = hello

"hello" : Any
// res2: Any = hello
```

`"hello"` a aussi un autre type : un "type singleton" qui appartient exclusivement à cette valeur. Le résultat est similaire à ce que l'on obtient lorsqu'on définit un object compagnon.

```
object Foo
```

```
Foo
// res3: Foo.type = Foo$@11ce0152
```

Le type `Foo.type` est le type de `Foo`, et `Foo` est la seule et unique valeur pour ce type.

Les types singleton appliqués aux valeurs littérales sont appelés *types littéraux*. Ils existent depuis longtemps dans Scala, mais il n'y a normalement pas d'interaction avec eux, car le compilateur "élargit" par défaut les littéraux au type non singleton le plus proche. Par exemple, ces deux expressions sont essentiellement équivalentes :

```
"hello"
// res4: String = hello

("hello" : String)
// res5: String = hello
```

Shapeless fournit quelques outils pour travailler avec les types littéraux. Tout d'abord, la macro `narrow` convertit une expression littérale en une expression littérale typée par un singleton :

```
import shapeless.syntax.singleton._
```

```
var x = 42.narrow
// x: Int(42) = 42
```

Notez que le type de x: `Int(42)` est un type littéral. Il s'agit d'un sous-type de `Int` qui contient uniquement la valeur 42. Si l'on essaie d'assigner une valeur différente à x, on obtient une erreur de compilation :

```
x = 43
// <console>:16: error: type mismatch;
//  found   : Int(43)
//  required: Int(42)
//        x = 43
//            ^
```

Pourtant, x est toujours un sous type normal de `Int`. Si l'on fait des opérations sur x, on obtient bien un type normal pour résultat :

```
x + 1
// res6: Int = 43
```

Nous pouvons utiliser `narrow` avec n'importe quel type littéral Scala :

```
1.narrow
// res7: Int(1) = 1

true.narrow
// res8: Boolean(true) = true

"hello".narrow
// res9: String("hello") = hello

// and so on...
```

Malheureusement, on ne peut pas les utiliser dans une expression composée :

```
math.sqrt(4).narrow
// <console>:17: error: Expression scala.math.`package`.sqrt(4.0) does
      not evaluate to a constant or a stable reference value
//        math.sqrt(4).narrow
//                  ^
// <console>:17: error: value narrow is not a member of Double
//        math.sqrt(4).narrow
//                     ^
```

> *Types littéraux en Scala*
>
> Jusque récemment, Scala ne disposait pas de syntaxe pour écrire les types littéraux. Les types étaient là, dans le compilateur, mais nous ne pouvions pas les écrire directement dans le code. Mais grâce aux versions de Lightbend Scala 2.12.1, Lightbend Scala 2.11.9, et Typelevel Scala 2.11.8, nous disposons désormais d'un support direct de la syntaxe pour les types littéraux. Dans ces versions de Scala, nous pouvons écrire les déclarations de la façon suivante :
>
> ```
> val theAnswer: 42 = 42
> ```
>
> Le type `42` est le même que le type `Int(42)` que nous avons vu précédemment. Vous verrez toujours `Int(42)` dans les sorties pour des raisons de pérennité, mais la syntaxe canonique restera `42`.

5.2 Le type tagging et les types fantômes

Shapeless utilise les types littéraux pour modéliser les noms de champs des *case classes*. Pour ce faire, il "tag" les types des champs avec le type littéral de leurs noms. Avant de découvrir comment shapeless réalise cela, faisons-le nous-même pour montrer que la magie n'y est pour rien (enfin… pas vraiment). Supposons un nombre :

```
val number = 42
```

Ce nombre est un `Int` dans deux mondes : à l'execution où il a réellement une valeur et des méthodes que l'on peut appeler, et à la compilation, où le compilateur utilise le type pour calculer quelle partie de code fonctionne ensemble et l'utilise pour rechercher les implicits.

On peut modifier le type du `number` à la compilation sans modifier ces valeurs à l'execution en le "taggant" avec un "type fantôme". Les types fantômes sont des types qui ne présentent pas de sémantique à l'exécution, comme ceci :

```
trait Cherries
```

Nous pouvons tagger le `number` en utilisant `asInstanceOf`. Nous finissons avec une valeur qui est à la fois un `Int` et une `Cherries` à la compilation et uniquement un `Int` à l'execution :

```
val numCherries = number.asInstanceOf[Int with Cherries]
// numCherries: Int with Cherries = 42
```

Shapeless utilise cette astuce pour tagger les champs et les sous-types d'un *ADT* avec le type singleton de leurs noms. Si l'utilisation de `asInstanceOf` vous dérange, ne vous inquiétez pas : Shapeless fournit deux syntaxes de tagging qui vous éviteront ce désagrément.

La première syntaxe : `->>` tag l'expression à droite de la flèche avec le type singleton de l'expression littérale à droite de la flèche.

```
import shapeless.labelled.{KeyTag, FieldType}
import shapeless.syntax.singleton._

val someNumber = 123
```

```
val numCherries = "numCherries" ->> someNumber
// numCherries: Int with shapeless.labelled.KeyTag[String("numCherries
    "),Int] = 123
```

Voila comment nous taggons `someNumber` avec le type fantôme suivant :

```
KeyTag["numCherries", Int]
```

Le tag détaille à la fois le nom et le type du champ ; la combinaison des deux est utile lorsque l'on recherche des éléments dans un `Repr` en utilisant la résolution d'`implicit`.

La seconde syntaxe considère le tag comme un type plutôt qu'une valeur littérale. C'est utile lorsque vous connaissez le tag à utiliser mais que vous n'avez pas la possibilité d'écrire ce type littéral dans notre code :

```
import shapeless.labelled.field
```

```
field[Cherries](123)
// res13: shapeless.labelled.FieldType[Cherries,Int] = 123
```

`FieldType` est un alias de type qui simplifie l'extraction du tag et du type de base du type tagger :

```
type FieldType[K, V] = V with KeyTag[K, V]
```

Comme nous le verrons plus tard, Shapeless emploie ce mécanisme pour tagger les champs et les sous-types avec leurs noms dans notre code source.

Les tags n'existent qu'à la compilation et n'ont pas de représentation à l'exécution. Alors, comment pouvons-nous les convertir en valeurs utilisables à l'exécution ? Shapeless fournit à cette fin une type class appelée `Witness`[ˆwitness]. Si l'on combine `Witness` et `FieldType`, on obtient quelque chose de vraiment intéressant : la possibilité d'extraire le nom d'un champ à partir d'un champ tagger :

[ˆwitness] : le terme "witness" est emprunté aux preuves mathématiques[2]

```
import shapeless.Witness
```

[2] https://en.wikipedia.org/wiki/Witness_(mathematics)

```
val numCherries = "numCherries" ->> 123
// numCherries: Int with shapeless.labelled.KeyTag[String("numCherries
    "),Int] = 123
```

```
// Get the tag from a tagged value:
def getFieldName[K, V](value: FieldType[K, V])
    (implicit witness: Witness.Aux[K]): K =
  witness.value
```

```
getFieldName(numCherries)
// res15: String = numCherries
```

```
// Get the untagged type of a tagged value:
def getFieldValue[K, V](value: FieldType[K, V]): V =
  value
```

```
getFieldValue(numCherries)
// res17: Int = 123
```

Si nous construisons une `HList` d'éléments taggés, nous obtenons une structure de données dotées de certaines propriétés d'une `Map`. Nous pouvons faire référénce à un champ par un tag, le manipuler et le remplacer, et conserver toutes les information de types et de noms tout du long. Shapeless appelle ces structures des "records".

5.2.1 Records et *LabelledGeneric*

Les records sont des `HLists` d'éléments taggés.

```
import shapeless.{HList, ::, HNil}
```

```
val garfield = ("cat" ->> "Garfield") :: ("orange" ->> true) :: HNil
// garfield: shapeless.::[String with shapeless.labelled.KeyTag[String
    ("cat"),String],shapeless.::[Boolean with shapeless.labelled.
    KeyTag[String("orange"),Boolean],shapeless.HNil]] = Garfield ::
    true :: HNil
```

Pour être clair, le type de `garfield` est le suivant :

```
// FieldType["cat",    String]  ::
// FieldType["orange", Boolean] ::
// HNil
```

Nous n'avons pas besoin ici d'approfondir les records : en bref, les records sont les représentations génériques utilisées par `LabelledGeneric`. `LabelledGeneric` tag chaque champ ou nom de type d'un *ADT* dans des produits ou des coproduits (bien que les noms soient représentés par des `Symbols` et non des `Strings`). Shapeless fournit un ensemble d'operations semblables à celles de `Map` pour manipuler les records, certaines seront traitées dans la section 6.4. Pour l'instant, contentons-nous de déduire quelquess type class à l'aide de `LabelledGeneric`.

5.3 Déduire l'instance d'un produit avec *LabelledGeneric*

Nous allons utiliser un exemple fonctionnel d'encodage de Json pour illustrer `LabelledGeneric`. Nous allons définir une type classe `JsonEncoder` qui va convertir nos valeurs en *AST JSON*. C'est l'approche adoptée par *Argonaut*, *Circe*, *Play JSON*, *Spray JSON* et de nombreuses autres bibliothèques Scala pour le *JSON*.

Tout d'abord, définissons le type de données de notre *JSON* :

```
sealed trait JsonValue
case class JsonObject(fields: List[(String, JsonValue)]) extends
    JsonValue
case class JsonArray(items: List[JsonValue]) extends JsonValue
case class JsonString(value: String) extends JsonValue
case class JsonNumber(value: Double) extends JsonValue
```

```
case class JsonBoolean(value: Boolean) extends JsonValue
case object JsonNull extends JsonValue
```

puis la *type class* pour encoder les valeurs en *JSON* :

```
trait JsonEncoder[A] {
  def encode(value: A): JsonValue
}

object JsonEncoder {
  def apply[A](implicit enc: JsonEncoder[A]): JsonEncoder[A] = enc
}
```

puis quelques instances primitives :

```
def createEncoder[A](func: A => JsonValue): JsonEncoder[A] =
  new JsonEncoder[A] {
    def encode(value: A): JsonValue = func(value)
  }

implicit val stringEncoder: JsonEncoder[String] =
  createEncoder(str => JsonString(str))

implicit val doubleEncoder: JsonEncoder[Double] =
  createEncoder(num => JsonNumber(num))

implicit val intEncoder: JsonEncoder[Int] =
  createEncoder(num => JsonNumber(num))

implicit val booleanEncoder: JsonEncoder[Boolean] =
  createEncoder(bool => JsonBoolean(bool))
```

et quelques combinateurs d'instance :

```
implicit def listEncoder[A]
    (implicit enc: JsonEncoder[A]): JsonEncoder[List[A]] =
  createEncoder(list => JsonArray(list.map(enc.encode)))
```

```
implicit def optionEncoder[A]
    (implicit enc: JsonEncoder[A]): JsonEncoder[Option[A]] =
  createEncoder(opt => opt.map(enc.encode).getOrElse(JsonNull))
```

Idéalement, lorsqu'on encode un *ADT* en *JSON*, on voudrait utiliser les noms des champs dans la sortie *JSON* :

```
case class IceCream(name: String, numCherries: Int, inCone: Boolean)

val iceCream = IceCream("Sundae", 1, false)

// Idéalement, on voudrait produire quelque chose comme suit :
val iceCreamJson: JsonValue =
  JsonObject(List(
    "name"        -> JsonString("Sundae"),
    "numCherries" -> JsonNumber(1),
    "inCone"      -> JsonBoolean(false)
  ))
```

C'est à ce moment que `LabelledGeneric` devient utile. Invoquons une instance de `IceCream` et regardons sa représentation :

```
import shapeless.LabelledGeneric
```

```
val gen = LabelledGeneric[IceCream].to(iceCream)
// gen: shapeless.::[String with shapeless.labelled.KeyTag[Symbol with
      shapeless.tag.Tagged[String("name")],String],shapeless.::[Int
     with shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[
     String("numCherries")],Int],shapeless.::[Boolean with shapeless.
     labelled.KeyTag[Symbol with shapeless.tag.Tagged[String("inCone")
     ],Boolean],shapeless.HNil]]] = Sundae :: 1 :: false :: HNil
```

Pour plus de clareté voici le type complet de la `HList`:

```
// String  with KeyTag[Symbol with Tagged["name"], String]     ::
// Int     with KeyTag[Symbol with Tagged["numCherries"], Int] ::
```

```
// Boolean with KeyTag[Symbol with Tagged["inCone"], Boolean]  ::
// HNil
```

Ce type est un peu plus complexe que ce que nous avons vu jusqu'à présent. Plutôt que représenter les noms des champs avec des types string littéraux, shapeless les représente avec des symbôles taggés grâce à des types littéraux. Les détails d'implémentation ne sont pas particulièrement importants : on peut toujours utiliser `Witness` et `FieldType` pour extraire les tags, mais ils ressortiront en `Symbols` et non pas en `Strings`[3].

5.3.1 Les instances de *HLists*

Définissons un instance de `JsonEncoder` pour `HNil` et `::`. Nos encodeurs vont générer et manipuler des `JsonObjects`, nous allons donc créer un nouveau type d'encodeur pour nous faciliter la vie :

```
trait JsonObjectEncoder[A] extends JsonEncoder[A] {
  def encode(value: A): JsonObject
}

def createObjectEncoder[A](fn: A => JsonObject): JsonObjectEncoder[A]
    =
  new JsonObjectEncoder[A] {
    def encode(value: A): JsonObject =
      fn(value)
  }
```

La définition de `HNil` devient donc triviale :

```
import shapeless.{HList, ::, HNil, Lazy}

implicit val hnilEncoder: JsonObjectEncoder[HNil] =
  createObjectEncoder(hnil => JsonObject(Nil))
```

La définitoon de `hlistEncoder` ajoute un peu de complexité, nous allons donc avancer pas à pas. Commençons avec la définition à laquelle on pourrait s'attendre si nous utilisions un `Generic`:

[3] Les futures versions de shapeless pourraient utiliser les `Strings` comme tags.

```
implicit def hlistObjectEncoder[H, T <: HList](
  implicit
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonEncoder[H :: T] = ???
```

`LabelledGeneric` nous donne une `HList` qui contient des types taggés, commençons donc par introduire une nouvelle variable de types pour le type de la clef :

```
import shapeless.Witness
import shapeless.labelled.FieldType

implicit def hlistObjectEncoder[K, H, T <: HList](
  implicit
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = ???
```

Dans le corps de notre méthode, nous aurons besoin de la valeur associée à K. Nous allons ajouter un `Witness` implicite qui le fera pour nous :

```
implicit def hlistObjectEncoder[K, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName = witness.value
  ???
}
```

On peut accéder à la valeur de K en utilisant `witness.value`, mais le compilateur n'a aucun moyen de savoir quel type de tag nous allons obtenir. `LabelledGeneric` utilise `Symbols` pour les tags, nous allons donc mettre un type bound a K et utiliser `symbol.name` pour le convertir en `String`:

```
implicit def hlistObjectEncoder[K <: Symbol, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName: String = witness.value.name
  ???
}
```

Le reste de la définition utilise les principes que nous avons abordés dans le chapitre 3:

```
implicit def hlistObjectEncoder[K <: Symbol, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName: String = witness.value.name
  createObjectEncoder { hlist =>
    val head = hEncoder.value.encode(hlist.head)
    val tail = tEncoder.encode(hlist.tail)
    JsonObject((fieldName, head) :: tail.fields)
  }
}
```

5.3.2 Les instances pour les produits concrêt.

Enfin, revenons à notre instance générique. Elle est identique aux définitions que nous avons vues précédemment, à l'exception que nous utilisons `LabelledGeneric` à la place de `Generic`:

```
import shapeless.LabelledGeneric

implicit def genericObjectEncoder[A, H](
  implicit
  generic: LabelledGeneric.Aux[A, H],
  hEncoder: Lazy[JsonObjectEncoder[H]]
```

```
): JsonEncoder[A] =
  createObjectEncoder { value =>
    hEncoder.value.encode(generic.to(value))
  }
```

Et voilà tout ce dont nous avons besoin ! Avec ces définitions en place nous pouvons sérialiser les instances de n'importe quelle case class et conserver les noms des champs dans le *JSON* obtenu :

```
JsonEncoder[IceCream].encode(iceCream)
// res14: JsonValue = JsonObject(List((name,JsonString(Sundae)), (
     numCherries,JsonNumber(1.0)), (inCone,JsonBoolean(false))))
```

5.4 Déduire les instances de coproduits avec *LabelledGeneric*

Appliquer les `LabelledGeneric` aux `Coproducts` revient à mélanger des concepts que nous avons déja traités. Commençons par examiner un type `Coproduct` déduit par `LabelledGeneric`. Nous allons revisiter notre *ADT* Shape du Chapitre 3:

```
import shapeless.LabelledGeneric

sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends
     Shape
final case class Circle(radius: Double) extends Shape
```

```
LabelledGeneric[Shape].to(Circle(1.0))
// res5: shapeless.:+:[Rectangle with shapeless.labelled.KeyTag[Symbol
      with shapeless.tag.Tagged[String("Rectangle")],Rectangle],
     shapeless.:+:[Circle with shapeless.labelled.KeyTag[Symbol with
     shapeless.tag.Tagged[String("Circle")],Circle],shapeless.CNil]] =
      Inr(Inl(Circle(1.0)))
```

Voici le type du `Coproduct` dans un format plus lisible :

```
// Rectangle with KeyTag[Symbol with Tagged["Rectangle"], Rectangle]
     :+:
// Circle    with KeyTag[Symbol with Tagged["Circle"],    Circle]
     :+:
// CNil
```

Comme vous pouvez le constater, le résultat est un `Coproduit` des sous-types de Shape, chacun taggé avec leur nom. Nous pouvons utiliser ces informations pour écrire un `JsonEncoders` pour `:+:` et `CNil`:

```
import shapeless.{Coproduct, :+:, CNil, Inl, Inr, Witness, Lazy}
import shapeless.labelled.FieldType

implicit val cnilObjectEncoder: JsonObjectEncoder[CNil] =
  createObjectEncoder(cnil => throw new Exception("Inconceivable!"))

implicit def coproductObjectEncoder[K <: Symbol, H, T <: Coproduct](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :+: T] = {
  val typeName = witness.value.name
  createObjectEncoder {
    case Inl(h) =>
      JsonObject(List(typeName -> hEncoder.value.encode(h)))

    case Inr(t) =>
      tEncoder.encode(t)
  }
}
```

`coproductEncoder` suit les mêmes règles que `hlistEncoder`. On dispose de trois paramètres de types : K pour le nom du type, H pour la valeur de la tête de la `HList` et T pour la valeur de la queue. Nous utilisons `FieldType` et `:+:` dans le type résultant pour garder la relation entre les trois, et nous utilisons `Witness` pour accéder à la valeur du nom du type pendant l'exécution. Le résultat est un objet contenant une seule paire de clef/valeur, la clef étant le nom du type et la valeur le résultat :

```
val shape: Shape = Circle(1.0)
```

```
JsonEncoder[Shape].encode(shape)
// res8: JsonValue = JsonObject(List((Circle,JsonObject(List((radius,
     JsonNumber(1.0)))))))
```

D'autres encodages sont possibles avec un peu plus de travail. Par exemple, on peut ajouter un champ `"type"` dans la sortie, ou encore permetre à l'utilisateur de configurer le format. La code base de Sam Halliday sur spray-json-shapeless[4] est un parfait exemple de code qui est accessible tout en offrant beaucoup de flexibilité.

5.5 Résumé

Dans ce chapitre, nous avons abordé `LabelledGeneric`, une variante de `Generic` qui donne accès aux noms des champs et aux noms des types dans sa représentation générique.

Les noms accessible grâce à `LabelledGeneric` sont encodés par des tags au type-level afin de les utiliser durant la résolution d'implicites. Nous avons commencer ce chapitre en abordant les *types littéraux* et la façon dont shapeless les utilise dans ces tags. Nous avons également abordé la *type class* `Witness` utilisée pour extraire les valeurs des types littéraux.

Enfin, nous avons combiné `LabelledGeneric`, les types littéraux et `Witness` pour construire une bibliothèque d'encodage *JSON* qui permet d'inclure les noms présents dans les *ADT* dans la sortie *JSON*.

Le plus important dans ce chapitre est que tout ce code n'utilise jamais la réflexion. Tout est implémenté avec des types, des implicits et le petit ensemble de macros interne à shapeless. Le code que nous générons est par conséquent très rapide à exécuter.

[4] https://github.com/fommil/spray-json-shapeless

Part II

Shapeless ops

Chapter 6

Travailler avec *HLists* et *Coproducts*

Dans la Partie I nous avons abordé des méthodes visant à déduire des instances de type class pour des types de données algébriques. Nous pouvons utiliser la déduction automatique d'instance pour enrichir presque toutes les type class, cependant dans les cas les plus compliqués, cela nous amène à rédiger beaucoup de code pour manipuler les `HLists` et les `Coproducts`.

Dans la Partie II, nous allons regarder de plus près le package `shapeless.ops`, il fournit un ensemble utile d'outils que l'on peut utiliser comme brique de construction pour nos programmes.

Chacun des op se présente en deux parties : une *type class* que nous pouvons utiliser pendant la résolution d'implicite et des *méthodes d'extension* que nous pouvons appeler sur `HList` et `Coproduct`.

Il existe trois ensembles d'ops, chacun disponible dans un des trois packages suivants :

- `shapeless.ops.hlist` définit des type classes pour `HLists`. Elles peuvent être utilisées directement sur les `HList` via les méthodes d'extension définies dans `shapeless.syntax.hlist`.

- `shapeless.ops.coproduct` définit des type classes pour `Coproducts`. Elles peuvent être utilisées directement sur les `Coproduct` via les méthodes d'extension définies dans `shapeless.syntax.coproduct`.
- `shapeless.ops.record` définit des type classes pour shapeless records (des `HLists` contenant des éléments taggés : Section 5.2). Elles peuvent être utilisées directement sur les `HLists` importées de `shapeless.record` via les méthodes d'extension définies dans `shapeless.syntax.record`.

Nous n'avons pas le temps dans ce livre d'aborder tous les ops disponibles. Par chance, dans la majorité des cas le code est compréhensible et bien documenté. Plutôt que d'établir un guide exhaustif, nous allons aborder les points théoriques et structurels les plus importants et vous montrer comment obtenir davantage d'informations de la base de code de shapeless.

6.1 Exemple d'ops simples

`HList` dispose d'une méthode d'extension `init` et d'un autre `last` basé sur deux type classes : `shapeless.ops.hlist.Init` et `shapeless.ops.hlist.Last`. `Coproduct` a des méthodes et des type classes similaires. Ils représentent un exemple parfait du pattern ops. Voici une définition simplifiée des méthodes d'extension :

```
package shapeless
package syntax

implicit class HListOps[L <: HList](l : L) {
  def last(implicit last: Last[L]): last.Out = last.apply(l)
  def init(implicit init: Init[L]): init.Out = init.apply(l)
}
```

Le type de retour de chaque méthode est déterminé par le type dépendant du paramètre implicite. Ce sont les instances de chaque type class qui fournissent la relation. Voici le squelette de l'implémentation de `Last` en guise d'exemple :

```
trait Last[L <: HList] {
  type Out
  def apply(in: L): Out
}

object Last {
  type Aux[L <: HList, O] = Last[L] { type Out = O }
  implicit def pair[H]: Aux[H :: HNil, H] = ???
  implicit def list[H, T <: HList]
    (implicit last: Last[T]): Aux[H :: T, last.Out] = ???
}
```

Nous pouvons faire quelques observations intéressantes sur cette implémentation. Premièrement, nous pouvons généralement implémenter les type classes ops avec un petit nombre d'instances (seulement deux dans cet exemple). Nous pouvons ainsi rassembler *toutes* les instances requises dans l'objet compagnon de la *type class*, ce qui nous permet d'appeler les methodes d'extention de `shapeless.ops` qui n'ont aucun rapport :

```
import shapeless._
```

```
("Hello" :: 123 :: true :: HNil).last
// res0: Boolean = true

("Hello" :: 123 :: true :: HNil).init
// res1: shapeless.::[String,shapeless.::[Int,shapeless.HNil]] = Hello
      :: 123 :: HNil
```

Ensuite, la *type class* n'est définie que pour les `HLists` qui contiennent au moins un élément. Ce qui nous donne quelques vérifications statiques. Si on essaie d'appeler `last` sur une `HList` vide, cela provoque une erreur de compilation :

```
HNil.last
// <console>:16: error: Implicit not found: shapeless.Ops.Last[
     shapeless.HNil.type]. shapeless.HNil.type is empty, so there is
     no last element.
//        HNil.last
```

```
//            ^
```

6.2 Faire son propre op (le patterne “lemma”)

Si l’on trouve un ensemble d’ops utile, on peut les regrouper sous la forme d’une autre type classe ops. C’est un exemple du pattern « lemma », un terme que nous avions introduit dans la section 4.4.

Prenons comme exercise la création de notre propre op. Nous allons combiner la puissance de `Last` et de `Init` pour créer la *type class* `Penultimate` qui retrouve l’avant-dernier élément d’une `HList`. Voici la définition de la *type class*, complétée par son `type alias` `Aux` et sa méthode `apply`:

```
import shapeless._

trait Penultimate[L] {
  type Out
  def apply(l: L): Out
}

object Penultimate {
  type Aux[L, O] = Penultimate[L] { type Out = O }

  def apply[L](implicit p: Penultimate[L]): Aux[L, p.Out] = p
}
```

Notez encore une fois que la méthode `apply` a comme type de retour `Aux[L, O]` au lieu de `Penultimate[L]`. Ce qui garanti que le membre de type est visible dans les instances invoquées, comme souligné dans la section 4.2.

Nous n’avons besoin que de définir une seule instance de `Penultimate`, qui combine `Init` et `Last` de la même façon que dans la section 4.3:

```
import shapeless.ops.hlist

implicit def hlistPenultimate[L <: HList, M <: HList, O](
  implicit
  init: hlist.Init.Aux[L, M],
```

```
  last: hlist.Last.Aux[M, O]
): Penultimate.Aux[L, O] =
  new Penultimate[L] {
    type Out = O
    def apply(l: L): O =
      last.apply(init.apply(l))
  }
```

Nous pouvons utiliser `Penultimate` de la façon suivante :

```
type BigList = String :: Int :: Boolean :: Double :: HNil

val bigList: BigList = "foo" :: 123 :: true :: 456.0 :: HNil
```

```
Penultimate[BigList].apply(bigList)
// res4: Boolean = true
```

Invoquer une instance de `Penultimate` oblige le compilateur à invoquer des instances pour `Last` et `Init`, nous bénéficions donc du même niveau de vérification des types sur les `HLists` courtes :

```
type TinyList = String :: HNil

val tinyList = "bar" :: HNil
```

```
Penultimate[TinyList].apply(tinyList)
// <console>:22: error: could not find implicit value for parameter p:
      Penultimate[TinyList]
//        Penultimate[TinyList].apply(tinyList)
//                    ^
```

Nous pouvons rendre les chose plus simples pour les utilisateurs finaux si nous définissons une méthode d’extension sur `HList`:

```
implicit class PenultimateOps[A](a: A) {
  def penultimate(implicit inst: Penultimate[A]): inst.Out =
    inst.apply(a)
}
```

```
bigList.penultimate
// res7: Boolean = true
```

Nous pouvons aussi fournir `Penultimate` pour tous les types produit en fournissant une instance basée sur `Generic` :

```
implicit def genericPenultimate[A, R, O](
  implicit
  generic: Generic.Aux[A, R],
  penultimate: Penultimate.Aux[R, O]
): Penultimate.Aux[A, O] =
  new Penultimate[A] {
    type Out = O
    def apply(a: A): O =
      penultimate.apply(generic.to(a))
  }

case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

```
IceCream("Sundae", 1, false).penultimate
// res9: Int = 1
```

Voici le point important à retenir : en définissant `Penultimate` comme une autre type class, nous avons créé un outil réutilisable que nous pouvons utiliser ailleurs. Shapeless fournit de nombreux outils pour de nombreux usages, mais il est simple d'ajouter le nôtre à la boîte à outils.

6.3 Étude de cas : migration de case class

La puissance des type classes ops se conrétise quand on assemble dans notre propre code. Nous allons conclure ce chapitre avec un exemple probant : une

type classe visant à effectuer des « migrations » (ou « évolutions ») de *case classes*[1]. Par exemple, si la version 1 de notre application contient la case classe suivante :

```
case class IceCreamV1(name: String, numCherries: Int, inCone: Boolean)
```

Notre bibliothèque de migration doit pouvoir faire certaines mises à jour « mécanique » pour nous :

```
// Remove fields:
case class IceCreamV2a(name: String, inCone: Boolean)

// Reorder fields:
case class IceCreamV2b(name: String, inCone: Boolean, numCherries: Int
    )

// Insert fields (provided we can determine a default value):
case class IceCreamV2c(
  name: String, inCone: Boolean, numCherries: Int, numWaffles: Int)
```

Idéalement on aimerait être capables d'écrire le code suivant :

```
IceCreamV1("Sundae", 1, false).migrateTo[IceCreamV2a]
```

La *type class* doit prendre soin de la migration sans boilerplate additionel.

6.3.1 La *type class*

La *type class* `Migration` représente la transformation d'une source vers un type destination. Ils seront tous les deux les paramètres de type pour notre déduction. Nous n'avons pas besoin d'alias de type `Aux` car il n'y a pas de membre de type à exposer :

[1] Le terme est tiré de « migrations de base de donnée » : des scripts SQL qui automatise la mise à jour des schémas de base de données.

```
trait Migration[A, B] {
  def apply(a: A): B
}
```

Nous allons également ajouter une méthode d'extension pour rendre les exemples plus simples à lire :

```
implicit class MigrationOps[A](a: A) {
  def migrateTo[B](implicit migration: Migration[A, B]): B =
    migration.apply(a)
}
```

6.3.2 Étape 1. Enlever les champs

Construisons notre solution pièce par pièce, en commençant par la suppression de champs. Nous pouvons y arriver en plusieurs étapes :

1. convertir `A` dans sa représentation générique ;
2. filter la `HList` de l'étape 1, garder uniquement les champs qui sont présents dans `A` et `B` ;
3. convertir le résultat de l'étape 2 en `B`.

Nous pouvons implémenter l'étape 1 et 3 avec `Generic` ou `LabelledGeneric`, l'étape 2 peut l'être avec un op appelé `Intersection`. `LabelledGeneric` semble être un choix intéressant car nous avons besoin d'identifier le nom des champs :

```
import shapeless._
import shapeless.ops.hlist

implicit def genericMigration[A, B, ARepr <: HList, BRepr <: HList](
  implicit
  aGen  : LabelledGeneric.Aux[A, ARepr],
  bGen  : LabelledGeneric.Aux[B, BRepr],
  inter : hlist.Intersection.Aux[ARepr, BRepr, BRepr]
): Migration[A, B] = new Migration[A, B] {
  def apply(a: A): B =
```

```
    bGen.from(inter.apply(aGen.to(a)))
}
```

Prenez un moment pour localiser `Intersection`[2] dans la base de code de shapeless. L'alias de type `Aux` prend en compte trois paramètres : deux `HLists` qui sont les entrées et une pour le type du résultat de l'intersection. Dans l'exemple au-dessus nous spécifions `ARepr` et `BRepr` comme type d'entrée et `BRepr` comme type de retour. Ce qui signifie que la résolution d'implicite ne fonctionnera que si `B` contient l'exact sous-ensemble des champs de `A`, avec exactement les mêmes noms et dans le même ordre :

```
IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2a]
// res6: IceCreamV2a = IceCreamV2a(Sundae,true)
```

Nous obtenons une erreur de compilation si nous essayons d'utiliser `Migration` avec un type non conforme :

```
IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2b]
// <console>:24: error: could not find implicit value for parameter
     migration: Migration[IceCreamV1,IceCreamV2b]
//        IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2b]
//                                                ^
```

6.3.3 Etape 2. Réordonner les champs

Nous avons besoin d'une autre ops type class pour pouvoir faire du réordonnement. L'op `Align`[3] nous permet de réordonner les champs d'une `HList` pour faire correspondre l'ordre d'une autre `HList`. Nous pouvons redéfinir notre instance en utilisant `Align` de la manière suivante :

[2] https://github.com/milessabin/shapeless/blob/shapeless-2.3.2/core/src/main/scala/shapeless/ops/hlists.scala#L1297-L1352

[3] https://github.com/milessabin/shapeless/blob/shapeless-2.3.2/core/src/main/scala/shapeless/ops/hlists.scala#L1973-L1997

```
implicit def genericMigration[
  A, B,
  ARepr <: HList, BRepr <: HList,
  Unaligned <: HList
](
  implicit
  aGen    : LabelledGeneric.Aux[A, ARepr],
  bGen    : LabelledGeneric.Aux[B, BRepr],
  inter   : hlist.Intersection.Aux[ARepr, BRepr, Unaligned],
  align   : hlist.Align[Unaligned, BRepr]
): Migration[A, B] = new Migration[A, B] {
  def apply(a: A): B =
    bGen.from(align.apply(inter.apply(aGen.to(a))))
}
```

Nous ajoutons un nouveau paramètre de type appelé `Unaligned` pour représenter l'intersection de `ARepr` et `BRepr` avant l'alignement, on utilise `Align` pour convertir `Unaligned` en `BRepr`. Avec cette modification de `Migration` nous pouvons à la fois retirer mais aussi réordonner les champs :

```
IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2a]
// res8: IceCreamV2a = IceCreamV2a(Sundae,true)

IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2b]
// res9: IceCreamV2b = IceCreamV2b(Sundae,true,1)
```

Cependant, si on essaie d'ajouter un champ, on obtient encore une erreur :

```
IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2c]
// <console>:26: error: could not find implicit value for parameter
     migration: Migration[IceCreamV1,IceCreamV2c]
//        IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2c]
//                                              ^
```

6.3.4 Etape 3. Ajouter de nouveaux champs

Pour soutenir l'ajout de champs nous avons besoin d'un mécanisme pour fournir les valeurs par défaut. Shapeless ne fournit pas une type class pour

cette raison, mais Cats le fait, sous la forme d'un `Monoid`. En voici une définition simplifiée :

```
package cats

trait Monoid[A] {
  def empty: A
  def combine(x: A, y: A): A
}
```

Un `Monoid` dispose de deux opérations : `empty` pour créer une valeur « zéro » et `combine` pour « fusionner » deux valeurs en une seule. Dans notre code nous n'avons besoin que de `empty` mais il serait trivial de définir également`combine`.

Cats fournit une instance de `Monoid` pour chacun des type primitf qui nous intéresse (`Int`, `Double`, `Boolean`, and `String`). Nous pouvons définir une instance pour `HNil`et `::` en utilisant les techniques du Chapitre 5:

```
import cats.Monoid
import cats.instances.all._
import shapeless.labelled.{field, FieldType}

def createMonoid[A](zero: A)(add: (A, A) => A): Monoid[A] =
  new Monoid[A] {
    def empty = zero
    def combine(x: A, y: A): A = add(x, y)
  }

implicit val hnilMonoid: Monoid[HNil] =
  createMonoid[HNil](HNil)((x, y) => HNil)

implicit def emptyHList[K <: Symbol, H, T <: HList](
  implicit
  hMonoid: Lazy[Monoid[H]],
  tMonoid: Monoid[T]
): Monoid[FieldType[K, H] :: T] =
  createMonoid(field[K](hMonoid.value.empty) :: tMonoid.empty) {
    (x, y) =>
      field[K](hMonoid.value.combine(x.head, y.head)) ::
        tMonoid.combine(x.tail, y.tail)
```

```
  }
```

Nous devons combiner `Monoid`[4] avec quelques autre ops pour compléter notre implémentation finale de `Migration`. Voici la liste complète des étapes :

1. utiliser `LabelledGeneric` pour convertir `A` dans sa représentation générique ;
2. utiliser `Intersection` pour calculer `HList` des champs communs entre `A` et `B` ;
3. calculer les types qui apparaissent dans `B` mais pas dans `A` ;
4. utiliser `Monoid` pour calculer une valeur par défaut au type de l'étape 3 ;
5. ajouter les champs de l'étape 2 aux nouveaux champs de l'étape 4 ;
6. utiliser `Align` pour réordonner les champs de l'étape 5 dans le même ordre que `B` ;
7. utiliser `LabelledGeneric` pour convertir le résultat de l'étape 6 vers `B`.

Nous avons déja vu comment implémenter les étapes 1, 2, 4, 6 et 7. Nous pouvons implémenter l'étape 3 en utilisant un op appelé `Diff` qui est très similaire à `Intersection`, et nous pouvons implémenter l'étape 5 en utilisant un autre op appélé `Prepend`. Voici la solution complète :

```
implicit def genericMigration[
  A, B, ARepr <: HList, BRepr <: HList,
  Common <: HList, Added <: HList, Unaligned <: HList
](
  implicit
  aGen    : LabelledGeneric.Aux[A, ARepr],
  bGen    : LabelledGeneric.Aux[B, BRepr],
  inter   : hlist.Intersection.Aux[ARepr, BRepr, Common],
  diff    : hlist.Diff.Aux[BRepr, Common, Added],
  monoid  : Monoid[Added],
  prepend : hlist.Prepend.Aux[Added, Common, Unaligned],
  align   : hlist.Align[Unaligned, BRepr]
): Migration[A, B] =
```

[4] Le jeu de mots est intentionel. (note du traducteur: jeu d emot intraduisible)

```
  new Migration[A, B] {
    def apply(a: A): B =
      bGen.from(align(prepend(monoid.empty, inter(aGen.to(a)))))
  }
```

Notez que ce code n'évalue pas toutes les type classes. Nous utilisons `Diff` uniquement pour calculer le type de données `Added`, mais nous n'avons pas besoin d'exécuter `diff.apply`. Au lieu de cela, nous utilisons notre `Monoid` pour invoquer une instance de `Added`.

Avec cette dernière version de la *type class* nous pouvons utiliser `Migration` pour tous les cas d'utilisation que nous avions données au début de cette étude de cas :

```
IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2a]
// res14: IceCreamV2a = IceCreamV2a(Sundae,true)

IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2b]
// res15: IceCreamV2b = IceCreamV2b(Sundae,true,1)

IceCreamV1("Sundae", 1, true).migrateTo[IceCreamV2c]
// res16: IceCreamV2c = IceCreamV2c(Sundae,true,1,0)
```

C'est incroyable tout ce que nous pouvons faire avec les type class ops. `Migration` ne contient qu'une `implicit def` avec une seul ligne d'implémentation au value-level. Cela nous permet d'automatiser les migrations entre *n'importe* quelle paire de *case classes*, avec approximativement la même quantité de code nécessiare pour gérer une *seule* paire de types en utilisant une bibliothèque standard. C'est la grande puissance de shapeless !

6.4 Record ops

Nous avons passé un certain temps dans ce chapitre à travailler avec les type classes du package `shapeless.ops.hlist` et `shapeless.ops.coproduct`. Nous ne devons pas passer à côté du troisième package :`shapeless.ops.record`.

« Record ops » de shapeless fournissent des opérations semblables à celles des `Map` sur les `HLists` d'éléments taggés. Voici quelques exemples mettant

en scène IceCreams :

```
import shapeless._

case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

```
val sundae = LabelledGeneric[IceCream].
  to(IceCream("Sundae", 1, false))
// sundae: shapeless.::[String with shapeless.labelled.KeyTag[Symbol
     with shapeless.tag.Tagged[String("name")],String],shapeless.::[
     Int with shapeless.labelled.KeyTag[Symbol with shapeless.tag.
     Tagged[String("numCherries")],Int],shapeless.::[Boolean with
     shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String
     ("inCone")],Boolean],shapeless.HNil]]] = Sundae :: 1 :: false ::
     HNil
```

Contrairement aux ops de `HList` et de `Coproduct` que nous avons vus précédemment la syntaxe des record ops nécessite un import explicite de `shapeless.record` :

```
import shapeless.record._
```

6.4.1 Sélectionner les champs

La méthode d'extension `get` et sa type classe `Selector` nous permettent d'extraire un champ par son tag :

```
sundae.get('name)
// res1: String = Sundae
```

```
sundae.get('numCherries)
// res2: Int = 1
```

Comme nous nous y attendions, essayer d'accéder à un champ qui n'est pas défini provoque une erreur de compilation :

```
sundae.get('nomCherries)
// <console>:22: error: No field Symbol with shapeless.tag.Tagged[
     String("nomCherries")] in record shapeless.::[String with
     shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String
     ("name")],String],shapeless.::[Int with shapeless.labelled.KeyTag
     [Symbol with shapeless.tag.Tagged[String("numCherries")],Int],
     shapeless.::[Boolean with shapeless.labelled.KeyTag[Symbol with
     shapeless.tag.Tagged[String("inCone")],Boolean],shapeless.HNil]]]
//        sundae.get('nomCherries)
//                   ^
```

6.4.2 Mettre à jour ou enlever des champs

La méthode `updated` ce type class `Updater` nous permet de modifier un champ via sa clé. La méthode `remove` ce type class `Remover` nous permet de supprimer un champ via sa clé :

```
sundae.updated('numCherries, 3)
// res4: shapeless.::[String with shapeless.labelled.KeyTag[Symbol
     with shapeless.tag.Tagged[String("name")],String],shapeless.::[
     Int with shapeless.labelled.KeyTag[Symbol with shapeless.tag.
     Tagged[String("numCherries")],Int],shapeless.::[Boolean with
     shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String
     ("inCone")],Boolean],shapeless.HNil]]] = Sundae :: 3 :: false ::
     HNil
```

```
sundae.remove('inCone)
// res5: (Boolean, shapeless.::[String with shapeless.labelled.KeyTag[
     Symbol with shapeless.tag.Tagged[String("name")],String],
     shapeless.::[Int with shapeless.labelled.KeyTag[Symbol with
     shapeless.tag.Tagged[String("numCherries")],Int],shapeless.HNil
     ]]) = (false,Sundae :: 1 :: HNil)
```

La méthode `updateWith` et sa case class `Modifier` nous permettent de modifier un champ avec une fonction de mise à jour :

```
sundae.updateWith('name)("MASSIVE " + _)
// res6: shapeless.::[String with shapeless.labelled.KeyTag[Symbol
     with shapeless.tag.Tagged[String("name")],String],shapeless.::[
     Int with shapeless.labelled.KeyTag[Symbol with shapeless.tag.
     Tagged[String("numCherries")],Int],shapeless.::[Boolean with
     shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String
     ("inCone")],Boolean],shapeless.HNil]]] = MASSIVE Sundae :: 1 ::
     false :: HNil
```

6.4.3 Convertir en une *Map* conventionnelle

La méthode `toMap` et sa case class `ToMap` nous permettent de convertir un record en `Map` :

```
sundae.toMap
// res7: Map[Symbol with shapeless.tag.Tagged[_ >: String("inCone")
     with String("numCherries") with String("name") <: String],Any] =
     Map('inCone -> false, 'numCherries -> 1, 'name -> Sundae)
```

6.4.4 Les autres opérations

Il existe d'autres ops pour les records mais nous n'avons pas le temps de les aborder ici. Nous pouvons renommer un champ, fusionner deux records, utiliser une fonction map sur leurs valeurs et bien plus. Consultez le code source de `shapeless.ops.record` et `shapeless.syntax.record` pour de plus amples informations.

6.5 Résumé

Dans ce chapitre, nous avons exploré quelques-unes des *types classes* qui sont fournies par le package `shapeless.ops`. Nous nous somme intéréssés à deux exemples simples du pattern ops : `Last` et `Init`, et nous avons construit nos propres type classes `Penultimate` et `Migration` en associant des briques de construction pré-existantes.

De nombreuses type classe ops partagent les mêmes techniques que celui des ops que nous avons vus jusqu'ici. La facon la plus simple de les apprendre et de regarder directement leur code source dans `shapeless.ops` and `shapeless.syntax`.

Dans le chapitre suivant, nous allons aborder deux ensembles de *type classe* ops qui nécessitent davantage de théorie. Le Chapitre 7 traite des opérations fonctionnelles comme `map` et `flatMap` sur les `HLists`, et le Chapitre 8 nous explique comment implémenter des type classes qui nécessitent une représentation au type level des nombres. Ces informations vont nous aider à mieux comprendre l'ensemble des type classes de `shapeless.ops`.

Chapter 7

Opération fonctionnelle sur les *HLists*

Les programmes Scala « classiques » utilisent beaucoup les opérations fonctionnelles telles que `map` et `flatMap`. Une question se pose : peut-on effectuer des opérations semblables sur les HLists‘ ? Les réponse est oui, mais nous devons faire les choses légèrement différemment par rapport au Scala classique. Nous ne serons pas surpris de constater que le mécanisme utilisé reste basé sur les type classes et un ensemble de *type class*es ops pour nous aider.

Avant de nous pencher directement sur les type classes, nous devons aborder la façon dont shapeless représente les *fonctions polymorphiques* qui seront capables de mapper sur une structure de données hétérogènes.

7.1 Motivation : mapper sur une *HList*

Nous allons traiter les fonctions polymorphiques à l'aide de la méthode `.map` La Figure 7.1 montre un diagramme des types d'une fonction map classique sur les listes. Nous commençons avec une `List[A]`, nous fournissons une fonction `A => B`, et nous terminons par `List[B]`.

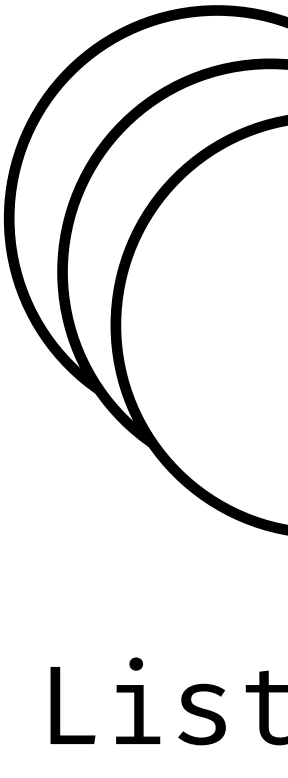


Figure 7.1: Mapper sur une liste classique (“monomorphic” map)

Le fait qu'il y ait des éléments hétérogènes dans une `HList` met ce modèle à mal. Les fonctions Scala ont des types d'entrée et de sortie fixés, le résultat de notre map devra donc avoir le même type pour toutes les positions.

Dans l'idéal, nous voulons que notre fonction `map` corresponde à la Figure 7.2, où la fonction inspecte le type de chaque entrée pour déterminer le type de chaque sortie. Ce qui nous donne un environement clos et composable qui conserve la nature hétérogène des `HList`.

Malheureusement nous ne pouvons pas utiliser les fonctions Scala pour implémenter ces opérations. Nous avons besoin d'une nouvelle infrastructure.

7.2 Fonctions polymorphique

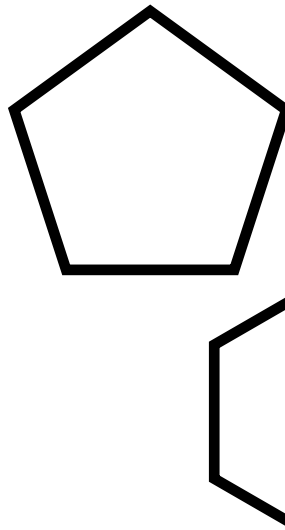
Shapeless fournit un type appelé `Poly` pour représenter les *fonctions polymorphiques* dont le type de retour dépend des paramètres de types. Voici une explication simplifiée de leur fonctionnement. Notez que la section suivante ne contient pas de vrai code shapeless, nous mettons de coté une partie de la flexibilité et des mécanismes qui servent à simplifier l'utilisation de la vraie bibliothèque de shapeless pour créer une version simplifiée dans le cadre de notre exemple.

7.2.1 Comment *Poly* fonctionne

Au cœur de `Poly`, on trouve un objet avec une méthode `apply` générique. En plus de l'habituel paramètre de type `A`, `Poly` prend un paramètre implicite de type `Case[A]`:

```
// Ceci n'est pas du code shapeless.
// Ceci n'est qu'une démonstration.

trait Case[P, A] {
  type Result
  def apply(a: A): Result
}
```



A :: B ::

Figure 7.2: mapper sur des listes hétérogènes (“polymorphic” map)

```
trait Poly {
  def apply[A](arg: A)(implicit cse: Case[this.type, A]): cse.Result =
    cse.apply(arg)
}
```

Lorsque qu'on définit un véritable `Poly`, on fournit une instance de `Case` pour chaque paramètre de type dont nous avons besoin. Ceci permet d'implémenter le vrai corps de la fonction :

```
// Ceci n'est pas du code shapeless.
// Ceci n'est qu'une démonstration.

object myPoly extends Poly {
  implicit def intCase =
    new Case[this.type, Int] {
      type Result = Double
      def apply(num: Int): Double = num / 2.0
    }

  implicit def stringCase =
    new Case[this.type, String] {
      type Result = Int
      def apply(str: String): Int = str.length
    }
}
```

Lorsqu'on appelle `myPoly.apply`, le compilateur recherche le `Case` implicite correspondant et l'intègre comme à l'accoutumée :

```
myPoly.apply(123)
// res8: Double = 61.5
```

Il y a ici quelques subltilités de scoping, qui permettent au compilateur de trouver les instances de `Case` sans import supplémentaire. `Case` a un paramètre de type supplémentaire appelé `P` qui référence le type singleton de `Poly`. Le scope de `Case[P, A]` contient comme implicit une référence à l'objet compagnon de `Case`, `P` et de `A`. Nous avons assigné `myPoly.type` à `P` et l'objet compagnon de `myPoly.type` est `myPoly` lui-même. En d'autre termes, `Cases` qui est défini dans le corps de `Poly` est toujours accesible dans le scope quel que soit l'emplacement de l'appel.

7.2.2 la syntaxe de *Poly*

Le code ci-dessus n'est pas le véritable code de shapeless. Heureusement, shapeless permet de définir les `Polys` de facon bien plus simple. Voici notre fonction `myPoly` réécrite avec la véritable syntaxe :

```
import shapeless._

object myPoly extends Poly1 {
  implicit val intCase: Case.Aux[Int, Double] =
    at(num => num / 2.0)

  implicit val stringCase: Case.Aux[String, Int] =
    at(str => str.length)
}
```

Il y a quelques différences notables par rapport à notre syntaxe précédente :

1. Nous étendons le trait appelé `Poly1` au lieu de `Poly`. Shapeless a un type `Poly` et un ensemble de sous-types, `Poly1` a `Poly22`, qui supporte les différentes variétés de fonctions polymorphiques.

2. Le type `Case.Aux` ne semble pas faire référence au type singleton de `Poly`. `Case.Aux` est en fait un alias de type défini dans le corp de `Poly1`. Le type singleton est bien présent, il est simplement caché.

3. Nous utilisons la méthode helper `at` pour définir nos cas. Elle agit à la façon d'une méthode constructrice d'instance, comme traité dans la section 3.1.2), elle sert principalement à éliminér le boilerplate.

Une fois les différences de syntaxe mises de côté, la version shapeless de `myPoly` a la même fonctionalité que notre version de démonstration. Nous pouvons l'appeler avec un paramètre de type `Int` ou `String` et obtenir en retour le type correspondant :

```
myPoly.apply(123)
// res10: myPoly.intCase.Result = 61.5

myPoly.apply("hello")
// res11: myPoly.stringCase.Result = 5
```

Shapeless supporte aussi les `Polys` avec plus d'un paramètre. En voici un exemple avec deux paramètres :

```
object multiply extends Poly2 {
  implicit val intIntCase: Case.Aux[Int, Int, Int] =
    at((a, b) => a * b)

  implicit val intStrCase: Case.Aux[Int, String, String] =
    at((a, b) => b.toString * a)
}
```

```
multiply(3, 4)
// res12: multiply.intIntCase.Result = 12

multiply(3, "4")
// res13: multiply.intStrCase.Result = 444
```

Les `Cases` n'étant que des valeurs implicites, nous pouvons définir des cases basés sur des type classes et ainsi appliquer toutes les résolutions implicites sophistiquées que nous avions traitées dans le chapitre précédent.

Voici un exemple simple qui donne le total de nombre selon différents contextes :

```
import scala.math.Numeric

object total extends Poly1 {
  implicit def base[A](implicit num: Numeric[A]):
      Case.Aux[A, Double] =
    at(num.toDouble)

  implicit def option[A](implicit num: Numeric[A]):
      Case.Aux[Option[A], Double] =
```

```
    at(opt => opt.map(num.toDouble).getOrElse(0.0))

  implicit def list[A](implicit num: Numeric[A]):
      Case.Aux[List[A], Double] =
    at(list => num.toDouble(list.sum))
}
```

```
total(10)
// res15: Double = 10.0

total(Option(20.0))
// res16: Double = 20.0

total(List(1L, 2L, 3L))
// res17: Double = 6.0
```

Particularité de l'inférence de type

`Poly` amène l'inférence de type de Scala en dehors de sa zone de confort. Nous pouvons facilement embrouiller le compilateur en lui demandant trop d'inférences de type d'un seul coup. Par exemple, le code suivant compile correctement :

```
val a = myPoly.apply(123)
val b: Double = a
```

Par contre, combiner les deux linges précédentes provoque une erreur :

```
val a: Double = myPoly.apply(123)
// <console>:17: error: type mismatch;
//  found   : Int(123)
//  required: myPoly.ProductCase.Aux[shapeless.HNil,?]
//     (which expands to)  shapeless.poly.Case[myPoly.type,
    shapeless.HNil]{type Result = ?}
//        val a: Double = myPoly.apply(123)
//                                     ^
```

Si on ajoute une annotation de type, le code compile à nouveau :

```
val a: Double = myPoly.apply[Int](123)
// a: Double = 61.5
```

Ce comportement est ennuyeux et peut porter à confusion. Malheureusement, il n'existe pas de règles précises pour éviter ces problèmes. La règle générale est d'essayer de ne pas trop contraindre le compilateur, de résoudre une seule contrainte à la fois et de lui donner un coup de pouce quand il se retrouve bloqué.

7.3 Mapping et flatMapping à l'aide de *Poly*

Shapeless fournit un ensemble d'opérations fonctionnelles basées sur `Poly`, chacune implémentée par un type classe ops. Jetons un œil à `map` et `flatMap` en guise d'exemple. voici `map` :

```
import shapeless._

object sizeOf extends Poly1 {
  implicit val intCase: Case.Aux[Int, Int] =
    at(identity)

  implicit val stringCase: Case.Aux[String, Int] =
    at(_.length)

  implicit val booleanCase: Case.Aux[Boolean, Int] =
    at(bool => if(bool) 1 else 0)
}
```

```
(10 :: "hello" :: true :: HNil).map(sizeOf)
// res1: shapeless.::[Int,shapeless.::[Int,shapeless.::[Int,shapeless.
     HNil]]] = 10 :: 5 :: 1 :: HNil
```

Notez que les éléments dans la `HList` résultante a des types qui correspondent au `Case` dans `sizeOf`. Nous pouvons utiliser `map` avec nimporte quel

`Poly` qui contient un `Case` pour chaque membre de notre `HList`. Si le compilateur ne peut pas trouver de `Case` pour un membre, nous obtenons une erreur de compilation :

```
(1.5 :: HNil).map(sizeOf)
// <console>:17: error: could not find implicit value for parameter
     mapper: shapeless.ops.hlist.Mapper[sizeOf.type,shapeless.::[
     Double,shapeless.HNil]]
//        (1.5 :: HNil).map(sizeOf)
//                          ^
```

Nous pouvons aussi utiliser `flatMap` sur une `HList`, tant que tout les cas de notre `Poly` retournent une autre `HList` :

```
object valueAndSizeOf extends Poly1 {
  implicit val intCase: Case.Aux[Int, Int :: Int :: HNil] =
    at(num => num :: num :: HNil)

  implicit val stringCase: Case.Aux[String, String :: Int :: HNil] =
    at(str => str :: str.length :: HNil)

  implicit val booleanCase: Case.Aux[Boolean, Boolean :: Int :: HNil]
    =
    at(bool => bool :: (if(bool) 1 else 0) :: HNil)
}
```

```
(10 :: "hello" :: true :: HNil).flatMap(valueAndSizeOf)
// res3: shapeless.::[Int,shapeless.::[Int,shapeless.::[String,
     shapeless.::[Int,shapeless.::[Boolean,shapeless.::[Int,shapeless.
     HNil]]]]]] = 10 :: 10 :: hello :: 5 :: true :: 1 :: HNil
```

Encore une fois, nous obtenons une erreur de compilation s'il manque des `Cases` ou si un `Case` ne retourne pas de `HList` :

```
// Using the wrong Poly with flatMap:
(10 :: "hello" :: true :: HNil).flatMap(sizeOf)
// <console>:18: error: could not find implicit value for parameter
     mapper: shapeless.ops.hlist.FlatMapper[sizeOf.type,shapeless.::[
     Int,shapeless.::[String,shapeless.::[Boolean,shapeless.HNil]]]]
```

```
//        (10 :: "hello" :: true :: HNil).flatMap(sizeOf)
//                                                ^
```

`map` et `flatMap` sont basés sur les *types classes* `Mapper` et `FlatMapper`. Nous en verrons un exemple d'utilisation de `Mapper` dans la section 7.5.

7.4 Utiliser Fold avec *Poly*

En plus de `map` et `flatMap`, shapeless fournit `foldLeft` et `foldRight` qui sont basés sur `Poly2` :

```
import shapeless._

object sum extends Poly2 {
  implicit val intIntCase: Case.Aux[Int, Int, Int] =
    at((a, b) => a + b)

  implicit val intStringCase: Case.Aux[Int, String, Int] =
    at((a, b) => a + b.length)
}
```

```
(10 :: "hello" :: 100 :: HNil).foldLeft(0)(sum)
// res7: Int = 115
```

Nous pouvons aussi utiliser `reduceLeft`, `reduceRight`, `foldMap` et bien d'autres. Chaque opération est associée à une type classe. Nous vous laissons le soin d'examiner les autres opérations disponibles.

7.5 Définir une type classe utilisant *Poly*

Nous pouvons utiliser `Poly` et les type classes en tant que `Mapper` et un `FlatMapper` comme brique d'assemblage pour nos propres type classes. Comme exemple, construisons une type classe pour transformer une case classe en une autre :

```
trait ProductMapper[A, B, P] {
  def apply(a: A): B
}
```

Nous pouvons créer une instance de `ProductMapper` en utilisant `Mapper` et une paire de `Generics` :

```
import shapeless._
import shapeless.ops.hlist

implicit def genericProductMapper[
  A, B,
  P <: Poly,
  ARepr <: HList,
  BRepr <: HList
](
  implicit
  aGen: Generic.Aux[A, ARepr],
  bGen: Generic.Aux[B, BRepr],
  mapper: hlist.Mapper.Aux[P, ARepr, BRepr]
): ProductMapper[A, B, P] =
  new ProductMapper[A, B, P] {
    def apply(a: A): B =
      bGen.from(mapper.apply(aGen.to(a)))
  }
```

Il est intéressant de noter que nous définissons un type `P` pour notre `Poly`, mais nous ne faisons aucune référence à `P` dans notre code. La *type class* `Mapper` utilise la résolution implicite pour trouver nos `Cases`, le compilateur n'a donc besoin que de connaître le type singleton de `P` pour retrouver les instances.

Créons une méthode d'extension pour simplifier l'utilisation de `ProductMapper`. Nous voulons que l'utilisateur n'ait qu'à spécifier le type de `B` au moment de l'appel, nous utilisons donc une voie détournée pour permettre au compilateur d'inférer le type de `Poly` à partir du type de la valeur en paramètre :

```
implicit class ProductMapperOps[A](a: A) {
  class Builder[B] {
    def apply[P <: Poly](poly: P)
        (implicit pm: ProductMapper[A, B, P]): B =
      pm.apply(a)
  }

  def mapTo[B]: Builder[B] = new Builder[B]
}
```

Voici un exemple de l'utilisation de cette méthode :

```
object conversions extends Poly1 {
  implicit val intCase:  Case.Aux[Int, Boolean]   = at(_ > 0)
  implicit val boolCase: Case.Aux[Boolean, Int]   = at(if(_) 1 else 0)
  implicit val strCase:  Case.Aux[String, String] = at(identity)
}

case class IceCream1(name: String, numCherries: Int, inCone: Boolean)
case class IceCream2(name: String, hasCherries: Boolean, numCones: Int
    )
```

```
IceCream1("Sundae", 1, false).mapTo[IceCream2](conversions)
// res2: IceCream2 = IceCream2(Sundae,true,0)
```

La syntaxe de `mapTo` ressemble à un simple appel de méthode, mais il s'agit en réalité de deux appels : un appel à `mapTo` pour fixer le paramètre de type `B` et un appel à `Builder.apply` pour spécifier le `Poly`. Certaines méthodes d'extension ops de shapeless utilisent la même astuce pour fournir une syntaxe plus simple à l'utilisateur.

7.6 Résumé

Dans ce chapitre, nous avons traité des *fonctions polymorpgiques* dont le type de retour varie en fonction du type de paramètre. Nous avons vu comment le type `Poly` de shapeless est défini et comment implémenter des opérations fonctionnelles comme `map`, `flatMap`, `foldLeft` et `foldRight`.

Chaque opération est implémentée comme une méthode d'extension sur `HList`, basée sur une des type classes correspondantes : `Mapper`, `FlatMapper`, `LeftFolder` et les autres. Nous pouvons utiliser ces type classes, `Poly` et les techniques de la section 4.3 pour créer nos propres type classes comportant un enchaînement de transformations sophistiquées.

Chapter 8

Compter avec les types

De temps en temps nous avons besoin de compter quelque chose au niveau des types. Par exemple, nous pourrions avoir besoin de connaître la taille d'une `HList`. Nous pouvons représenter les nombres facilement avec les valeurs, mais si nous voulons influencer la résolution d'implicite, nous avons besoin de les représenter au niveau des types. Ce chapitre traite des théories qui permettent de compter avec les types, et nous fournirons quelques cas d'utilisation pour la déduction de *type classes*.

8.1 Représenter des nombres par des types.

Shapeless utilise « le codage de Church » pour représenter les nombres naturels au niveau des types. Il introduit le type `Nat` avec deux sous-types : `_0` représente zero et `Succ[N]` représente `N+1` :

```
import shapeless.{Nat, Succ}

type Zero = Nat._0
type One  = Succ[Zero]
type Two  = Succ[One]
// etc...
```

Shapeless fournit des aliases pour les 22 premiers `Nats` via `Nat._N` :

```
Nat._1
Nat._2
Nat._3
// etc...
```

`Nat` n'a pas de sémantique à l'exécution. Nous devons utiliser la *type class* `ToInt` pour convertir un `Nat` en un `Int` à l'exécution :

```
import shapeless.ops.nat.ToInt

val toInt = ToInt[Two]
```

```
toInt.apply()
// res7: Int = 2
```

La méthode `Nat.toInt` constitue un moyen plus pratique d'appeler `toInt.apply()`. Elle prend en paramètre implicite une instance de `ToInt` :

```
Nat.toInt[Nat._3]
// res8: Int = 3
```

8.2 La longueur des représentations génériques

Un des cas d'utilisation de `Nat`consiste à déterminer la taille des `HLists` et des `Coproducts`. Shapeless fournit les type classes `shapeless.ops.hlist.Length` et `shapeless.ops.coproduct.Length` pour cela :

```
import shapeless._
import shapeless.ops.{hlist, coproduct, nat}
```

```
val hlistLength = hlist.Length[String :: Int :: Boolean :: HNil]
// hlistLength: shapeless.ops.hlist.Length[shapeless.::[String,
     shapeless.::[Int,shapeless.::[Boolean,shapeless.HNil]]]]{type Out
      = shapeless.Succ[shapeless.Succ[shapeless.Succ[shapeless._0]]]}
```

```
     = shapeless.ops.hlist$Length$$anon$3@5a8a80b1

val coproductLength = coproduct.Length[Double :+: Char :+: CNil]
// coproductLength: shapeless.ops.coproduct.Length[shapeless.:+:[
     Double,shapeless.:+:[Char,shapeless.CNil]]]{type Out = shapeless.
     Succ[shapeless.Succ[shapeless._0]]} = shapeless.ops.
     coproduct$Length$$anon$29@6dd27140
```

Les instances de `Length` ont un membre de type `Out` qui représente la taille avec un `Nat` :

```
Nat.toInt[hlistLength.Out]
// res0: Int = 3

Nat.toInt[coproductLength.Out]
// res1: Int = 2
```

Utilisons un exemple concret. Nous allons créer une type class `SizeOf` qui compte le nombre de champs d'une case class et retourne cette valeur comme un simple `Int` :

```
trait SizeOf[A] {
  def value: Int
}

def sizeOf[A](implicit size: SizeOf[A]): Int = size.value
```

Pour créer une instance de `SizeOf`, nous avons besoin de trois choses :

1. un `Generic` pour calculer la `HList` correspondant au type ;
2. une `Length` pour calculer la taille de la `HList` pour obtenir un `Nat` ;
3. un `ToInt` pour convertir le `Nat` en `Int`.

Voici une implémentation qui marche, elle est écrite comme décrit dans le Chapitre 4 :

```
implicit def genericSizeOf[A, L <: HList, N <: Nat](
  implicit
  generic: Generic.Aux[A, L],
  size: hlist.Length.Aux[L, N],
  sizeToInt: nat.ToInt[N]
): SizeOf[A] =
  new SizeOf[A] {
    val value = sizeToInt.apply()
  }
```

Nous pouvons tester notre code de la façon suivante :

```
case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

```
sizeOf[IceCream]
// res3: Int = 3
```

8.3 Étude de cas: générateur de valeur aléatoire

Les bibliothèques de test axées sur les propriétés comme ScalaCheck[1] utilisent des type classes dont le but est de générer des valeurs aléatoires pour les test unitaires. Par exemple, ScalaCheck fournit la *type class* `Arbitrary` que nous pouvons utiliser de la façon suivante :

```
import org.scalacheck._
```

```
for(i <- 1 to 3) println(Arbitrary.arbitrary[Int].sample)
// Some(747857793)
// Some(2147483647)
// Some(-578177219)

for(i <- 1 to 3) println(Arbitrary.arbitrary[(Boolean, Byte)].sample)
// Some((false,118))
// Some((true,58))
```

[1] https://scalacheck.org

```
// Some((false,-128))
```

ScalaCheck founit deja des instances d'`Arbitrary` pour un grand nombre de types scala standards. Pourtant, créer des instances d'`Arbitrary` pour les utilisateurs d'*ADT*s reste un travail manuel et chronophage. C'est ce qui rend très intéressant l'intégration de shapeless via des bibliothèques comme scalacheck-shapeless[2].

Dans cette section, nous allons créer une simple type class `Random` pour générer des valeurs aléatoires pour un *ADT* donné. Nous allons voir comment `Length` et `Nat` font partie intégrante de l'implémentation. Comme toujours, nous commençons par définir la *type class* elle-même.

```
trait Random[A] {
  def get: A
}

def random[A](implicit r: Random[A]): A = r.get
```

8.3.1 De simples valeurs aléatoires

Commencons avec une instance basique de `Random` :

```
// Instance constructor:
def createRandom[A](func: () => A): Random[A] =
  new Random[A] {
    def get = func()
  }

// Random numbers from 0 to 9:
implicit val intRandom: Random[Int] =
  createRandom(() => scala.util.Random.nextInt(10))

// Random characters from A to Z:
implicit val charRandom: Random[Char] =
  createRandom(() => ('A'.toInt + scala.util.Random.nextInt(26)).
     toChar)
```

[2] https://github.com/alexarchambault/scalacheck-shapeless

```
// Random booleans:
implicit val booleanRandom: Random[Boolean] =
  createRandom(() => scala.util.Random.nextBoolean)
```

Nous pouvons utiliser ces générateurs simples via la methode `random` :

```
for(i <- 1 to 3) println(random[Int])
// 0
// 8
// 9

for(i <- 1 to 3) println(random[Char])
// V
// N
// J
```

8.3.2 Produits aléatoires

Nous pouvons créer des valeurs aléatoires pour les produits en utilisant `Generic` et `HList` avec les techniques vues dans le Chapitre 3 :

```
import shapeless._

implicit def genericRandom[A, R](
  implicit
  gen: Generic.Aux[A, R],
  random: Lazy[Random[R]]
): Random[A] =
  createRandom(() => gen.from(random.value.get))

implicit val hnilRandom: Random[HNil] =
  createRandom(() => HNil)

implicit def hlistRandom[H, T <: HList](
  implicit
  hRandom: Lazy[Random[H]],
  tRandom: Random[T]
): Random[H :: T] =
  createRandom(() => hRandom.value.get :: tRandom.get)
```

Ce qui nous conduit à invoquer des instances aléatoires pour les *case classes* :

```
case class Cell(col: Char, row: Int)
```

```
for(i <- 1 to 5) println(random[Cell])
// Cell(H,1)
// Cell(D,4)
// Cell(D,7)
// Cell(V,2)
// Cell(R,4)
```

8.3.3 Coproduits aléatoires

C'est ici que nous commençons à rencontrer des problèmes. Générer une valeur aléatoire d'un coproduit implique de choisir aléatoirement un sous-type. Commençons avec une implémentation naïve :

```
implicit val cnilRandom: Random[CNil] =
  createRandom(() => throw new Exception("Inconceivable!"))

implicit def coproductRandom[H, T <: Coproduct](
  implicit
  hRandom: Lazy[Random[H]],
  tRandom: Random[T]
): Random[H :+: T] =
  createRandom { () =>
    val chooseH = scala.util.Random.nextDouble < 0.5
    if(chooseH) Inl(hRandom.value.get) else Inr(tRandom.get)
  }
```

Le problème de cette implémentation est qu'elle repose sur un choix binaire dans le calcul de `chooseH`. Ce qui provoque des inégalités. Par exemple, étudiez le type suivant :

```
sealed trait Light
case object Red extends Light
case object Amber extends Light
```

```
case object Green extends Light
```

Le `Repr` de `Light` est `Red :+: Amber :+: Green :+: CNil`. Pour ce type, une instance de `Random` choisira `Red` une fois sur deux et `Amber :+: Green :+: CNil` une fois sur deux. Alors qu'une distribution correcte serait à 33 % `Red` et 67 % `Amber :+: Green :+: CNil`.

Et ce n'est pas tout. Si nous regardons la distribution de probabilité globale nous constatons quelque-chose d'alarmant :

- `Red` est choisi une fois sur deux
- `Amber` est choisi une fois sur quatre
- `Green` est choisi une fois sur huit
- *`CNil` est choisi une fois sur seize*

Notre instance de coproduit lèvera une exception 6,75 % du temps !

```
for(i <- 1 to 100) random[Light]
// java.lang.Exception: Inconceivable!
//   ...
```

Pour résoudre ce problème, nous devons modifier la probabilité de choisir `H` par rapport a `T` Il faudrait choisir `H` `1/n` du temps, où n est la taille du coproduit, ce qui assurerait une distribution des probabilités égale entre les sous-types du coproduit. Cela assure aussi de choisir toujours la tête d'un coproduit comportant un seul sous-type, ce qui assure que `cnilProduct.get` n'est jamais choisi. Voici notre implémentation mise à jour :

```
import shapeless.ops.coproduct
import shapeless.ops.nat.ToInt

implicit def coproductRandom[H, T <: Coproduct, L <: Nat](
  implicit
  hRandom: Lazy[Random[H]],
  tRandom: Random[T],
  tLength: coproduct.Length.Aux[T, L],
  tLengthAsInt: ToInt[L]
): Random[H :+: T] = {
```

```
  createRandom { () =>
    val length = 1 + tLengthAsInt()
    val chooseH = scala.util.Random.nextDouble < (1.0 / length)
    if(chooseH) Inl(hRandom.value.get) else Inr(tRandom.get)
  }
}
```

Avec ces modifications, nous pouvons générer une valeur aléatoire pour n'importe quel coproduit :

```
for(i <- 1 to 5) println(random[Light])
// Green
// Red
// Red
// Red
// Green
```

Générer des données de test pour ScalaCheck nécessite normalement beaucoup de boilerplate. La génération de valeurs aléatoires est un exemple fascinant de l'utilisation de shapeless où les `Nat` ont une place essentielle.

8.4 Les autres opérations impliquant *Nat*

Shapeless fournit tout un ensemble d'autres opérationss basées sur `Nat`. La méthode `apply` sur les `HList` ou les `Coproduct` peut accepter un `Nats` comme paramètre ou comme paramètre de type :

```
import shapeless._

val hlist = 123 :: "foo" :: true :: 'x' :: HNil
```

```
hlist.apply[Nat._1]
// res1: String = foo
```

```
hlist.apply(Nat._3)
// res2: Char = x
```

Il y a aussi des opérations comme `take`, `drop`, `slice` et `updatedAt` :

```
hlist.take(Nat._3).drop(Nat._1)
// res3: shapeless.::[String,shapeless.::[Boolean,shapeless.HNil]] =
     foo :: true :: HNil

hlist.updatedAt(Nat._1, "bar").updatedAt(Nat._2, "baz")
// res4: shapeless.::[Int,shapeless.::[String,shapeless.::[String,
     shapeless.::[Char,shapeless.HNil]]]] = 123 :: bar :: baz :: x ::
     HNil
```

Ces opérations et leurs type classes associées sont utiles pour manipuler individuellement les éléments d'un produit ou d'un coproduit.

8.5 Résumé

Dans ce chapitre, nous avons traité la façon dont shapeless représente les nombre naturels et comment nous pouvons les utiliser dans les type classes. Nous avons vu qu'il existe déjà des type classe ops qui nous permettent de faire des choses comme calculer des tailles et accéder à des éléments par leurs index. Nous avons aussi vu comment créer nos type classes qui utilisent `Nat` pour leurs propres besoins.

Avec `Nat`, `Poly` et les variétés de type que nous avons vues dans le dernier chapitre, nous n'avons abordé qu'une fraction des outils fournit par `shapeless.ops`. Il existe de nombreux autres type classes qui apportent des fondations structurées sur lesquelles nous pouvons construire notre propre code.

Cependant, la théorie décrite ici est suffisante pour comprendre la majorité des opérations nécessaires à la déduction de nos propre type classes. La code source du package `shapeless.ops` devrait vous être désormais suffisament compréhensible pour que vous y séléctionniez d'autre ops qui vous seront utiles.

Préparez-vous au lancement!

Avec la partie II et son traitement de `shapeless.ops` nous touchons à la fin de ce guide. Nous espérons qu'il vous a aidé à comprendre cette librairie captivante et plaine de ressources, et nous vous souhaitons un excellent futur voyage en tant qu'astronaute des types.

En tant que développeur fonctionnel, nous plaçons l'abstraction au-dessus de tout. Les concepts comme les functors et les monads sont apparues suite à des années de recherche : écrire du code, repérer les patterns, et créer des abstractions pour supprimer les redondances. Shapeless augmente le niveau d'abstraction dans Scala. Les outils comme `Generic` et `LabelledGeneric` fournissent une interface pour abstraire les types de données auparavant frustrants de part leur propriété unique.

Les nouveaux utilisateurs shapeless rencontrent traditionnellement deux obstacles. Le premier est le baggage théorique et la connaissance des détails d'implémentation requis pour comprendre les patterns utilisés. Nous espérons que ce guide vous aidera sur ce point.

La seconde barrière est la peur et l'incertitude qui entoure cette librairie considérée comme « academique » ou « complexe. » On peut y remédier via le partage de savoir, par exemple les cas d'utilisations, les pour et les contre, les stratégies d'implémentation… le tout pour elargir la compréhension de ce précieux outil. Alors s'il-vous-plaît partagez ce livre avec vos amis… et mettons fin au boiler plate ensemble !