Generics in protocol/extension

Hikaru Yoshimura（吉村 優）

Recruit Markting Partners Co., Ltd.
yyu@mental.poker

try! Swift Tokyo on March 21, 2019

(y-yu/try-swift-slide@03e5f55)

Table of Contents

Self-introduction

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

Self-introduction

Twitter @_yyu_
Qiita yyu
GitHub y-yu

- University of Tsukuba (Bachelor of CS)
  - Programming Logic Group
- Scala engineer at Recruit Marketing Partners Co., Ltd.
- I'm interested in
  - Cryptography
  - Functional Programming
  - Quantum Information
  - Typesetting (LaTeX)

Problem

- First we are thinking about these data structures: `Tuple` and `List`

```
struct Tuple<A, B> {
  let left: A
  let right: B

  init(_ l: A, _ r: B) {
    this.left = l
    this.right = r
  }
}
```

```
enum List<A> {
  indirect case Cons(
    h: A,
    t: List<A>
  )
  case Nil
}
```

Problem

- First we are thinking about these data structures: `Tuple` and `List`

```
struct Tuple<A, B> {
  let left: A
  let right: B

  init(_ l: A, _ r: B) {
    this.left = l
    this.right = r
  }
}
```

```
enum List<A> {
  indirect case Cons(
    h: A,
    t: List<A>
  )
  case Nil
}
```

- `Tuple` represents *exact two* values for each type `A` and `B`

Problem

- First we are thinking about these data structures: `Tuple` and `List`

```
struct Tuple<A, B> {
  let left: A
  let right: B

  init(_ l: A, _ r: B) {
    this.left = l
    this.right = r
  }
}
```

```
enum List<A> {
  indirect case Cons(
    h: A,
    t: List<A>
  )
  case Nil
}
```

- `Tuple` represents *exact two* values for each type `A` and `B`
- And `List` represents fixed type `A` values whose number is *more than or equal 0*

Problem

- First we are thinking about these data structures: `Tuple` and `List`

```
struct Tuple<A, B> {
  let left: A
  let right: B

  init(_ l: A, _ r: B) {
    this.left = l
    this.right = r
  }
}
```

```
enum List<A> {
  indirect case Cons(
    h: A,
    t: List<A>
  )
  case Nil
}
```

- `Tuple` represents *exact two* values for each type `A` and `B`
- And `List` represents fixed type `A` values whose number is *more than or equal 0*

Can we make a List which have some type values?

Heterogeneous List

- We are looking at `HList` as follows

```
protocol HList {
  associatedtype Head
  associatedtype Tail: HList
}
```

Heterogeneous List

- We are looking at `HList` as follows

```
protocol HList {
  associatedtype Head
  associatedtype Tail: HList
}
```

- `HList` has two `associatedtype`: `Head` and `Tail`

Heterogeneous List

- And there are two structs: `HCons` and `HNil`

```
public enum Nothing { }
struct HNil: HList {
  typealias Head = Nothing
  typealias Tail = HNil

  init() { }

  static var hNil = HNil()
}
```

```
struct HCons<H, T: HList>: HList {
  typealias Head = H
  typealias Tail = T
  let head: Head
  let tail: Tail

  init(_ h: Head, _ t: Tail) {
    self.head = h
    self.tail = t
  }
}
```

Heterogeneous List

- And there are two structs: `HCons` and `HNil`

```
public enum Nothing { }
struct HNil: HList {
  typealias Head = Nothing
  typealias Tail = HNil

  init() { }

  static var hNil = HNil()
}
```

```
struct HCons<H, T: HList>: HList {
  typealias Head = H
  typealias Tail = T
  let head: Head
  let tail: Tail

  init(_ h: Head, _ t: Tail) {
    self.head = h
    self.tail = t
  }
}
```

Is it the same to `List`?

Heterogeneous List

- And there are two structs: `HCons` and `HNil`

```
public enum Nothing { }
struct HNil: HList {
  typealias Head = Nothing
  typealias Tail = HNil

  init() { }

  static var hNil = HNil()
}
```

```
struct HCons<H, T: HList>: HList {
  typealias Head = H
  typealias Tail = T
  let head: Head
  let tail: Tail

  init(_ h: Head, _ t: Tail) {
    self.head = h
    self.tail = t
  }
}
```

Is it the same to `List`?

No, it's not just `List`

Heterogeneous List

- `HList` can have some values and their type order

Heterogeneous List

- `HList` can have some values and their type order

```
let a = HCons(
  1,
  HCons(
    true,
    HCons(
      "string",
      HNil.hnil()
    )
  )
)
```

Heterogeneous List

- `HList` can have some values and their type order

```
let a = HCons(
  1,
  HCons(
    true,
    HCons(
      "string",
      HNil.hnil()
    )
  )
)
```

```
a: HCons<Int, HCons<Bool, HCons<String, HNil>>>
```

Heterogeneous List

- `HList` can have some values and their type order

```
let a = HCons(
  1,
  HCons(
    true,
    HCons(
      "string",
      HNil.hnil()
    )
  )
)
```

```
a: HCons<Int, HCons<Bool, HCons<String, HNil>>>
```

- `List` cannot have more than 1 type, but `HList` have any of the number of types

Heterogeneous List

- `HList` can have some values and their type order

```
let a = HCons(
  1,
  HCons(
    true,
    HCons(
      "string",
      HNil.hnil()
    )
  )
)
```

```
a: HCons<Int, HCons<Bool, HCons<String, HNil>>>
```

- `List` cannot have more than 1 type, but `HList` have any of the number of types

Today, we'll consider the `append` method of two `HLists`

append for Lists

- For now, in the case of the ordinary `List`

append for Lists

- For now, in the case of the ordinary `List`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

append for Lists

- For now, in the case of the ordinary `List`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

- But, `HList` may have more than 1 type, so it's not easy... 😇

append for Lists

- For now, in the case of the ordinary `List`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

- But, `HList` may have more than 1 type, so it's not easy... 😇

What should we use to implement it?

append for Lists

- For now, in the case of the ordinary `List`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

- But, `HList` may have more than 1 type, so it's not easy... 😇

What should we use to implement it?

It's difficult for Java but Swift has `protocol` and `extension`

protocol/extension

protocol/extension

- Swift `protocol` sometimes could be said “it’s similler to `interface` of Java”

protocol/extension

- Swift `protocol` sometimes could be said “it’s similler to `interface` of Java”

```
interface JsonWrite<A> {
  public JSON toJson(A a);
}
class User(???) implements JsonWrite<User> {
  public JSON toJson(User a) {
    ???
  }
}
```

Listing: Java interface

```
protocol JsonWrite {
  associatedtype A
  func toJson(_ a: A) -> JSON
}
class User: JsonWrite {
  typealias A = User
  func toJson(_ a: A) -> JSON {
    ???
  }
}
```

Listing: Swift protocol

protocol/extension

- Swift `protocol` sometimes could be said “it’s similler to `interface` of Java”

```
interface JsonWrite<A> {
  public JSON toJson(A a);
}
class User(???) implements JsonWrite<User> {
  public JSON toJson(User a) {
    ???
  }
}
```

Listing: Java interface

```
protocol JsonWrite {
  associatedtype A
  func toJson(_ a: A) -> JSON
}
class User: JsonWrite {
  typealias A = User
  func toJson(_ a: A) -> JSON {
    ???
  }
}
```

Listing: Swift protocol

It’s not at all...

protocol/extension

- If we need to extend a fixed data type such a `Integer`

protocol/extension

- If we need to extend a fixed data type such a `Integer`
- However in Java, we cannot extend a fixed data type so we can't help but to use the *Adapter pattern*

```
class RichInteger implements JsonWrite<Integer> {
  public Integer n;
  public RichInteger(Integer i) {
    this.n = i
  }

  public JSON toJson(Integer a) {
    ???
  }
}
```

Listing: Adapter pattern in Java

protocol/extension

- If we need to extend a fixed data type such a `Integer`
- However in Java, we cannot extend a fixed data type so we can't help but to use the *Adapter pattern*

```
class RichInteger implements JsonWrite<Integer> {
  public Integer n;
  public RichInteger(Integer i) {
    this.n = i
  }

  public JSON toJson(Integer a) {
    ???
  }
}
```

Listing: Adapter pattern in Java

```
extension Int: JsonWrite {
  typealias A = Int

  func toJson(_ a: Int) -> JSON {
    ???
  }
}
```

Listing: `extension` in Swift

- But Swift has `extension`, which find the function by the given types

protocol/extension

What is the meaning of finding functions by the types?

protocol/extension

What is the meaning of finding functions by the types?

- For example, let's think about *overloading* like following

```
1 + 1 // 2
```

```
"hot" + "dog" // "hotdog"
```

protocol/extension

What is the meaning of finding functions by the types?

- For example, let's think about *overloading* like following

```
1 + 1 // 2
```

```
"hot" + "dog" // "hotdog"
```

- Operator + has two behaviors:
  - `(Int, Int) -> Int`
  - `(String, String) -> String`

protocol/extension

What is the meaning of finding functions by the types?

- For example, let's think about *overloading* like following

```
1 + 1 // 2
```

```
"hot" + "dog" // "hotdog"
```

- Operator + has two behaviors:
  - `(Int, Int) -> Int`
  - `(String, String) -> String`
- So Swift looks up a suitable function by the arguments' types

protocol/extension

What is the meaning of finding functions by the types?

- For example, let's think about *overloading* like following

```
1 + 1 // 2
```

```
"hot" + "dog" // "hotdog"
```

- Operator + has two behaviors:
  - `(Int, Int) -> Int`
  - `(String, String) -> String`
- So Swift looks up a suitable function by the arguments' types
- `extension` is simillar like overloading

protocol/extension

- We can write + function using `protocol` and `extension`

```
protocol Plusable {
  associatedtype A
  func +(_ a: A, _ b: A) -> A
}
```

```
extension Int: Plusable {
  typealias A = Int
  func +(_ a: A, _ b: A) -> A {
    // `Int.plus` maybe not exist...
    return Int.plus(a, b)
  }
}
```

```
extension String: Plusable {
  typealias A = String
  func +(_ a: A, _ b: A) -> A {
    return a.stringByAppendingString(b)
  }
}
```

protocol/extension

- We can write + function using `protocol` and `extension`

```
protocol Plusable {
  associatedtype A
  func +(_ a: A, _ b: A) -> A
}
```

```
extension Int: Plusable {
  typealias A = Int
  func +(_ a: A, _ b: A) -> A {
    // `Int.plus` maybe not exist...
    return Int.plus(a, b)
  }
}
```

```
extension String: Plusable {
  typealias A = String
  func +(_ a: A, _ b: A) -> A {
    return a.stringByAppendingString(b)
  }
}
```

- Overloading can be done by `protocol` and `extension`

protocol/extension

- `extension` can use another `extension` defined by a `protocol`

```
extension Array: JsonWrite where Element: JsonWrite {
  typealias A = Array<Element>

  func toJson(_ a: Array<Element>): JSON {
    ???
  }
}
```

protocol/extension

- `extension` can use another `extension` defined by a `protocol`

```
extension Array: JsonWrite where Element: JsonWrite {
  typealias A = Array<Element>

  func toJson(_ a: Array<Element>): JSON {
    ???
  }
}
```

Can we make the `append` for `HLists` by `extension`?

Implementation

- This is a append for two normal `Lists`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

Implementation

- This is a `append` for two normal `Lists`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

- This `append` has two cases depending on the first argument
  - If it is `Cons(h, t)`, then the `append` calls itself passing `t` as the first argument recursively
  - If it is `Nil`, then the function returns the second argument

Implementation

- This is a `append` for two normal `Lists`

```
func append<A>(_ a: List<A>, _ b: List<A>) -> List<A> {
  switch a {
  case let .Cons(h, t):
    return .Cons(h, append(t, b))
  case .Nil:
    return b
  }
}
```

- This `append` has two cases depending on the first argument
  - If it is `Cons(h, t)`, then the `append` calls itself passing `t` as the first argument recursively
  - If it is `Nil`, then the function returns the second argument
- This `append` is using *run time* dispatching, but we want to dispatch in *compile time*

Implementation

- We are making a `protocol` for the `HLists` `append` as follows

```
protocol HAppend {
  associatedtype Left: HList
  associatedtype Right: HList
  associatedtype Result: HList

  static func append(_ l1: Left, _ l2: Right) -> Result
}
```

Implementation

- We are making a `protocol` for the `HLists` `append` as follows

```
protocol HAppend {
  associatedtype Left: HList
  associatedtype Right: HList
  associatedtype Result: HList

  static func append(_ l1: Left, _ l2: Right) -> Result
}
```

- Then we'll think about its `extension`

Implementation

- First we can create a pettern for that the first argument is `HNil`

```
class App1<A: HList> {
  init() {}
}

extension App1: HAppend {
  typealias Left = HNil
  typealias Right = A
  typealias Result = A

  static func append(_ l1: HNil, _ l2: A) -> A {
    return l2
  }
}
```

Implementation

- First we can create a pettern for that the first argument is `HNil`

```
class App1<A: HList> {
  init() {}
}

extension App1: HAppend {
  typealias Left = HNil
  typealias Right = A
  typealias Result = A

  static func append(_ l1: HNil, _ l2: A) -> A {
    return l2
  }
}
```

- It just returns the second argument

Implementation

- Next we have to make a `HCons` pattern

```
extension App1: HAppend where A == HCons<?, ?> {
  ??? // There is non-working......
}
```

Implementation

- Next we have to make a `HCons` pattern

```
extension App1: HAppend where A == HCons<?, ?> {
  ??? // There is non-working......
}
```

- How do we get the ? types? so it doesn't work well... 😇

Implementation

- Next we have to make a `HCons` pattern

```
extension App1: HAppend where A == HCons<?, ?> {
  ??? // There is non-working......
}
```

- How do we get the ? types? so it doesn't work well... 😇
- There are two problems
  1. A structure (`class`, `struct` and `protcol`) can be conformed by the same `protocol` *at most one*
  2. And how do we get the `append` function that can concat previous `HList`s?

Solution for the former

- We make an adapter class App3 following below

```
class App3<A: HList, B: HList, C> {
  init() {}
}
```

```
extension App3: HAppend where A == HCons<C, B> {
  ???
}
```

- We can use App3 for HCons case instead of App1

Solution for the former

- We make an adapter class App3 following below

```
class App3<A: HList, B: HList, C> {
  init() {}
}
```

```
extension App3: HAppend where A == HCons<C, B> {
  ???
}
```

- We can use App3 for HCons case instead of App1
- Then we have to decide the associatedtypes of HAppend

```
extension App3: HAppend where A == HCons<C, B> {
  typealias Left = A
  typealias Right = ? // What should we put?
  typealias Result = ? // What should we put?
}
```

Solution for the former

- We make an adapter class App3 following below

```
class App3<A: HList, B: HList, C> {
  init() {}
}
```

```
extension App3: HAppend where A == HCons<C, B> {
  ???
}
```

- We can use App3 for HCons case instead of App1
- Then we have to decide the associatedtypes of HAppend

```
extension App3: HAppend where A == HCons<C, B> {
  typealias Left = A
  typealias Right = ? // What should we put?
  typealias Result = ? // What should we put?
}
```

What should we put into types: Right and Result?

Solution for the latter

- We add a type parameter called EV to get the previous append

Solution for the latter

- We add a type parameter called EV to get the previous append
- For this, we add 2 type parameters then the adapter class takes 5 parameters rather than 3

```
class App5<A: HList, B: HList, C: HList, D, EV: HAppend> {
  init() {}
}
```

Solution for the latter

- We add a type parameter called EV to get the previous append
- For this, we add 2 type parameters then the adapter class takes 5 parameters rather than 3

```
class App5<A: HList, B: HList, C: HList, D, EV: HAppend> {
  init() {}
}
```

- Finally we can substitute the types for associatedtypes declared by HAppend

```
extension App5: HAppend where A == EV.Left, B == EV.Right, C == EV.Result {
  typealias Left = HCons<D, A>
  typealias Right = B
  typealias Result = HCons<D, C>
}
```

Solution for the latter

```
extension App5: HAppend where A == EV.Left, B == EV.Right, C == EV.Result {
  typealias Left = HCons<D, A>
  typealias Right = B
  typealias Result = HCons<D, C>
}
```

- It can be figured out by the picture

Solution for the latter

```
extension App5: HAppend where A == EV.Left, B == EV.Right, C == EV.Result {
  typealias Left = HCons<D, A>
  typealias Right = B
  typealias Result = HCons<D, C>
}
```

- It can be figured out by the picture

$$\mathsf{A} := \boxed{?_1 \mid ?_2}, \ \mathsf{B} := \boxed{?_3 \mid ?_4}, \ \mathsf{C} := \mathsf{A} + \mathsf{B} = \boxed{?_1 \mid ?_2 \mid ?_3 \mid ?_4}$$

Solution for the latter

```
extension App5: HAppend where A == EV.Left, B == EV.Right, C == EV.Result {
  typealias Left = HCons<D, A>
  typealias Right = B
  typealias Result = HCons<D, C>
}
```

- It can be figured out by the picture

$$\mathsf{A} := \boxed{?_1 \mid ?_2},\ \mathsf{B} := \boxed{?_3 \mid ?_4},\ \mathsf{C} := \mathsf{A} + \mathsf{B} = \boxed{?_1 \mid ?_2 \mid ?_3 \mid ?_4}$$

$$\begin{aligned} \text{Left} &:= \mathit{HCons}(\mathbf{D}, A) = \boxed{\mathbf{D} \mid ?_1 \mid ?_2} \\ \text{Right} &:= \mathsf{B} = \boxed{?_3 \mid ?_4} \\ \text{Result} &:= \mathit{HCons}(\mathbf{D}, \mathsf{A}) + \mathsf{B} = \boxed{\mathbf{D} \mid ?_1 \mid ?_2 \mid ?_3 \mid ?_4} = \mathit{HCons}(D, C) \end{aligned}$$

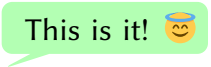
Complete implementation

- We get the perfect implementation for `HLists` append function

```
extension App1: HAppend {
  typealias Left = HNil
  typealias Right = A
  typealias Result = A

  static func append(_ l1: HNil, _ l2: A) -> A {
    return l2
  }
}
extension App5: HAppend where EV.Left == A, B == EV.Right, C == EV.Result {
  typealias Left = HCons<D, A>
  typealias Right = B
  typealias Result = HCons<D, C>

  static func append(_ l1: HCons<D, A>, _ l2: B) -> HCons<D, C> {
    return HCons(l1.head, EV.append(l1.tail, l2))
  }
}
```

Example

```
let a = HCons(1, HCons("string", HNil.hNil))
let b = HCons(true, HCons(1.001, HNil.hNil))

typealias AA = HCons<String, HNil>
typealias A = HCons<Int, AA>
typealias BB = HCons<Double, HNil>
typealias B = HCons<Bool, BB>
typealias CC = HCons<String, B>
typealias C = HCons<Int, CC>
typealias AppC = App1<B>
typealias AppB = App5<HNil, B, B, String, AppC>
typealias AppA = App5<AA, B, CC, Int, AppB>

let appended = AppA.append(a, b)
```



Example

- We have to give Swift what the type EV is because Swift doesn't infer it

Conclusion

Conclusion

- There are some interesting data types like `HList`

Conclusion

- There are some interesting data types like `HList`
- `append` method is able to make by `protocol` and `extension`

Conclusion

- There are some interesting data types like `HList`
- `append` method is able to make by `protocol` and `extension`
- Swift's type inference is wanted by people to improve itself
  - `HList`s are very useful but we'll not use their `append` in the production

References

[1] Benjamin C. Pierce.
Types and Programming Languages.
The MIT Press, 1st edition, 2002.

[2] Benjamin C. Pierce, 英二郎住井, 侑介遠藤, 政裕酒井, 敬吾今井, 裕介黒木, 宜洋今井, 隆文才川, 健男今井.
型システム入門 : プログラミング言語と型の理論.
オーム社, 2013.

Thank you for your attention!