

# Haskell로 타입 안전한 GraphQL과 gRPC 서버 만들기

김은민

Haskell로 타입 안전한 GraphQL과 gRPC  
서버 만들기



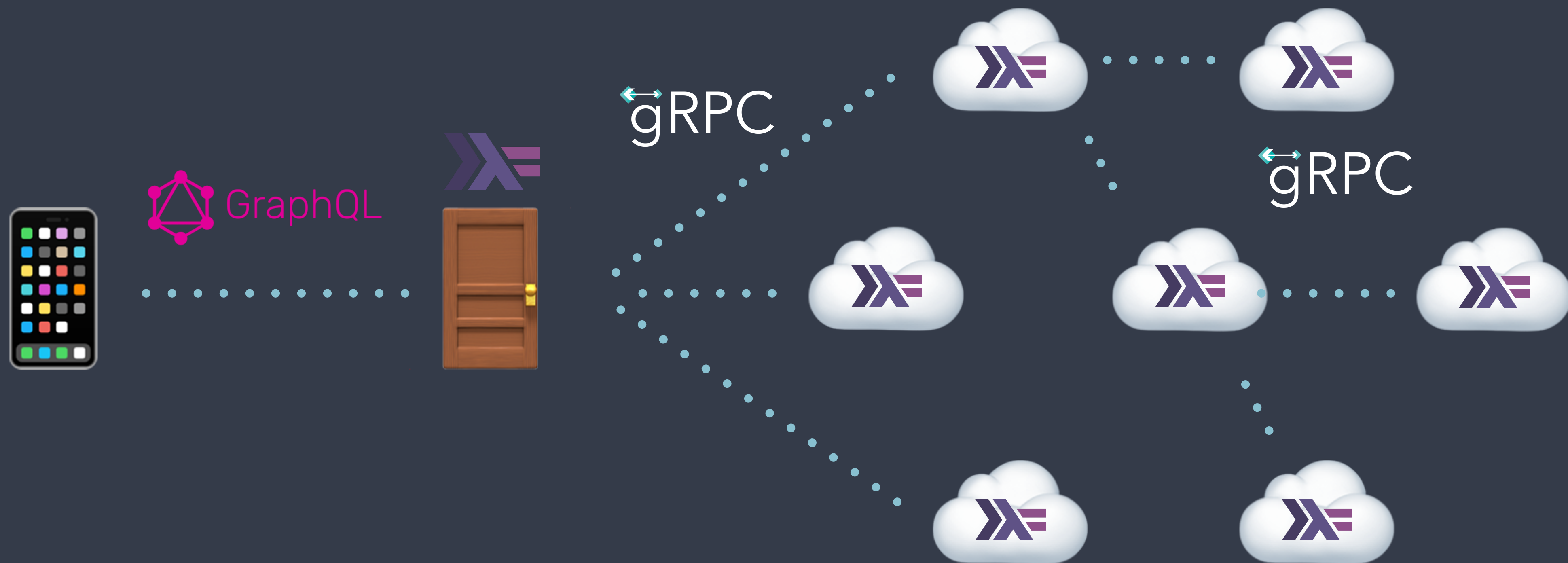
# Haskell Type-level 맛보기

김은민

# 어떤 이야기를 할 것인가요?

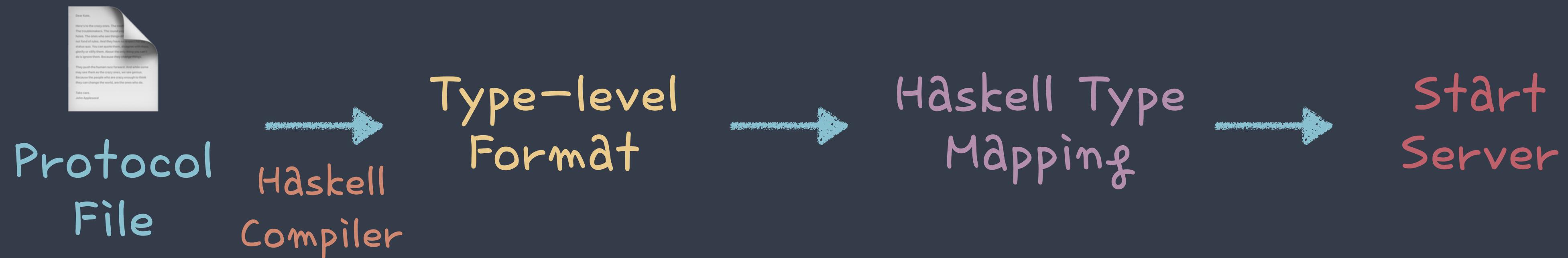
- Haskell로 처음 서비스를 만들면서 배운 지식을 널리 널리 공유
  - 하지만 제한 시간 30분에 지식 공유가 아닌 자신 혼란을 가져올 수도 ... 🌀
- Haskell이 최고다! 🙄 Type level이 최고다! 🙄 GraphQL과 gRPC가 최고다! 🙄
- 🍏 사실 Haskell로 GraphQL과 gRPC 서버를 만드는 내용은 아닙니다. 🙄
- Haskell **Type-level** 프로그래밍 맛보기입니다.
  - 타입 안정성
  - Meta data

# Constacts System 너무 개략도



# Mu-Haskell

- 마이크로서비스를 위한 순수 함수형 프레임워크
- Mu-Scala
- GraphQL, gRPC, gRPC Client, OpenAPI, REST를 지원



# Mu-Haskell gRPC 예제

- schema.proto

```
service Service {  
  rpc SayHello (HelloRequest) returns (HelloResponse) {}  
}
```

```
message HelloRequest { string name = 1; }  
message HelloResponse { string message = 1; }
```



-- 1. GraphQL 파일 로드

```
grpc "QuickstartSchema" (const "QuickstartService") "quickstart.proto" ← Haskell Template
```

-- 2. Message와 하스켈 타입 매핑

```
newtype HelloRequest
```

```
  = HelloRequest { name :: T.Text }
```

```
  deriving (Generic
```

```
    , ToSchema QuickstartSchema "HelloRequest" ← Haskell Type Mapping
```

```
    , FromSchema QuickstartSchema "HelloRequest")
```

```
newtype HelloResponse
```

```
  = HelloResponse { message :: T.Text }
```

```
  deriving (Generic
```

```
    , ToSchema QuickstartSchema "HelloResponse"
```

```
    , FromSchema QuickstartSchema "HelloResponse")
```

```
sayHello :: (MonadServer m) => HelloRequest → m HelloResponse
```

```
sayHello (HelloRequest nm) = pure $ HelloResponse ("hi, " < nm)
```

-- 3. Service와 하스켈 함수 매핑

```
quickstartServer :: (MonadServer m) => SingleServerT QuickstartService m _
```

```
quickstartServer = singleService (method @"SayHello" sayHello) ← Haskell Type Mapping
```

```
main :: IO ()
```

```
main = runGRpcApp msgProtoBuf 8080 quickstartServer ← Start Server
```

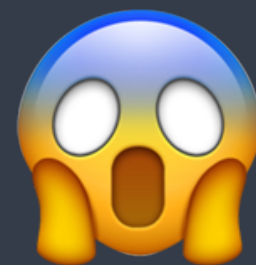


# 그럼 Mu-Haskell이 만들어 주는 중간 Format Type은 어떨까요?

- grpc, graphql Haskell Template 함수는 다음과 같은 타입을 생성합니다

```
type QuickstartSchema
= '[ 'DRecord "HelloRequest" '[ 'FieldDef "name" ('TPrimitive T.Text) ]
    , 'DRecord "HelloResponse" '[ 'FieldDef "message" ('TPrimitive T.Text) ] ]

type QuickstartService
= 'Service "Greeter"
    '[ 'Method "SayHello" ...
    , 'Method "SayManyHellos" '[]
    '[ 'ArgStream 'Nothing '[] ('FromSchema QuickstartSchema "HelloRequest")]
    ('RetStream ('FromSchema QuickstartSchema "HelloResponse")) ]
```



간소화 버전입니다  
실제로는 조금 다르게 생겼습니다



# Type-level Programming

- Haskell에서 type 구문은 `type` alias입니다
- 그래서 뒤에 있는 `'[ 'DRecord "HelloRequest" '[ ...` 은 `Type`입니다
- 네? 우리가 아는 `String`, `Int` 같은 `Type`이라고요? 네, 맞습니다
- Haskell `Type-level`으로 표현한 `Type`입니다

```
type QuickstartSchema
  = '[ 'DRecord "HelloRequest" '[ 'FieldDef "name"      ('TPrimitive T.Text) ]
    , 'DRecord "HelloResponse" '[ 'FieldDef "message" ('TPrimitive T.Text) ] ]
```

# Value(data)와 Type

- Haskell은 data 구문으로 Value 생성자들로 타입을 만들 수 있습니다
- False Value 생성자로 생성한 Value은 Bool Type입니다

```
data Bool = False | True
```

```
> :type False  
False :: Bool
```

Haskell REPL인 GHCi에서 확인

# Type과 Kind

- Value가 Type에 속하는 것처럼 Type도 Kind에 속합니다
- 그럼 Bool Type은 어떤 Kind에 속할까요?

```
> :kind Bool
```

```
Bool :: Type
```

NoStartIsType 를 켜서 \* 대신 Type으로 표시됩니다



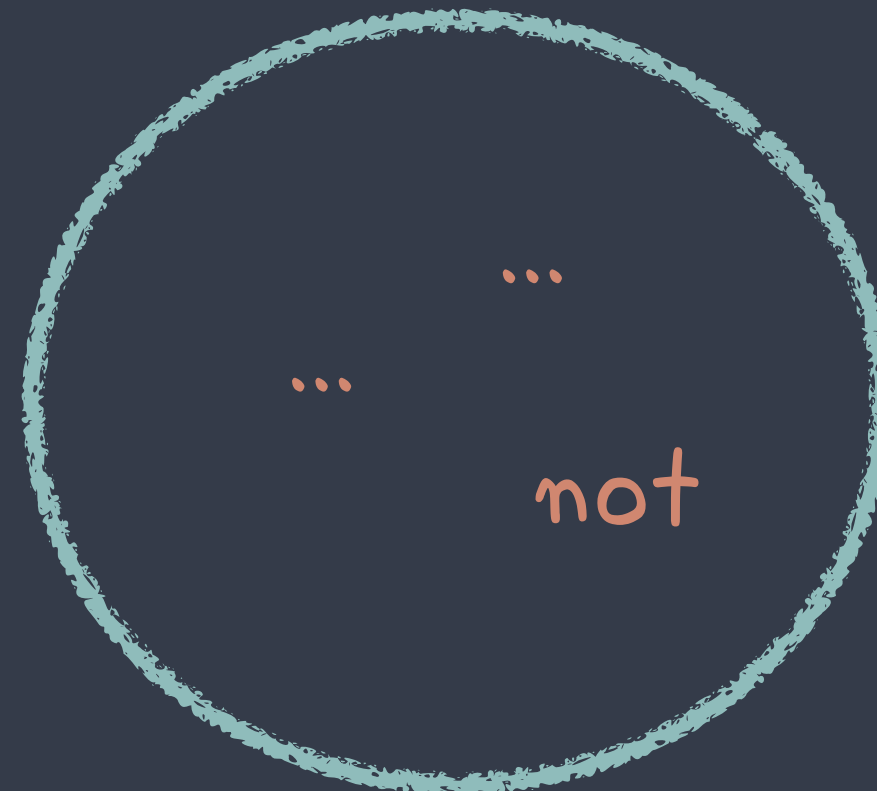
# 함수(Value)와 Type 생성자(Type)

- 함수는 Value이기 때문에 Type에 속합니다
- Type 생성자는 Type이기 때문에 Kind에 속합니다

```
> :type not  
not :: Bool → Bool
```

```
> :kind Maybe  
Maybe :: Type → Type
```

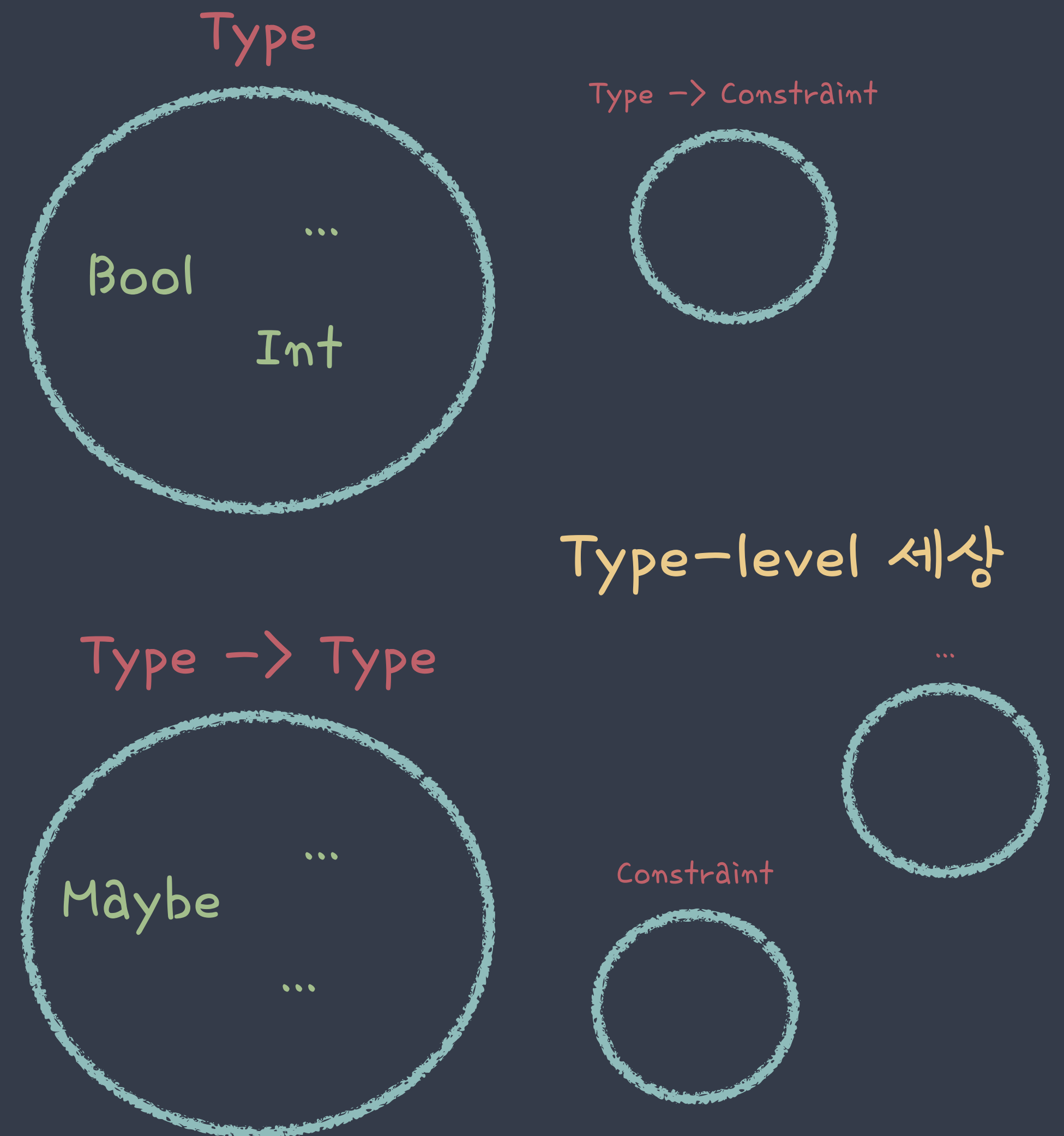
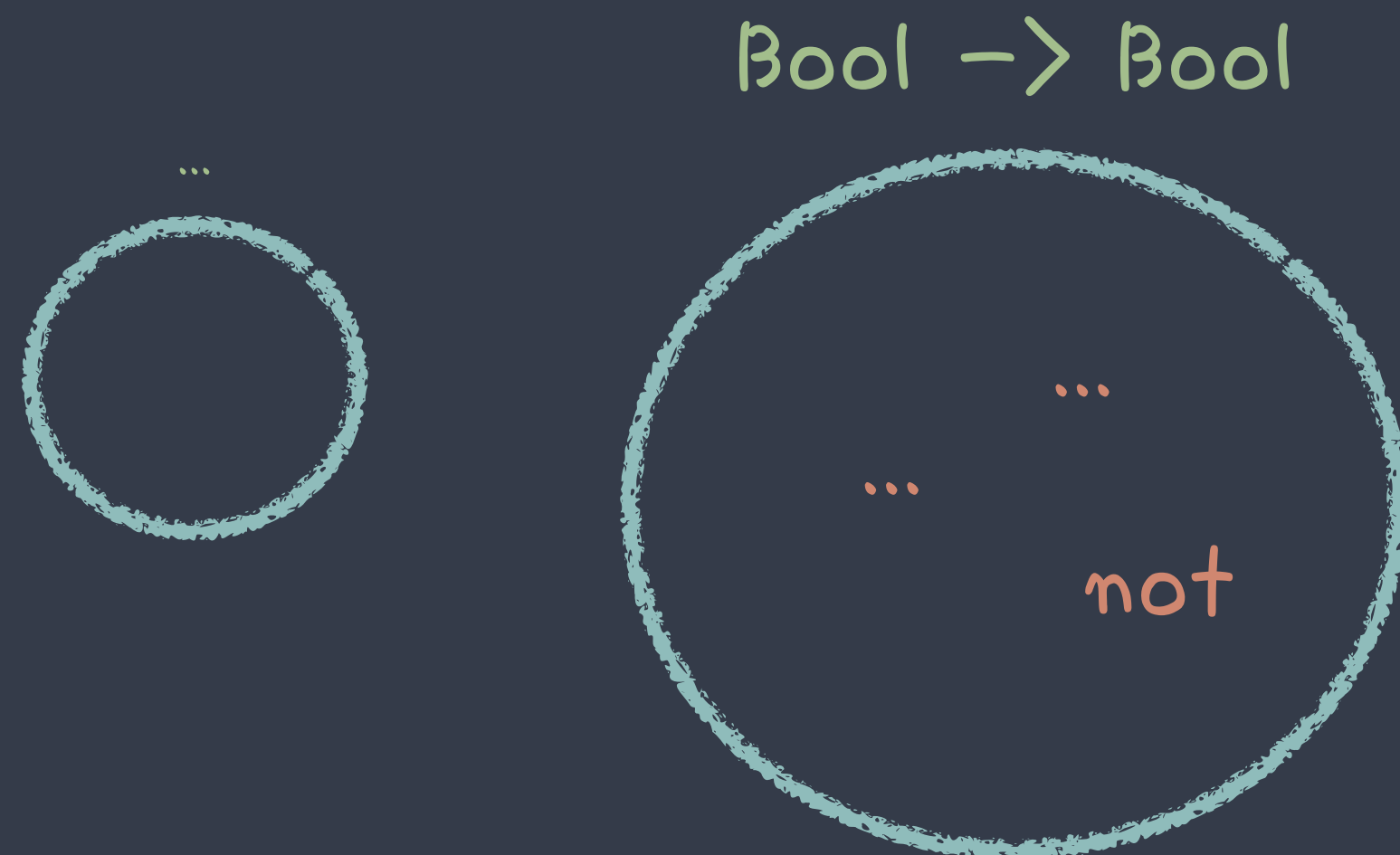
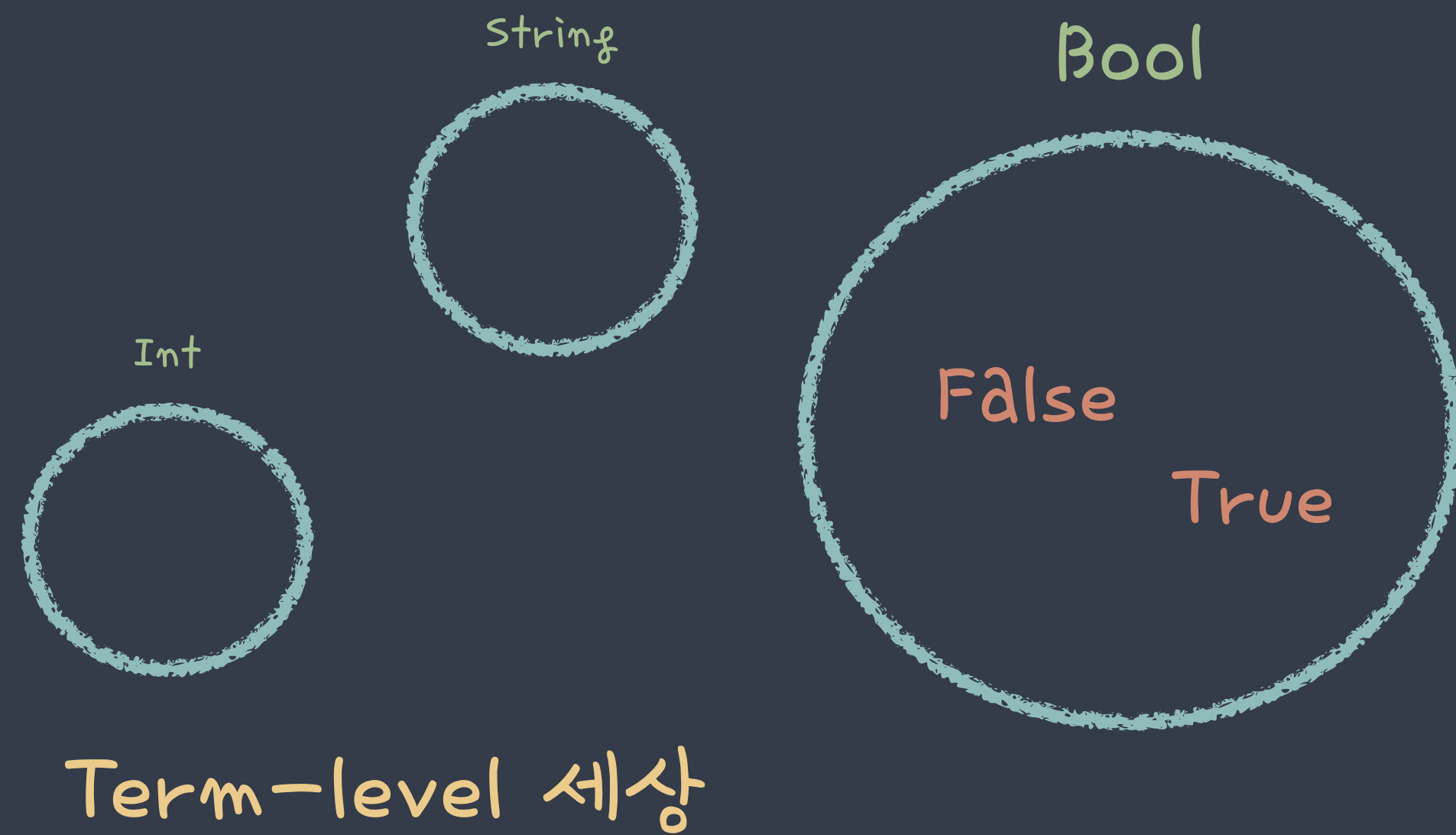
Bool → Bool



Type → Type



# Term-level과 Type-level 세상



# 그럼 Kind는 어디에 쓰나요?

- Type은 어디에 쓰나요? *Term level* 두뇌 🤔
  - 임의의 Value을 Type으로 컴파일 시점에 제한
  - 임의의 Value에 대한 Meta data - 코드를 읽기 위한
- Kind는 어디에 쓰나요? *Type level* 두뇌 🤯
  - 임의의 Type을 Kind로 컴파일 시점에 제한
  - 임의의 Type에 대한 Meta data - 코드를 읽기 위한 + 런타임 활용?

# 임의의 Type에 대한 Kind 제한

- 임의의 Type을 하나 받는 Haskell Type class

```
class Semigroup a where  
  (◇) :: a → a → a  
  ...
```

어떤 Kind를 받을 수 있나?

```
class Functor f where  
  fmap :: (a → b) → f a → f b  
  ...
```

어떤 Kind를 받을 수 있나?



# 임의의 Type에 대한 Kind 제한

- 임의의 Type을 하나 받는 Haskell Type class

```
class Semigroup a where  
  (◇) :: a → a → a  
  ...
```

Type Kind를 받을 수 있음

```
class Functor f where  
  fmap :: (a → b) → f a → f b  
  ...
```

Type → Type Kind를 받을 수 있음

```
> :k Semigroup  
Semigroup :: Type → Constraint
```

```
> :k Functor  
Functor :: (Type → Type) → Constraint
```

명시적인 Kind 제약이 없지만 타입 추론에 따라  
f는 Type → Type이라는 것을 알고 있음

# Kind에 대한 명시적 제약

- PolyKinds 언어 확장으로 Type 변수에 Kind 제약을 줄 수 있음
- UnitName Type class에 u 타입 변수는 Type Kind여야 함

```
class UnitName (u :: Type) where  
  ...
```

- Term Type 생성자에 s 타입 변수는 Schema Kind여야 함

```
data Term (s :: Schema) where  
• Sc | ...
```

처음 보는데요?

# 새로운 Kind 만들기

- 새로운 타입을 만들 수 있는 것처럼 새로운 Kind도 만들 수 있습니다
- Haskell에서 Value 생성자로 Type을 만들 때 data 구문을 사용합니다

```
data Bool = False | True
```

- 역시 Type 생성자로 Kind를 만들 때도 data 구문을 사용합니다.
- **DataKinds** 언어 확장을 활성화 하면 됩니다.

```
{-# LANGUAGE DataKinds #-}
```

```
data Bool = False | True
```

# 새로운 Kind 만들기

- 아래 구문은 False, True Value 생성자와 Bool Type을 만듭니다
- 그리고 False, True Type 생성자와 Bool Kind도 함께 만듭니다
- Value 생성자인 False와 Type 생성자인 False의 혼란을 막기 위해 '를 사용합니다


```
{-# LANGUAGE DataKinds #-}
```

```
data Bool = False | True
```

```
> :type False  
False :: Bool
```

```
> :kind 'False  
'False :: Bool
```

'를 붙여 Type(생성자)임을 표시합니다  
Promoted Type이라고 합니다



# 너무 많이 왔습니다! 다시 mu-haskell을 봅시다

- 이제 ' 표시의 의미와 DRercord, FieldDef 같은 것이 **Type**이라는 것을 알 수 있습니다

```
type QuickstartSchema
= '[ 'DRecord "HelloRequest" '[ 'FieldDef "name" ('TPrimitive T.Text) ]
    , 'DRecord "HelloResponse" '[ 'FieldDef "message" ('TPrimitive T.Text) ] ]
```

- 그래도 "HelloRequest"와 '[] 리스트? 같은 것이 **Type**이라는 것은 이상하네요

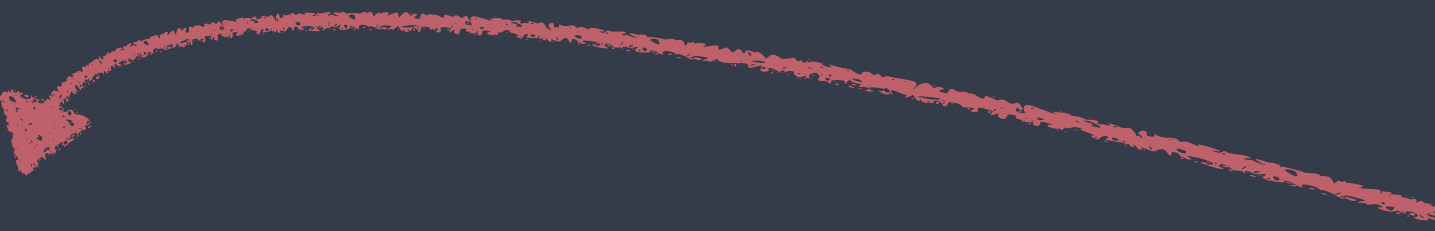
# Type-level literals

- DataKinds는 숫자와 문자열 Value를 Type으로 promotion 시켜줍니다
- 따라서 1, 2, 3, "hello", "world"와 같은 것을 Type으로 쓸 수 있습니다
- 이런 타입을 Type-level literals라고 합니다
- 1, 2, 3, "hello", "world"가 Type이라면 어떤 Kind에 속하나요?

```
> :kind 1
1 :: GHC.Types.Nat

> :kind "hello"
"hello" :: GHC.Types.Symbol

> :kind "HelloRequest"
"HelloRequest" :: GHC.Types.Symbol
```



정말 타입이네요!

' 표시를 안 해도 되나요?

모호하지 않은 경우에는 '를 붙이지 않아도 됩니다.

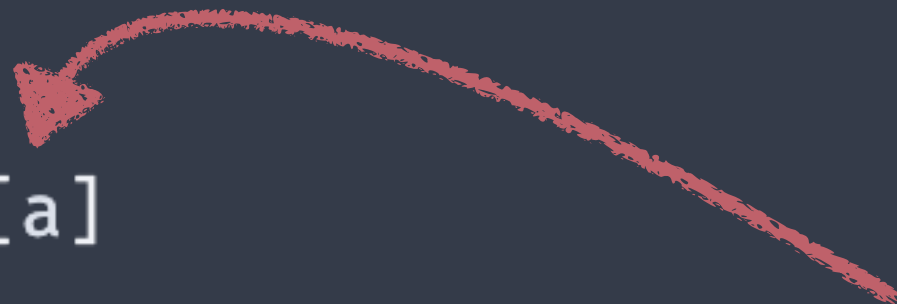
:kind는 뒤에 Type이 와야 하기 때문에

"hello"가 Value가 아닌 Type이란 것을 추론합니다

# 리스트 Type-level literal

- [] 정의는 다음과 같고 DataKinds를 사용하면 promoted Type이 생깁니다

```
data [] a = [] | a : [a]  
  
> :kind [Int, String, Bool]  
[Int, String, Bool] :: [Type]
```



term-level에서는 a에 Value를  
type-level에서는 a에 Type을  
넘깁니다



# 간단하게 mu-haskell 스키마 타입 선언해보기

- 이제 mu-haskell이 생성한 스키마를 선언해 볼 수 있습니다 (간소화 했습니다)

```
type Schema = '[ 'DRecord "HelloRequest" '[ 'FieldDef "name" ]  
|               , 'DRecord "HelloResponse" '[ 'FieldDef "message" ] ]
```

- FieldDef Type은 FieldDef Kind에 속하고 DRecord는 TypeDef Kind에 속합니다

```
data FieldDef fieldName = FieldDef fieldName
```

- 0 | data TypeDef typeName fieldName = DRecord typeName [FieldDef fieldName]

# Schema의 값은 어떻게 표현하나요?

- HelloRequest만 따로 보면 실제 Text Type의 값을 하나 갖습니다
- 따라서 HelloRequest에 들어가는 값은 문자열입니다
- 개념을 간소화하기 위해 FieldDef의 값을 어떻게 갖을 수 있는지 봅시다

```
type QuickstartSchema
  = '[ 'DRecord "HelloRequest" '[ 'FieldDef "name" ('TPrimitive T.Text) ]
    , 'DRecord "HelloResponse" '[ 'FieldDef "message" ('TPrimitive T.Text) ] ]
```

# 간단한 FieldDef 스키마와 값

- 함수는 Type Kind 값을 받기 때문에 값을 사용하려면 Type Kind 안에 넣습니다
- 중요한 부분은 FieldDef에 정의한 타입이 실제 값의 타입이 되도록 값 생성자 Term이 생성하는 타입에 지정한 부분입니다

```
data FieldDef = FieldDef Symbol Type

data Term (s :: FieldDef) where
  Term :: v → Term ('FieldDef name v)

type HelloRequestSchema = ('FieldDef "name" String)

value :: Term HelloRequestSchema
value = Term "김은민"
```

Term Value 생성자는

`FieldDef name v`

v는 인자와 같은 타입이어야 합니다

# Mu-Haskell Type-level 스키마가 어떻게 더 안전한가요?

- 만약 HelloRequestSchema와 실제 값이 다르면 컴파일 에러가 납니다
- 다시 말하지만 외부에서 들어오는 값에 대한 검증은 필요합니다
- Mu-Haskell 프레임워크 안에서 처리됩니다
- 코드 레벨에서는 처리할 필요가 없습니다

```
type HelloRequestSchema = ('FieldDef "name" String)
```

```
invalidValue :: Term HelloRequestSchema
```

```
invalidValue = Term 42
```

```
data Term (s :: FieldDef) where
```

```
  Term :: v → Term ('FieldDef name v)
```

v Type이 일치하지 않습니다

FieldDef "name" Int 인데

이 값은 HelloRequestSchema 인

FieldDef "name" String이어야 합니다

# Type-level literals 사용하기

- Type-level literal 정보는 런타임에 사용할 수 있습니다
- mu-haskell도 Type-level literal 정보를 사용해 GraphQL introspection 같은 기능을 제공합니다.
- 간단한 예를 통해 Type-level literal을 어떻게 사용하는지 봅시다
- 다음은 문자열 길이가 최대 10개로 제한된 LimitedString10 Type입니다

```
data LimitedString10 = LimitedString10 String
```

```
mkLimitedString10 :: String → LimitedString10
```


```
mkLimitedString10 s = assert (length s ≤ 10) $ LimitedString10 s
```

# Type-level literals 사용하기

- 10 글자가 아닌 타입이 필요하면 새로 만들어야 합니다
- 함수에 길이를 받게하면 타입 안전하지 않습니다

```
data LimitedStringN = LimitedStringN String

mkLimitedStringN :: Int → String → LimitedStringN
mkLimitedStringN limit s = assert (length s ≤ limit) $ LimitedStringN s
```



mkLimitedStringN 10과  
mkLimitedStringN 100은  
같은 LimitedStringN 타입입니다



# Type-level literals 사용하기

- 길이 제한을 타입 변수 정보에서 가져옵니다
- 타입 변수는 Nat Kind의 Type으로 제한되어 있습니다

```
data LimitedString limit = LimitedString String
```

```
mkLimitedString :: forall limit. KnownNat limit => String -> LimitedString limit
```

```
mkLimitedString s = assert (toInteger (length s) <= natVal (Proxy :: Proxy limit)) $ LimitedString s
```

```
> mkLimitedString "abc" :: LimitedString 3
LimitedString "abc"
> mkLimitedString "abc" :: LimitedString 2
*** Exception: Assertion failed
```

LimitedString 10과

LimitedString 100은 다른 타입입니다



# Summary

- mu-haskell로 Haskell에서 GraphQL, gRPC 서버, gRPC Client를 만들 수 있다
- mu-haskell은 Protocol 포맷과 언어 타입간 의존성을 줄이기 위해 중간 Type을 사용한다
- 중간 Type은 Type-level Format으로 되어 있다
- Type-level Format으로 타입 안정성 가진다
- Type-level literals를 사용해 Type에 추가 정보를 런타임에 활용 할 수 있다
- 이것은 Type-level programming의 맛보기이다
- Constacts는 Full-time Haskell 개발자를 구인 중이다

## 더 궁금하시면

- Haskell in depth - Vitaly Bragilevsky (11~13 Chapter)
- Basic Type Level Programming in Haskell - Matt Parsons
- The Inner Workings of Mu-Haskell - by Alejandro Serrano
- GHC Language extensions (절반 이상이 type-level에 관련된 것)
- Constacts에 지원하기!