Catching More Bugs at Compile Time

Using Phantom Types with Implicits

Jacob Van De Weert (he/him)  Maryam Aly (she/her)
@jvandeweert  @maryamaaly
Problem:

Programmers need access to a database, using their favorite language

- Easy to construct typesafe data models
- Much harder to encode query rules and logic
- How do we catch logic errors in db queries?
Problem

How do we catch logic errors in db queries?

Venue
- `.where(_.mayor eqs 1234)`
- `.limit(10)`
- `.fetch(100)`

Checkin
- `.where(_.shout eqs "hello NEScala!")`
- `.modify(_.likes inc 1)`
- `.updateOne()`

User
- `.where(_.id eqs 5678)`
- `.select(_.name)`
- `.modify(_.email setTo "you@me.com")`
- `.updateMulti()`
Traditional Approach

- Invoke validation logic *somewhere*
  - Query builder
  - Query executor
  - Worst case, database server (hopefully!)

- Very flexible, but happens at runtime

- Can we move at least some query validation into the compiler?
  - How do we recognize query logic at compile time?
case class QueryBuilder[MetaRecord, Result, +State]

- **MetaRecord** - database model to query
- **Result** - query return type
- **State**?
  - Not meaningful to programmers, and usually invisible outside the query library itself
  - Hence "phantom" type
  - Capture and enforce attributes of the query under construction
    - HasQuantity, Selected, ShardAware, etc
State Attributes

trait HasQuantity
trait Unquantified
trait Quant extends HasQuantity with Unquantified

trait Selected
trait SelectedOne extends Selected
trait Unselected
trait Sel extends Selected with SelectedOne with Unselected

trait ShardKeyNotSpecified
trait ShardAware
trait ShardKeySpecified extends ShardAware
trait AllShardsOk extends ShardAware
trait Sh extends ShardKeyNotSpecified with ShardKeySpecified with AllShardsOk
trait HasQuantity
trait Unquantified
trait Quant extends HasQuantity with Unquantified

trait Selected
trait SelectedOne extends Selected
trait Unselected
trait Sel extends Selected with SelectedOne with Unselected

trait ShardKeyNotSpecified
trait ShardAware
trait ShardKeySpecified extends ShardAware
trait AllShardsOk extends ShardAware
trait Sh extends ShardKeyNotSpecified with ShardKeySpecified with AllShardsOk
A <: B → "A is a subtype of B", A :> B → "A is a supertype of B"

Subtypes are "more specific" than their supertypes
   o Eg. Int is a subtype of Any
   o For traits, A with B is a subtype of A

MyType[+T] → "T is a covariant type parameter of MyType"
   o MyType[Int] is a subtype of MyType[Any]

MyType[-T] → "T is a contravariant type parameter of MyType"
   o MyType[Any] is a subtype of MyType[Int]

https://docs.scala-lang.org/tour/variances.html
Implicit method parameters
- Optionally specified by the programmer, or filled in by the compiler
- `def doSomething[T](n: Int)(implicit myType: MyType[T]): Int = ...`

Implicit values
- Provide an implicit method parameter
- `implicit val magic: Int = 5`

Implicit methods
- Provide an implicit value
- `implicit def getMyType[T]: MyType[T] = ...`

https://docs.scala-lang.org/tour/implicit-parameters.html
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that $A <: B")

trait Required[-A, +B]

object Required {
    implicit def conforms[A]: Required[A, A] = null
}
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that $\{A\} <: \{B\}\)

trait Required[-A, +B]

object Required {
  implicit def conforms[A]: Required[A, A] = null
}

Semantics: A must be a sub-type of B
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that $\{A\} <: \{B\}"")

trait Required[-A, +B]

object Required {

  implicit def conforms[A]: Required[A, A] = null

}

Semantics: A must be a sub-type of B

But only one type param, huh?
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that ${A} <: ${B}"")
trait Required[-A, +B]
object Required {
    implicit def conforms[A]: Required[A, A] = null
}

@implicitNotFound(msg = "Query must be Unquantified, but it's actually ${In}")
trait AddQuantity[-In, +Out] extends Required[In, Unquantified]
object AddQuantity {
    implicit def addQuantity[
        Rest >: Sel with Sh
    ]: AddQuantity[Rest with Unquantified, Rest with HasQuantity] = null
}
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that $\{A\} <: $\{B\}"")
trait Required[-A, +B]
object Required {
    implicit def conforms[A]: Required[A, A] = null
}

@implicitNotFound(msg = "Query must be Unquantified, but it's actually $\{In\}"")
trait AddQuantity[-In, +Out] extends Required[In, Unquantified]
object AddQuantity {
    implicit def addQuantity[
        Rest >: Sel with Sh
    ]: AddQuantity[Rest with Unquantified, Rest with HasQuantity] = null
}
Enforcing State Attributes via Implicits

@implicitNotFound(msg = "Cannot prove that \$\{A\} <: \$\{B\}"")
trait Required[-A, +B]
object Required {
  implicit def conforms[A]: Required[A, A] = null
}

@implicitNotFound(msg = "Query must be Unquantified, but it's actually \$\{In\}"
trait AddQuantity[-In, +Out] extends Required[In, Unquantified]
object AddQuantity {
  implicit def addQuantity[
    Rest >: Sel with Sh
  ]: AddQuantity[Rest with Unquantified, Rest with HasQuantity] = null
}
case class QueryBuilder[MetaRecord, Result, +State](
  ...,
  lim = None,
  ...
)

  def limit[NewState](n: Int)(
    implicit ev: AddQuantity[State, NewState]
  ): QueryBuilder[MetaRecord, Result, NewState] = {
    this.copy(lim = Some(n))
  }

}
Enforcing State Attributes via Implicits

```scala
case class QueryBuilder[MetaRecord, Result, +State](
    ...,
    lim = None,
    ...
) {
    def limit[NewState](n: Int)(
        implicit ev: AddQuantity[State, NewState]
    ): QueryBuilder[MetaRecord, Result, NewState] = {
        this.copy(lim = Some(n))
    }
}
```

---

AddQuantity implicit updates builder State with HasQuantity after calling .limit!
Questions?

Let’s take make sure we’re all on the same page before moving on.

Now, how do we prevent this?

Venue
  .where(_.mayor eqs 1234)
  .limit(10)
  .fetch(100)
Now, how do we prevent this?

Venue
.where(_.mayor eqs 1234)
.limit(10)
.fetch(100)

The builder can track when fetch and limit are called and enforce that at most one is called at a time.
Example: Not Already of Type

sealed trait NotAlreadyQuantified[-A]
object NotAlreadyQuantified {
  implicit def notAlready[A](implicit ev: RequiredNot[A, HasQuantity]):
      NotAlreadyQuantified[A] = null
}
Example: Not Already of Type

We invoke the implicit ev by instantiating a NotAlreadyQuantified object. This instantiates a RequiredNot.
Example: Not Already of Type

```scala
sealed trait RequiredNot[A, B]
object RequiredNot {
  implicit def sub1[A, B >: A]: RequiredNot[A, B] = null
  implicit def sub2[A, B >: A]: RequiredNot[A, B] = null
  implicit def anyOther[A, B]: RequiredNot[A, B] = null
}
```

Example: Not Already of Type

```scala
sealed trait RequiredNot[A, B]
object RequiredNot {
  implicit def sub1[A, B >: A]: RequiredNot[A, B] = null
  implicit def sub2[A, B >: A]: RequiredNot[A, B] = null
  implicit def anyOther[A, B]: RequiredNot[A, B] = null
}
```

Invoking the implicit instantiates a RequiredNot object, which can’t be resolved if A is a subtype of B.
Example: Not Already of Type

@implicitNotFound(
  msg = "Your builder is setting the number of returned results more than once!"
)
sealed trait NotAlreadyQuantified[-A]
object NotAlreadyQuantified {
  implicit def notAlready[A](implicit ev: RequiredNot[A, HasQuantity]):
    NotAlreadyQuantified[A] = null
}
Example: Not Already of Type

This displays the failure message as the compile error to the programmer.

```scala
@implicitNotFound(
  msg = "Your builder is setting the number of returned results more than once!"
)
sealed trait NotAlreadyQuantified[-A]
object NotAlreadyQuantified {
  implicit def notAlready[A](implicit ev: RequiredNot[A, HasQuantity]):
    NotAlreadyQuantified[A] = null
}
```
Example: Not Already of Type

sealed trait RequiredNot[A, B]
object RequiredNot {
  implicit def sub1[A, B :> A]: RequiredNot[A, B] = null
  implicit def sub2[A, B :> A]: RequiredNot[A, B] = null
  implicit def anyOther[A, B]: RequiredNot[A, B] = null
}

@implicitNotFound(msg = "Your builder is setting the number of returned results more than once!")
sealed trait NotAlreadyQuantified[-A]
object NotAlreadyQuantified {
  implicit def notAlready[A](implicit ev: RequiredNot[A, HasQuantity]):
    NotAlreadyQuantified[A] = null
}

Example: Not Already of Type

```scala
def limit[NewState](count: Int)(
  implicit addQuantity: AddQuantity[State, NewState]
  checkNotAlreadyQuantified: NotAlreadyQuantified[State]
): QueryBuilder[MetaRecord, Result, NewState] = {
  ...
}
```
Example: Not Already of Type

We’re adding the HasQuantity phantom type to keep track of this kind of query attribute.

def limit[NewState](count: Int)(
    implicit addQuantity: AddQuantity[State, NewState]
    checkNotAlreadyQuantified: NotAlreadyQuantified[State]
): QueryBuilder[MetaRecord, Result, NewState] = {
    ...
}
We’re checking to make sure that this type of query attribute wasn’t already added.

def limit[NewState](count: Int)(
    implicit addQuantity: AddQuantity[State, NewState]
    checkNotAlreadyQuantified: NotAlreadyQuantified[State]
): QueryBuilder[MetaRecord, Result, NewState] = {
    ...
}
Example: Not Already of Type

def fetch[NewState](count: Int)(
    implicit addQuantity: AddQuantity[State, NewState]
    checkNotAlreadyQuantified: NotAlreadyQuantified[State]
): QueryBuilder[MetaRecord, Result, NewState] = {
    ...
}

def limit[NewState](count: Int)(
    implicit addQuantity: AddQuantity[State, NewState]
    checkNotAlreadyQuantified: NotAlreadyQuantified[State]
): QueryBuilder[MetaRecord, Result, NewState] = {
    ...
}
Getting More Complex

What if you wanted some functions to be conditionally required?
Getting More Complex

Let’s go through an example where, for performance reasons, you want to enforce that every sharded query has a limit on the number of results returned.
Conditionally Required

```
type Sharded
  with HasQuantity
  with HasShardKey
```

New type that links the required concepts together
Conditionally Required

```scala
type Sharded
  with HasQuantity
  with HasShardKey

def shardKey[NewState](key: Long)(
  implicit addShardKey: AddShardKey[State, NewState]
  checkNotAlreadySharded: NotAlreadySharded[State]
): QueryBuilder[Goal with Sharded, NewState] = {
  ...
}
```

Keep track of when shardKey is called in the type state.
Conditionally Required

type Sharded
  with HasQuantity
  with HasShardKey

def shardKey[NewState](key: Long)(
  implicit addShardKey: AddShardKey[State, NewState]
  checkNotAlreadySharded: NotAlreadySharded[State]
): QueryBuilder[Goal with Sharded, NewState] = {
  ...
}

Make sure the programmer only calls shardKey once.
Conditionally Required

```java
type Sharded
    with HasQuantity
    with HasShardKey

def shardKey[NewState](key: Long)(
    implicit addShardKey: AddShardKey[State, NewState]
    checkNotAlreadySharded: NotAlreadySharded[State]
): QueryBuilder[Goal with Sharded, NewState] = {
    ...
}
```

Track an end goal type on the QueryBuilder
Enforce that all of the requirements are satisfied when you convert the builder to built object.

```scala
def shardKey[NewState](key: Long)(
    implicit addShardKey: AddShardKey[State, NewState]
    checkNotAlreadySharded: NotAlreadySharded[State]
  ):QueryBuilder[Goal with Sharded, NewState] = {
  ...
}

def result()(
    implicit checkAllRequiredAreSet: Equals[Goal, State]
  ): Query = {
    ...
}
Conditionally Required

```scala
type Sharded
    with HasQuantity
    with HasShardKey

def shardKey[NewState](key: Long)(
    implicit addShardKey: AddShardKey[State, NewState]
    checkNotAlreadySharded: NotAlreadySharded[State]
): QueryBuilder[Goal with Sharded, NewState] = {
...
}

def result()
    implicit checkAllRequiredAreSet: Equals[Goal, State]
): Query = {
...
}
```
How else can you use this pattern?

Creating a type-safe QueryBuilder

Building a Localization Library

Twitter Finagle's ClientBuilder and ServerBuilder

The scala standard library! (kinda)

```scala
trait IterableOnceOps[+A, +CC[_], +C] extends Any { this: IterableOnce[A] =>
  def toMap[K, V](implicit ev: A <:< (K, V)): immutable.Map[K, V] = ...
}
```
Thank You!
Questions?